Benchmarking OpenMP Programs for Real-Time Scheduling

Yang Wang[1], Nan Guan[2], Jinghao Sun[1], Mingsong Lv[1], Qingqiang He[1], Tianzhang He[1], Wang Yi[1,3]

[1]Northeastern University, China
[2]Hong Kong Polytechnic University, China
[3]Uppsala University, Sweden

***Abstract*—Real-time systems are shifting from single-core to multi-core processors. Software must be parallelized to fully utilize the computation power of multi-core architecture. OpenMP is a popular parallel programming framework in general and high-performance computing, and recently has drawn a lot of interests in embedded and real-time computing. Much recent work has been done on real-time scheduling of OpenMP-based parallel workload. However, these studies conduct evaluations with randomly generated task systems, which cannot well represent the structure features of OpenMP workload. This paper presents a benchmark suite, ompTGB, to support research on real-time scheduling of OpenMP-based parallel tasks. ompTGB does not only collect realistic OpenMP programs, but also models them into task graphs so that the real-time scheduling researchers can easily understand and use them. We also present a new response time bound for a subset of OpenMP programs and use it to demonstrate the usage of ompTGB.**

I. Introduction

Multi-cores are more and more widely used in real-time systems to meet the rapidly increasing requirements in high performance and low power consumption. To fully utilize the computation power of multi-core processors, software must be parallelized. OpenMP [1] is a parallel programming framework widely used in general and high-performance computing. Recently, there have been increasing interests to use OpenMP in embedded and real-time computing [2]–[5].

A fundamental problem in real-time system design is how to schedule the workload to satisfy the timing constraints. A common way to model parallel software system is using graphs, and there have been much work in the area of real-time scheduling for graph-based parallel task models [5]–[12]. The execution semantics of OpenMP are closely related to the graph-based task models, and this motivates many recent work in real-time scheduling of parallel workload. In particular, some work has extended the traditional graph-based task models to include some unique features introduced by the OpenMP semantics. For example, recent work on real-time scheduling of conditional DAG models [13] is motivated by the branching structures (e.g., **if-else**) in OpenMP programs. Another example is the concept of TSP (Task Scheduling Points) in OpenMP, which motivates the work on real-time scheduling of DAG models with limited preemption points [14].

978-1-5386-1898-1/17/$31.00

As real-time scheduling of OpenMP-based parallel workload becomes a hot research topic, we are facing a significant problem: how to evaluate the scheduling algorithms and analysis techniques. The real-time scheduling research community has a tradition to use randomly generated task sets for performance evaluation, for not only the simple periodic/sporadic task models, but also more complex graph-based task models where both the timing parameters and the graph structures are randomly generated [15]. Unfortunately, this is not suitable when we are scheduling OpenMP-based workload. OpenMP is a programming framework, and the corresponding task graph models should have certain structure features, which cannot be well captured by randomly generated task graphs. Performance evaluations with randomly generated tasks can be very biased: a scheduling/analysis algorithm performing well in evaluations with randomly generated task sets may not be suitable to realistic OpenMP programs. Without a fair performance evaluation methodology, it will be difficult to conduct meaningful research on real-time scheduling of OpenMP programs.

The target of this work is to provide a benchmark suite, **ompTGB** (OpenMP Task Graph Benchmarks), to fill the gap between theoretical real-time scheduling research and the OpenMP software reality. **ompTGB** is not a simple collection of OpenMP programs. Instead, it is prepared in a way that the real-time scheduling researchers can easily understand and use them to evaluate their scheduling and analysis algorithms. More specifically, in **ompTGB** we

- model OpenMP programs as task graphs annotated with information relevant to real-time scheduling,
- develop a tool to transfer OpenMP programs into DAGs, and
- provide a collection of task graphs generated from realistic OpenMP programs.

Apart from introducing **ompTGB**, this work also develops new response time bounds for a subset of OpenMP programs and use these results to demonstrate the usage of **ompTGB**. In previous work, the response time of these programs are considered as unbounded [4]. In this paper, we illustrate how the task graph modeling provided by **ompTGB** helps us to find the key features of their workload structures and thus bound their response times, and how the benchmarks are used

to evaluate our theoretical results.

II. Related Work

Much work has been done on scheduling DAG-based parallel real-time task systems on multi-core processors [5]–[12]. The task models in these papers are closely related to the workload model of OpenMP, but missing many features in realistic OpenMP programs. Recently, some of these features have been taken into consideration. Motivated by the concept of Task Scheduling Points (TSP) in OpenMP, some work has been done on the scheduling of parallel tasks with limited preemption points [14]. Motivated by the branching structures of OpemMP programs, the parallel real-time task models have been extended to combine the fork-join and conditional semantics [13], [16].

Very recently, [3], [4] studied the possibility to apply OpenMP to real-time systems mainly from the real-time scheduling perspective. These work highlighted some important features in OpenMP that are relevant to real-time scheduling. In particular, they discussed how Task Scheduling Points (TSP) and Task Scheduling Constraints (TSC) affect the real-time scheduling behavior. However, [3], [4] didn't consider the branching structures when modeling OpenMP-based workload. In contrast, the modeling of OpenMP programs in our paper includes the branching structures (and the `final` and `if` directives that leads to branching semantics).

[4] also studied the problem of bounding the response time (makespan) of the (non-recurring) task system generated by an OpenMP application on multi-cores. [4] developed response time bounds for the case containing only `untied` tasks, and claimed that bounding the response time in the presence of `tied` tasks is inherently hard. However, in this paper we will show that for a subset of OpenMP programs that contains `tied`, the response times can also be well bounded as for the case having only `untied` tasks.

There are several source-code level benchmark suits for real-time systems [17]–[21], which are all with sequential programs and serve for the purpose of WCET analysis. To the best of our knowledge, the **opmTGB** benchmark suite presented in this paper is the first one oriented to real-time scheduling of parallel software.

III. Overview

We first introduce our tool to transform OpenMP program source codes into task graph models, then brief the OpenMP program collection in **ompTGB**.

A. Transformation Tool

Figure. 1 shows the architecture of the transformation tool. The light blue boxes are existing tools utilized by us, and the dark blue boxes are functionalities developed by us. Currently our transformation tool can only process OpenMP programs written in C.

For lexical and parsing analysis of OpenMP programs, we use Lex & Yacc [22], which is embedded in ompi, a lightweight, open source OpenMP compiler system for C

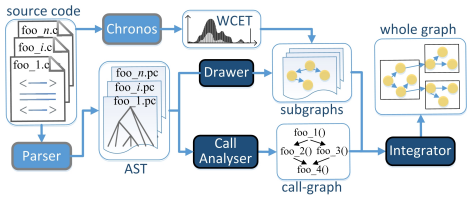

Fig. 1. Architecture of the transformation tool.

programs. The output of the parser are abstract syntax trees (AST), which store useful abstract syntactic structures of the programs. The AST is used (1) by the Drawer to construct the task graph models of individual functions, and (2) by the Call Analyzer to generate call-graphs. The integrator combines both of them to generate the task graph models of the whole application, which are stored in the DOT formate [23] and are visualized by Graphviz [24]. Details about the task graph model will be presented in the next section.

Besides the topology of task graphs, the weight of each vertex, representing its worst-case execution time (WCET), is also important information when studying scheduling problems. The WCET of a vertex heavily depends on the underlying hardware architecture. However, **ompTGB** is a hardware independent benchmark, so it is impossible to provide precise WCET information of OpenMP programs in **ompTGB**. Nevertheless, to reduce the burden of users for whom the precise WCET estimation is not a critical issue, we provide reference weight values for each vertex. We do not claim these reference values to be able to precisely characterize the WCET on any particular hardware platform, and expect the users to estimate the WCET values by themselves whenever necessary. We provide two types of reference weight values obtained by the following two approaches:

- **Static Analysis.** The first type of reference values are obtained using the static WCET analysis tool Chronos [25]. Chronos uses the SimpleScalar as the underlying processor architecture, and safely bounds the WCET of a C program executing on SimpleScalar simulator with modeling and analysis the timing effect of hardware components such as caches and pipelines. We use Chronos to compute a safe WCET bound for the codes in each individual vertex. The strength of this approach is to provide *safe* WCET estimations, while the weakness is assuming oversimplified underlying hardware architecture.
- **Measurement.** We also measure the execution time for the vertices by executing the programs on the real hardware. We instrument the OpenMP programs with instructions reading the timer at the beginning and end of each vertex, and the execution time of the vertex is the difference between the two time stamps. Although this approach cannot provide strictly safe WCET bounds, these reference WCET values give a rough idea of the workload of each vertex. In particular, currently the reference values provided by this approach are obtained on an Intel i7-4770 CPU with 3.5GHZ and 8192KB cache

size.

B. Benchmarks

We collected C based OpenMP applications from several benchmarks that consist of `task` directives. Table. I summarizes these applications and shows their structure features. Note that **ompTGB** does *not* provide the source codes of these OpenMP programs[1]. Instead, we only provide the corresponding task graph models and the reference WCET values.

The first column in Table. I gives the name of each application, and the second column shows which benchmark suite an application belongs to. The third column reports the code size, i.e., the lines of code (LOC). Here we use the Unified Code-Count tool (UCC) [26] and report the logical SLOC. Columns 4 to 9 show whether or not a application contains certain structure features. Columns 4 to 8 are OpenMP directives, and column 9 is the keyword in the base language C.

All the applications in Table. I contain `tied` tasks. These applications are categorized into two types:

- The first type of applications implement task synchronization using `taskwait` directive. These applications are from benchmarks spec2012, bots-1.1.2, dash-1.0m, ect, which are developed for OpenMP version 3.0.
- The second type of applications implement task synchronization using data dependency. These applications are from kastors-1.1, which is recently developed and based on OpenMP 4.0. (See Section. V-D for details)

The last three columns show the conditional branching structures in the applications. Some applications use `if` and `final` clauses to restrict the execution behavior of tasks as listed in columns 7 and 8 (see in Section. IV-B for details). Moreover, as shown in the last column of Table. 1, almost four-fifths(78%) of the applications contain the condition branches (i.e., `if-else` structure) that are nested with the creation of and synchronization among tasks.

Apart from the benchmark programs listed in Table. 1, we are currently working on collecting more OpenMP programs, especially those are oriented to embedded and real-time systems. In particular, we provide an "one-click" service for code-to-graph transformation on our **ompTGB** website [23], so that users can upload their own OpenMP program source codes and then download the generated task graph models. These user-uploaded programs will be included in our benchmark suite under a specific category.

IV. Modeling

A. Basic Structures

An OpenMP program is modeled as a directed acyclic task graph $G = (V, E)$ with some extra annotations. Each vertex v in V represents a piece of codes, and is associated with a WCET $c(v)$. Several vertices may belong to a *task*, which is represented by the red rectangle in Fig. 2. Each task contains a unique entry vertex and several exit vertices. The vertices

[1]As the benchmarks are proprietary, we do not own the right to distribute these source codes.

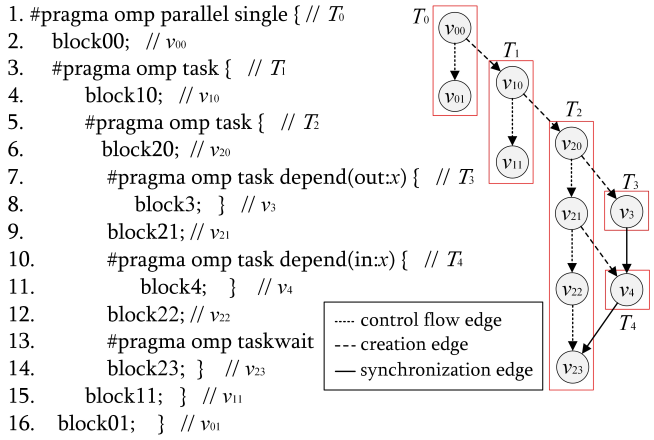

Fig. 2. An example OpenMP program.

in one task are forced to be executed by the same thread unless additionally specified (See `tied` and `untied` tasks in Section. IV-C for more details). There are three types of edges in a graph, i.e., $E = E_1 \cup E_2 \cup E_3$, as detailed in the following.

Control Flow Edges (E_1), denoted by dotted-line arrows in Figure. 2, which model the control flow dependencies, and only connect vertices within the same task.

Task Creation Edges (E_2), denoted by dashed-line arrows in Figure. 2. In OpenMP, a new task is created by the `task` directive (e.g., Line 3 in the code of Figure. 2). The destination vertex of a task creation edge must be the entry vertex of a task. If a vertex in task T_i points to the entry vertex of task T_j with a task creation edge, then T_i is the *parent* of T_j and T_j is a *child* of T_i. Two tasks sharing the same parent are *siblings*.

Synchronization Edges (E_3), denoted by solid-line arrows in Figure. 2. The source vertex of a synchronization edge must be the exit vertex of a task. There are two subtypes of synchronization edges, corresponding to the `taskwait` (e.g., Line 13 in the code of Figure. 2) and `depend` (e.g., Lines 7 and 10 in the code of Figure. 2) directives, respectively:

- `taskwait` blocks the parent task until all of its children created beforehand have been finished. The source of an edge of this type must be the exit vertex of a child task of itself, and the destination can be any vertex in the parent task after creating these children, e.g., edge (v_4, v_{23}) in Figure. 2.
- `depend` imposes an order between two sibling tasks. If a task has an `in` dependence on a variable, it cannot start execution until all its sibling tasks with an `out` or `inout` dependence on the same variable have been completed. The source of an edge of this type must be the exit vertex of a task, and the destination must be the entry vertex of its sibling task, e.g., edge (v_3, v_4) in Figure. 2.

B. Branching Structures

OpenMP programs commonly contain two types of branching structures. One corresponds to the statement of base languages, such as `if-else`. The other one is induced from

TABLE I
SUMMARY OF THE BENCHMARK PROGRAMS CURRENTLY INCLUDED IN **OMPTGB**.

Applications	Source	LOC	`tied`	`taskwait`	`depend`	`if`	`final`	`if-else` (C/C++)
botsspar$^\diamond$		209	√	√	×	×	×	√
botsalgn*	spec2012 [27]	1277	√	×	×	×	×	√
kdtree$^\diamond$		1132	√	√	×	√	×	√
poisson2D*$^\diamond$		474	√	×	√	×	×	√
sparseLU*$^\diamond$	kastors-1.1 [28]	405	√	×	√	×	×	√
strassen*		736	√	×	√	×	×	√
fft$^\diamond$		4447	√	√	×	×	×	√
fib		118	√	√	×	√	√	√
floorplan$^\diamond$	bots-1.1.2 [29]	365	√	√	×	×	√	√
nqueens$^\diamond$		274	√	√	×	√	√	√
sort		237	√	√	×	×	×	×
sparseLU$^\diamond$		218	√	√	×	×	×	√
strassen		735	√	√	×	√	×	√
health$^\diamond$		455	√	√	×	√	×	√
dense_algebra*		1479	√	×	×	×	×	×
finite_state_machine*	dash-1.0 [30]	641	√	×	×	×	×	×
nbody_methods$^\diamond$		5478	√	√	×	×	×	√
sparse_algebra*		971	√	×	×	×	×	×
pt_to_pt_pingpong$^\diamond$	openmpmpi [31]	309	√	√	×	×	×	√
pt_to_pt_overlap$^\diamond$		313	√	√	×	×	×	√
cpp_sortOpenMP	ompSCR_v2.0 [32]	785	√	√	×	×	×	×
taskbench$^\diamond$	ompbench_C_v31 [33]	167	√	√	×	√	×	√

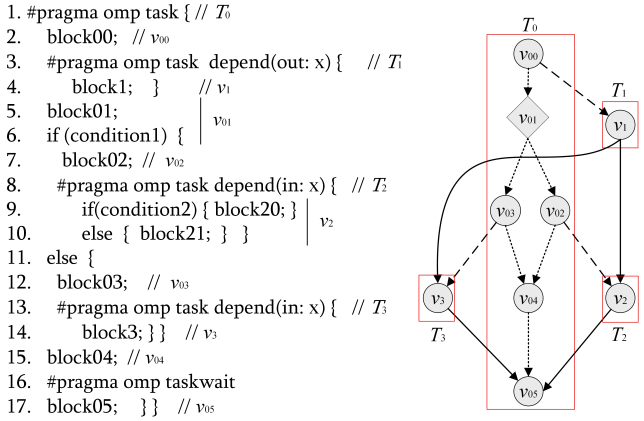

Fig. 3. An example with `if-else` branches.

OpenMP clauses, e.g., `if` and `final` contained in the `task` construct.

`if-else` Structures

If the `if-else` structures are nested with the creation of and synchronization among tasks, it is necessary to explicitly model these branching structures to precisely represent the workload. The `if-else` structures contained in a single vertex are not needed to be represented in the task graph. Figure. 3 gives an example of OpenMP DAG with conditional branches.

The `if-else` structure from Lines 6 to 14 in the program of Figure. 3 contains task creations and thus is explicitly modeled so that the corresponding DAG has a branching structure starting with v_{01} and ending at v_{04}. In contrast, the `if-else` structure from Lines 9 to 10 (in v_2) is not explicitly modeled, since this code segment does not contain any task creation or synchronization point.

For convenience, in order to distinguish the conditional structure, we use a diamond to represent the entry vertex of a conditional structure, e.g., vertex v_{01} as shown in Figure. 3. We call such diamond vertices as *conditional* vertices, which are defined more formally as follows.

Definition 1. *A vertex v is a conditional vertex if v (1) has more than one outgoing edges in E_1 and (2) does not have any outgoing edge in E_2 or E_3.*

In our model, for any conditional vertex v, at most one successor of v can be executed. In contrast, For any non-conditional vertex v, all the successors of v must be executed if v has been executed.

Previous work on real-time scheduling of conditional parallel task graphs all assume the branching structures and the fork-join structures are well-nested (i.e., there is no edge between a vertex in a conditional branch and another vertex outside that branch) [13], [16]. Unfortunately, this is not necessarily the case for realistic OpenMP programs. For example, in Figure. 3, task T_2 is created within the branching structure between v_{00} and v_{04}, but joints with a vertex v_{05} after this branching structure. More than half (59%) of applications in Table. I, which are marked with diamond, contain non-well-nested branching structures. Motivated by this observation, our proposed new response time analysis technique in Section. V will be able to handle cases where branching and fork-join structures are not well-nested.

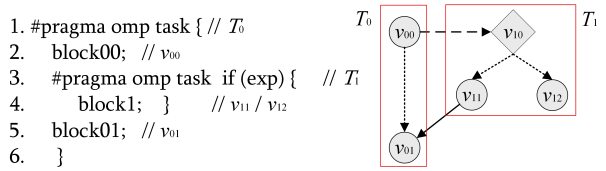

Fig. 4. An example OpenMP program with `if`.

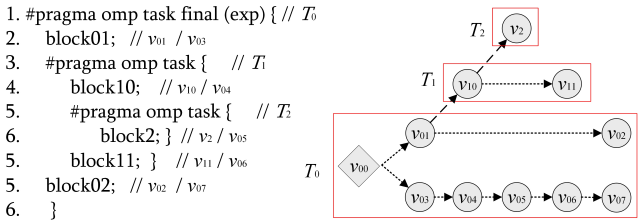

Fig. 5. An example OpenMP program with `final`.

Besides `if-else`, the loop keywords in the base language C, such as `for-loop` and `while-loop`, also indicate conditional structures. Similarly, we only need to model the loop structures that are nested with the creation of and synchronization among tasks. A loop body that only contains sequentially executed codes is formulated into a vertex, and does not need be explicitly modeled in the task graph.

We assume that the number of iterations of each loop can be bounded by a constants, and the loops are unrolled (so the task graphs are acyclic). The bounded loop is a common assumption in real-time system design [34], since the timing analysis for program with unbounded loops is very difficult and even impossible. Many works devoted to bounding loop iterations [35]–[38].

`if/final` Structures

Besides the `if-else` statements, OpenMP `task` constructs following the `if` and `final` clauses may also create conditional branches in a DAG graph.

Figure. 4 gives an example of an OpenMP program which contains an `if` clause. Different evaluations of the associated expression corresponds to two conditional branches. If the `if` clause expression evaluates to `false`, the associated task (T_1 in Figure. 4) is not deferred with respect to its parent task (T_0 in Figure. 4). More specifically, the parent task T_0 should be suspended until T_1 is completed. A synchronization edge from v_{11} to v_{01} can express this dependency between the child and its parent due to the `false` evaluation of the `if` clause. In contrast, a `true` evaluation of `if` clause does not result in any additional restriction to the execution of corresponding tasks, and there is no synchronization edge added to the `true`-evaluation related branch containing vertex v_{12} in Figure. 4.

Figure. 5 shows a `task` construct following a `final` clause. If the associated expression of T_0 evaluates to `true`, T_0 and its (grand)children T_1 and T_2 must be executed by the same thread. Moreover, for any (grand)child of T_0, denoted by T', the parent task of T' should be suspended until T' is completed. This indicates that task T_0 and all of its (grand)children should be executed sequentially as if they are integrally united as a task (called *virtual* task), which is represented by branch $(v_{03}, \cdots, v_{07})$ in Figure. 5. Otherwise, T_0, T_1 and T_2 cannot be seen as forming a virtual task (See branch (v_{01}, v_{02}) of T_0 in Figure. 5).

C. Concepts Related to Scheduling

In OpenMP, the execution entity for executing tasks is called *thread*. Scheduling OpenMP tasks is to assign tasks (or the vertices of tasks) onto threads ensuring that the scheduling constraints are satisfied. In the following we briefly introduce the OpenMP tasks scheduling constrains.

Task Scheduling Points

In OpenMP, a Task Scheduling Point (TSP) is a program point at which the execution can be interrupted and scheduling may be triggered. TSPs often occur upon task creation and completion, and at task synchronization points such as `taskwait` directives. Following the convention in previous work [3], [4], we call the code segment between two TSPs a *part*.

Note that in [3], [4] which didn't consider branching structures, each task has a linear structure and each part corresponds to a vertex in the task graph. Therefore, the TSPs exist after each vertex, implying that the execution of each vertex should not be interrupted, and a context switch may occur when a vertex completes its execution.

However, in our model which contains branching structures, a part may correspond to more than one vertices. For example, in Figure. 3, the TSPs exist in front of the vertices v_{00}, v_3 and v_2. There is no TSP after the vertex v_{01}. Therefore, either v_{01} and v_{02} or either v_{01} and v_{03} form a part.

In our model, a TSP may exist in two cases:

- A TSP may exist after a vertex that has at least one outgoing edge of type E_2. In this case, a new part is released at this TSP.
- A TSP may exist before a vertex that has at least one incoming edge of type E_3. In this case, the current part completes at this TSP.

Eligibility

In the following, we will introduce conditions for a part to be *eligible* for execution. We first introduce some auxiliary concepts.

A vertex u is a *trigger vertex* of v if $(u, v) \in E_1 \cup E_2$. For example, in Figure. 3, v_{00} is the trigger vertex of v_{01}. Vertices v_{04} and v_2 share the same trigger vertex v_{02}. The conditional vertex v_{01} is the trigger vertex of its successors v_{03} and v_{02}, both of which are the trigger vertices of v_{04}.

If a trigger vertex u is conditional (See in Definition. 1), u can only trigger one of its immediate successors. For example, in Figure. 3, only one of v_{02} and v_{03} is triggered by v_{01}. Otherwise, if u is a non-conditional vertex, the completion of u must trigger all of its immediate successors. For example,

in Figure. 3, the vertices v_{04} and v_2 should be triggered at the same time when v_{02} is finished.

A task T_j is a *depended* task of vertex v if u is the exit vertex of T_j and $(u, v) \in E_3$. Suppose that the depended task T_j of v has been created before v is triggered, v should not be executed until T_j is finished. For example, in Figure. 3, T_1 is the depended task of vertices v_2 and v_3. We know that T_1 must be created before either v_2 or v_3 is triggered. Thus, the vertex v_2 as well as v_3 should wait for the completion of task T_1. Moreover, vertex v_{05} has two depended Tasks T_2 and T_3, and only one of them can be created. If T_2 is created, v_{05} will be executed after T_2 is finished.

Now we are ready to introduce the conditions for a part to be eligible:

Let v be the first vertex of a part P. P is eligible for execution iff v is triggered but has not been executed, and none of v's depended task is being executed or suspended.

For example, in Figure. 3, the single vertex v_{05} forms a part of task T_0, which becomes eligible when its trigger vertex v_{04} and one of its depended task T_2 or T_3 are both finished. Another example is the part that consists of a single vertex v_3, which becomes eligible when the trigger vertex v_{03} and the depended task T_1 of v_3 are both finished.

`Tied` and `Untied` Tasks

A task in OpenMP is either `tied` or `untied`. If a `tied` task starts execution on a thread then it will only execute on this thread in its whole life cycle. In particular, if a `tied` task is suspended, later it has to be resumed on the same thread. In contrast, a `untied` task is not tied to any thread and so in case it is suspended it can later be resumed by any thread.

Moreover, a `tied` task can only be suspended by its (grand) children or by `untied` tasks. Therefore, at any time, if a `tied` task T can be assigned to a thread S, we should guarantee that all the unfinished tasks which have been tied to S have a child or a grand child T. In contrast, a `untied` task or any part of a `untied` task can be executed on any available thread.

In OpenMP, tasks are by default `tied`, unless explicitly specified as `untied`. A `untied` task T becomes `tied` if a (grand)parent of T or T itself follows `final` and the `final` clause expression evaluates `true` [1]. In this case, T and its (grand)children are forced to be executed by the same thread, which can be seen as forming a virtual `tied` task, denoted by T'. The execution order of the parts in T' is defined as follows. For any two parts P and P' of T', P is executed before P' if P is not the successor of P' and there is a path from a predecessor of P' to P. For example, if exp in Line 1 of Figure. 5 evaluates `true`, the `untied` clauses are ignored, and three `untied` tasks T_0, T_1 and T_2 are combined into a virtual `tied` task containing vertices $v_{03}, \cdots, v_{07}$ in Figure. 5. Any part of T_1 (v_{04} and v_{06}) is executed before the last part of T_0 (v_{07}), and any part of T_2 (v_{05}) is executed before the last part of T_1 (v_{06}).

`Tied` tasks enjoy the following benefits: (1) `tied` task precludes immigrations among threads, which simplifies the implementation of the scheduling algorithm and reduces context switch cost; (2) `tied` task automatically prevent deadlocks in the presence of critical sections [1], [4], [39]. However, the extra scheduling constraints of `tied` tasks make it more difficult to analyze their timing behaviors. It is an open question of how to bound the response time of programs containing `tied` tasks. Previous work [4], gives a negative answer to this question: their response times are in general unbounded. In the next section we will report some positive results regarding this problem.

V. A Case Study

In this section, we present new theoretical results for analyzing the worst-case timing behavior of a subclass of OpenMP programs. On one hand, our results are obtained by a better understanding of the workload structure of OpenMP programs with the task graph models in **ompTGB** as presented in last section. On the other hand, we use benchmark programs in **ompTGB** to evaluate our theoretical results.

In [4], a response time bound is derived for OpenMP programs exclusively executing on a parallel platform of m processing units, in which all tasks must be `untied`. For OpenMP programs containing `tied` tasks, [4] claimed that *a timing analysis, besides being conceptually very difficult to achieve, ... would lead to unacceptably pessimistic response-time bounds*. However, in the following we will show that for many OpenMP programs containing `tied` tasks, we can still well bound their response times.

A. Breadth First Scheduling (BFS)

The runtime environments in most OpenMP implementations supports two scheduling policies BFS (Breadth First Scheduling) [40] and WFS (Work First Scheduling) [41]. Roughly speaking, BFS creates all child tasks before executing them, while WFS executes new tasks immediately after they are created.

WFS is not suitable for scheduling `tied` tasks [4]. By WFS, a thread always suspends the current task and executes the child task immediately after it is created. Therefore, the parent task and its children are both tied to the same thread, and thus the parallel program will be executed sequentially, which makes the parallel programming meaningless.

Therefore, in the following we focus on how to bound the response time under BFS for `tied` tasks. Similar to previous work [4], [42], we assume each thread is statically allocated to and exclusively occupies one processing core.

Algorithm. 1 shows the pseudo-code of BFS. For any time when a thread S finishes the execution of a part $P_{i,z}$ (which is in task T_i), the routine in Algorithm. 1 is called to select from existing eligible parts to execute on idle threads.

We assume that the next part of T_i to be executed is $P_{i,z+1}$. If $P_{i,z+1}$ is eligible, S prefers to continue the current task T_i as shown in Lines 1 to 3.

After that, there may still exist some new eligible parts and idle threads due to the completion of $P_{i,z}$. For any eligible part $P_{j,w}$ of task T_j and idle thread S', if T_j is `untied`, then

Algorithm 1 Pseudo-code of BFS

1: **if** $P_{i,z+1}$ is eligible **then**
2: execute $P_{i,z+1}$ on S;
3: **end if**
4: **for** any eligible part $P_{j,w}$ and idle thread S' **do**
5: execute $P_{j,w}$ on S' iff
6: (a) T_j is `untied`; or
7: (b) T_j has been tied to S'; or
8: (c) $P_{j,w}$ is the first part of T_j and any unfinished task T_i that has been tied to S' has a (grand) child T_j.
9: **end for**

$P_{j,w}$ can be assigned to S' (Line 6: condition (a)). Otherwise, S' can execute $P_{j,w}$ if T_j is tied to S' (Line 7: condition (b)) or $P_{j,w}$ is the first part of T_j and any unfinished task T_i that has been tied to S' has a (grand) child T_j. (Line 8: condition (c)).

It should be emphasized that a `tied` task may be a real OpenMP task or a virtual task corresponding to several tasks that are forced to be executed by the same thread due to the `final` clauses. In the following context, without any confusion, the `tied` task by default is a real OpenMP task, and we will give additional specification when we mention virtual `tied` tasks.

B. Response Time Analysis

In this section we will show that the response time of OpenMP programs containing `tied` tasks can also be bounded, as long as any `tied` task T does not contain `taskwait` and any child of T does not follow the `if` clause.

Definition 2 (Work-Conserving Scheduling)**.** *A scheduling algorithm is work-conserving iff it never idles threads when there exist eligible parts waiting for execution.*

The following response time bound of a task graph is known for work-conserving scheduling algorithms [4].

Theorem 1. *[4] The response time $R^{ub}(G)$ of a task graph G under any work-conserving scheduling algorithm on m threads is bounded by the following:*

$$R^{ub}(G) = len(G) + \frac{vol(G) - len(G)}{m} \tag{1}$$

where $len(G)$ is the length of the longest path in G and $vol(G)$ is the maximal makespan of G executing on a uniprocessor.

Unfortunately, as discussed in [4], when a task graph contains `tied` tasks, BFS is not work-conserving, and thus the response time bound in (1) is not valid. An counter-example given in [4] indicates that the pessimistic scheduling may occur when a `tied` task is required to synchronize with its children.

A `tied` task T may synchronize with its children only if: (1) T contains `taskwait` clauses; or (2) a child of T follows a `if` clause; or (3) T contains a `final` clause. In cases (1) and (2), the child of T that needs to be synchronized with T only comes from the first level children of T. In case (3), T are required to synchronize with all of its (grand)children. By preventing cases (1) and (2), in the following we will show that BFS is still work-conserving even if `tied` tasks are present and synchronize with their children.

Lemma 1. *BFS is work-conserving if any `tied` task T does not contain `taskwait` and no child of T contains `if`.*

Proof. We will prove that if there exists an eligible part $P_{i,z}$ of T_i waiting for execution, then there cannot be an idle thread at the same time. This is clearly true if T_i is an `untied` task, since an eligible part of it can be executed on any idle thread. In the following, we focus on the case that T_i is a `tied` task [2].

We will prove this by contradiction, assuming $P_{i,z}$, an eligible part of a `tied` T_i that does not contain `taskwait` and no child of T_i contains `if`, is waiting for execution while there is an idle thread S.

First, we prove that $P_{i,z}$ must be the first part of T_i. This is because, otherwise, $P_{i,z}$ will immediately start execution on its tied thread as soon as it becomes eligible according to Lines 1 to 3 in Algorithm. 1.

We consider an arbitrary task T_j that has been tied to S. For two consecutive parts $P_{j,z}$ and $P_{j,z+1}$, $P_{j,z+1}$ will become eligible immediately after $P_{j,z}$ unless one of the following two cases is met:

- There is a `taskwait` in between[3] .
- There is a child T_l of T_j in between, and T_l follows a `if` and the associated expression evaluates `false` [4] .

Therefore, according to Algorithm. 1 (Lines 1 to 3), $P_{j,z+1}$ will continue to execute on the same thread S. So we can conclude that for an idle thread S, any task tied to it must have been finished.

In summary, we have proved that (1) $P_{i,z}$ is the first part of T_i and (2) any task tied to S has been finished. Therefore, the condition (c) in Line 8 holds, and thus $P_{i,z}$ will be executed on S, which contradicts out assumption. □

By combining Theorem. 1 and Lemma. 1, we can get the following conclusion.

Theorem 2. *The response time of a task graph G in which any `tied` task T does not contain `taskwait` and no child of T contains `if` clauses scheduled by BFS on m threads is bounded by the upper bound in (1).*

According to Theorem. 2, in any OpenMP program having a response time bound in (1), a `untied` task can freely synchronized with other tasks, and moreover, a `tied` task T can freely synchronized with the other task T' if T' is not a (grand)child of T. Otherwise, the synchronization between

[2]Note that a tied task may be a virtual task that is a combination of several tasks due to `final` clauses.

[3]The other type of synchronization edges due to the `depend` directives can only points to the first part (i.e., the entry vertex) of a task.

[4]The `final` clauses belonging to T_j and the (grand)parent of T_j force all the children of T_j to be combined into T_j if the associated expression evaluates `true`. In this case, there is no child in between any consecutive parts $P_{j,z}$ and $P_{j,z+1}$ of T_j.

T and its (grand)child T' should be caused by `final` clauses. For any synchronization between a tied task and its children due to `taskwait` and `if` clauses, it can be equivalently transformed into a synchronization between a tied task and its siblings by using `depend` clauses [28] (See the related discussion in Section. V-D for details).

In Table. I, there are 7 applications fulfilling the condition of Theorem. 2. All of them are marked with stars as shown in the first column of Table. I.

C. Calculating $len(G)$ and $vol(G)$

We can calculate $len(G)$ recursively as follows. Let $L(v)$ denote the length of the subgraph of G consisting of vertices reachable from v, which is calculated by:

$$L(v) = c(v) + \max_{(v,u)\in E} L(u).$$

Finally, $len(G) = L(v_0)$, where v_0 is the vertex corresponding to the entry point of the whole program (each program has only one entry point).

In the following we calculate $vol(G)$. We use $V(v)$ to denote the *volume* of the subgraph of G consisting of vertices reachable from v, which is recursively computed as follows:

$$V(v) = \begin{cases} c(v) + \max\limits_{(v,u)\in E_1 \bigcup E_2} V(u), \ v \text{ is a conditional vertex;} \\ c(v) + \sum\limits_{(v,u)\in E_1 \bigcup E_2} V(u), \text{ otherwise.} \end{cases}$$

Finally, $vol(G) = V(v_0)$, where v_0 is the vertex corresponding to the entry point of the whole program. The above presented computation of both $len(G)$ and $vol(G)$ can be implemented by dynamic programming in polynomial time.

In [13], a method was proposed to compute $len(G)$ and $vol(G)$ for conditional DAG models with well-nested branching and fork-join structures (i.e., there is no edge between a vertex in a branch of a conditional statement and another vertex outside that branch). However, their method may overcount the $vol(G)$ for general cases with non-well-nested structures, for example, the applications marked with diamond in Table. I.

The computation of $vol(G)$ in [13] differs from ours in that they use E, rather than $E_1 \cup E_2$, in the subscript of the max and $\sum$ operations, which may overcount the workload. For example, according to their calculation, T_2 and T_3 in Figure. 3, which are forked from different branches of the same conditional statement at Line 6, both contribute to $vol(G)$. However, in reality, only one of them can be executed.

In our computation, we fix this problem by excluding E_3 in the computation procedure (or equivalently, deleting all the edges in E_3 from the task graph). This is because, after deleting all the edges in E_3, tasks created in different branches can only submit their costs to their common conditional predecessors. In this case, only one of them contributes to $vol(G)$. On the other hand, the synchronization edges can only delay the execution of a vertex, but does not decide whether this vertex will be actually executed or not, so our computation method will not miss any vertex in an actual execution sequence of the program, and thus can compute $vol(G)$ correctly.

D. Evaluation

We use benchmark programs in **ompTGB** to evaluate the response time bounds in Theorem. 2 for OpenMP programs with no `if` clauses and with `tied` tasks containing no `taskwait`. In particular, we use 7 benchmark programs that are marked with stars in Table. 1. The inter-task synchronization in these programs are realized by `depend`.

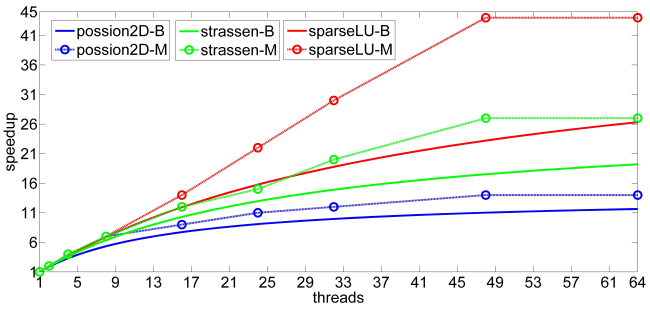

Fig. 6. Evaluation of the applications in KASTORS benchmark.

For each benchmark program, we compare the response time bound obtained using Theorem. 2 and the measured response time in simulations when executing on different number of threads (each thread exclusively occupies a physical core). The response time bound and the measured response time are reported in the form of the *speedup* comparing with the response time on a single thread, which equals $vol(G)$. The speedup for the response time bound is smaller than the measured response time since the former is an upper bound of and in general larger than the later. For example, suppose $vol(G) = 12$, the response time bound is 8 and the measured response time is 6, then the speedup for the response time bound is $12/8 = 1.5$ and the speedup for the measurement is $12/6 = 2$.

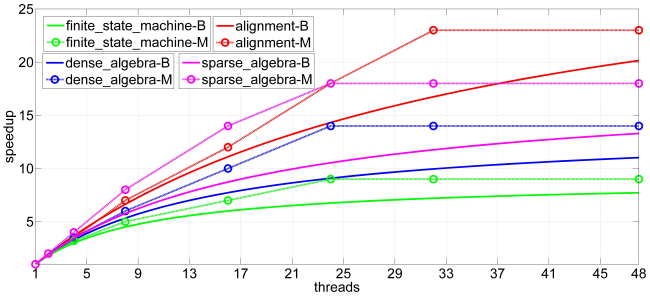

Fig. 7. Evaluation of the applications in SPEC and DASH benchmarks.

Figures. 6 and 7 show the experiment results, in which each curve "xxx-B" is the speedup of the response time bound of program "xxx", and each curve "xxx-M" is the speedup of the measured response time of program "xxx". The gap between the speedup of the bounds and the measured values first increases as the number of threads increases, until the

maximal parallelism of the program is achieved. After that point, adding more threads will not improve its measured response time, while then the response time bound continues to improve as the number of threads increases, and eventually approach to the measured values.

Discussions: `depend` vs. `taskwait`

Using Theorem. 2 we can bound the response time for `tied` tasks that use `depend` instead of `taskwait` for inter-task synchronization. However, from Table. I we can see that most of the benchmark programs we have collected so far use `taskwait`, while only the ones in "kastors-1.1" use `depend`. This is mainly because `depend` is a feature newly introduced in OpenMP 4.0 and most of the benchmarks are presented before that.

Recently, some work has been done to compare the average-case performance of different implementations (with `depend` or `taskwait`) of OpenMP applications in both high performance and embedded domains [2], [28], [43]. These studies show that using `depend` can increase the parallelism and accelerate the execution of the programs compared to their `taskwait` implementations. Moreover, [28] developed techniques to transform programs with `taskwait` into their equivalent `depend` implementations to achieve a higher performance.

The result in Theorem. 2 echoes the above preference to `depend` in the perspective of *worst-case* timing behaviors. Therefore, we advocate to use `depend` for inter-task synchronizations in the development of OpenMP based real-time softwares, for both average-case and worst-case performance reasons.

VI. CONCLUSIONS

Multi-cores are more and more widely used in real-time systems. To fully utilize the power of multi-core processors, we must parallelize the software. OpenMP is a popular parallel programming framework in general and high-performance computing and is also promising for real-time computing. To support the research of real-time scheduling of OpenMP based real-time workload, we present a benchmark suite, **ompTGB**, which collects realistic OpenMP programs and transforms them into task graph models, so that the real-time scheduling researchers can easily understand and use them. We also present a new response time bound for a subset of OpenMP programs and use it to demonstrate the usage of **ompTGB**. Currently, **ompTGB** only models explicit tasks. In the next step, we will include the modeling of implicit tasks (due to the directives such as `worksharing`). We will also continuously collect more realistic OpenMP programs into **ompTGB**.

REFERENCES

[1] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[2] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quinones, "A lightweight openmp4 run-time for embedded systems," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 43–49.

[3] R. Vargas, E. Quinones, and A. Marongiu, "Openmp and timing predictability: A possible union?" in *Design, Automation & Test in Europe Conference & Exhibition*, 2015, pp. 617–620.

[4] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, "Timing characterization of openmp4 tasking model," in *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, 2015, pp. 157–166.

[5] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 261–272.

[6] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[7] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global edf scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 51, no. 4, pp. 395–439, 2015.

[8] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*. IEEE, 2010, pp. 259–268.

[9] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of dags," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3242–3252, 2014.

[10] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 97–105.

[11] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," in *Proceedings of the 21st International conference on Real-Time Networks and Systems*. ACM, 2013, pp. 287–296.

[12] M. Qamhieh, L. George, and S. 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*. ACM, 2014, p. 13.

[13] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 211–221.

[14] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones, "Response-time analysis of dag tasks under fixed priority scheduling with limited preemptions," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1066–1071.

[15] M. Stigge and W. Yi, "Graph-based models for real-time workload: a survey," *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, 2015.

[16] J. Sun, N. Guan, Y. Wang, Q. Deng, P. Zeng, and W. Yi, "Feasibility of fork-join real-time task graph models: Hardness and algorithms," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, p. 14, 2016.

[17] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[18] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Sørensen, P. Wägemann, and S. Wegener, "Taclebench: A benchmark collection to support worst-case execution time research," in *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET16)*, 2016.

[19] H. Li, P. De Meulenaere, and P. Hellinckx, "Powerwindow: a multi-component taclebench benchmark for timing analysis," in *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer, 2016, pp. 779–788.

[20] P. Wägemann, T. Distler, and et.al, "Gene: A benchmark generator for wcet analysis," in *OASIcs-OpenAccess Series in Informatics*, vol. 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[21] J. Harbin, T. Fleming, L. S. Indrusiak, and A. Burns, "Gmcb: An industrial benchmark for use in real-time mixed-criticality networks-on-chip," *Proc. WATERS, 27th ECRTS*, 2015.

[22] V. V. Dimakopoulos and A. Georgopoulos, "The ompi openmp/c compiler," in *Proc of the 10th Panhellenic Conference on Informatics*, 2005, pp. 156–162.

[23] "ompTGB homepage. "http://www4.comp.polyu.edu.hk/~csguannan/openmp"."

[24] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz¡xopen source graph drawing tools," in *International Symposium on Graph Drawing*. Springer, 2001, pp. 483–484.

[25] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[26] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *COCOMO II Forum*, vol. 2007, 2007.

[27] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux *et al.*, "Spec omp2012¡xan application benchmark suite for parallel systems using openmp," in *International Workshop on OpenMP*. Springer, 2012, pp. 223–236.

[28] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, "Evaluation of openmp dependent tasks with the kastors benchmark suite," in *International Workshop on OpenMP*. Springer, 2014, pp. 16–29.

[29] A. Duran González, X. Teruel, R. Ferrer, X. Martorell Bofill, and E. Ayguadé Parra, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *38th International Conference on Parallel Processing*, 2009, pp. 124–131.

[30] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, and A. Cristal, "Dash: a benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models," in *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 2014, p. 4.

[31] J. M. Bull, J. P. Enright, and N. Ameer, "A microbenchmark suite for mixed-mode openmp/mpi," in *International Workshop on OpenMP*. Springer, 2009, pp. 118–131.

[32] A. J. Dorta, C. Rodriguez, and F. de Sande, "The openmp source code repository," in *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2005, pp. 244–250.

[33] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in *International Workshop on OpenMP*. Springer, 2012, pp. 271–274.

[34] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem¡xoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[35] G. Bernat, A. Colin, and S. Petters, *pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems*. University of York, Department of Computer Science, 2003.

[36] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 57–66.

[37] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley, "Bounding loop iterations for timing analysis," in *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*. IEEE, 1998, pp. 12–21.

[38] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. v. Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol. 18, no. 2, pp. 129–156, 2000.

[39] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[40] G. J. Narlikar, "Scheduling threads for low space requirement and good locality," *Theory of Computing Systems*, vol. 35, no. 2, pp. 151–187, 2002.

[41] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *ACM Sigplan Notices*, vol. 33, no. 5. ACM, 1998, pp. 212–223.

[42] E. Ruffaldi, F. Brizzi, G. Dabisias, and G. Buttazzo, "Soma: an openmp toolchain for multicore partitioning," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1231–1237.

[43] P. Virouleau, A. Roussel, F. Broquedis, T. Gautier, F. Rastello, and J.-M. Gratien, "Description, implementation and evaluation of an affinity clause for task directives," in *IWOMP 2016*, 2016.