Timing Analysis of First-Come First-Served Scheduled Interval-Timed Directed Acyclic Graphs

R.M.W.Frijns*, S.Adyanthaya*, S.Stuijk*, J.P.M.Voeten*†, M.C.W.Geilen*, R.R.H.Schiffelers*‡, and H.Corporaal*
*Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
Email: R.M.W.Frijns@tue.nl
†TNO-ESI, Eindhoven, The Netherlands
‡ASML, Veldhoven, The Netherlands

Abstract—**Analyzing worst-case application timing for systems with shared resources is difficult, especially when non-monotonic arbitration policies like First-Come-First-Served (FCFS) scheduling are used in combination with varying task execution times. Analysis methods that conservatively analyze these systems are often based on state-space exploration, which is not scalable due to its inherent susceptibility to combinatorial explosion.**

We propose a scalable timing analysis method on periodically restarted Directed Acyclic Task Graphs, that can provide conservative bounds on task timing properties when shared resources with FCFS scheduling are used. By expressing task enabling and completion times in intervals, denoting best-case and worst-case timing properties, contention on the shared resources can be estimated using conservative approximations.

With an industrial case study we show that our approach can easily analyze models with thousands of tasks in less than 10 seconds, and the worst-case bounds obtained show an average improvement of 46% compared to bounds obtained by static worst-case analysis.

I. Introduction

With the scale and complexity of modern-day embedded systems skyrocketing, the process of designing these systems is relying heavily on modeling and automated construction. Designers need to understand the behavior of such systems in early design stages, in order to reason about the impact of design decisions on the overall system performance.

In the domain of high-performance mechatronics, applications calculate actuator output based on periodically arriving sensor input, in order to control e.g. the motion of parts of the system. These applications are mapped to (typically general-purpose) distributed computation platforms, and are executed under strict application timing requirements [1]. Since pipelining has little effect on control performance, these applications are acyclic and are therefore expressed as periodically restarted Directed Acyclic Task Graphs (DAGs) [2]. Because the underlying execution platforms are often unpredictable, task execution times can vary, so typically tasks in these DAGs are characterized by minimum and maximum execution times.

Many (interconnect) networks in modern embedded systems, like e.g. FlexRay [3] and RapidIO [4], employ First-Come-First-Served (FCFS) scheduling. The timing of DAGs that are (partially) mapped to this type of resource is hard to analyze, since in addition to the varying execution times, tasks suffer from additional delay due to contention on these resources. This delay depends on the timing of the DAG itself, which varies with each execution of the graph. With timing analysis, the worst-case timing properties are to be found, like completion times and response times of tasks, which must be conservative for any execution of the DAG.

The Synchronous Dataflow (SDF) [5] model of computation is commonly used in performance analysis and synthesis of e.g. Multi-Processor Systems-On-Chip. With SDF analysis, the impact of mapping and scheduling on application timing and platform memory requirements can be analyzed. SDF is a more expressive model of computation than the restricted homogeneous SDF commonly used with DAGs, however, there are no SDF-based timing analysis techniques that can analyze DAGs with aforementioned properties and requirements.

Timing analysis based on model checking [6] is also a widely used approach in performance analysis. Here, timing properties are verified by analyzing a set of Timed Automata [7]. These techniques are able to deal with non-monotonic models, however, as their underlying state-space easily explodes, model checking does not scale very well. Especially when tasks are allowed to have variation in their execution time, state-space explosion is easily encountered.

In this paper, we propose a scalable analysis technique that can analyze the timing of task graphs consisting of tasks with varying execution times mapped to shared resources under a FCFS scheduling policy. In our approach, timing is expressed in intervals, which denote the best-case and the worst-case timing properties of tasks. By taking into account the best-case timing as well, the contention on the shared resources can be estimated. Time intervals are propagated through the nodes in the graph taking into account the contention. Scalability is ensured by using a conservative approximation on the interval bounds of the tasks, and is demonstrated in experiments on a realistic industrial-sized model.

II. Related Work

SDF analysis is a well-known approach in the design and analysis of distributed real-time systems. SDF is commonly used in static timing analysis, assuming tasks with fixed execution times. However, shared resources with arbiters that are not in the class of budget schedulers can not be modeled

978-3-9815370-2-4/DATE14/ © 2014 EDAA

with these techniques. In [8], a resource-aware extension of SDF analysis is used in a design-space exploration technique to dimension multiprocessor systems. Shared resource accesses are explicitly modeled, and their impact is analyzed by considering the state-space of the different executions of an SDF graph. Even with fixed execution times, heuristics are needed to prevent a state-space explosion.

Analysis methods based on model checking tools, like the popular UPPAAL [9] tool, use Timed-Automata (TA) [7] to verify timing properties. The execution state-space of these TA is exhaustively searched to e.g. find worst-case timing properties. This results in very accurate bounds, but model checking inherently suffers from state-space explosion. Different approaches focus on heuristics and model-reductions to limit the state-space. In [10], UPPAAL is used to analyze the timing of a RapidIO network. By applying heuristics, somewhat larger models can be handled at the cost of accuracy, but analysis of the full system model under high traffic load is not possible without reverting to simulation techniques. The work of [11] combines model checking with the Real-Time Calculus (RTC) [12]. With RTC, event streams are expressed in arrival curves, which bound the minimum and maximum number of events seen in a stream for any time window of some size. Scalability of the TA-analysis is improved by abstracting part of the system under consideration into a single arrival curve. Since the RTC arrival curves are defined over *any* time window, information about absolute time is lost. Therefore, it cannot distinguish absolute time offsets in streams, resulting in pessimistic bounds when merging e.g. two strictly periodic streams which have a phase shift relative to each other [13]. Also, RTC assumes arrival curves are given. In our approach it is sufficient to know the minimum and maximum task execution times.

III. Timing Properties

We assume that a Directed Acyclic Graph (DAG) $G = (T, D)$ is given, with T a finite set of tasks and $D \subseteq T \times T$ a set of dependencies. We further let R denote a collection of resources and assume that $M : T \to R$ maps any task to the (private or shared) resource it is statically bound to. The execution timing of tasks in G is expressed in intervals.

Definition 1. (Interval) *I is the set of closed intervals defined by $I = \{[a, b] \mid a, b \in \mathbb{N}, b \geq a\}$. Intervals are ordered according to a subset ordering, i.e. $(I, \subseteq)$ is a partially ordered set. The lower bound L of an interval is given by $L([a, b]) = a$, the upper bound U is given by $U([a, b]) = b$.*

The timing properties can be expressed in the following task labelings on tasks in G:

- $E : T \to I$ maps tasks to their execution interval. $E(t) = [a, b]$ denotes that an execution of task t will require at least a, and at most b time units.
- $C : T \to I$ maps tasks to their completion interval bounds, i.e. $C(t)$ is the interval at which task t will finish its execution.
- $En : T \to I$ maps tasks to their enabled interval bounds.
- $B : T \to I$ maps tasks to a busy interval. The busy interval of a task reflects the delay between its enabling and its completion time, including both waiting time as a result of contention, and its execution time.

Note that the busy interval B is not a response interval, but an algebraic term that, given a best-case (worst-case) enabling of a task, denotes the delay that results in a best-case (worst-case) completion of that task. The worst-case response time of t, i.e. its maximum delay, is not necessarily in $B(t)$; it is bounded by $U(C(t)) - L(En(t))$.

The execution interval labeling E is assumed to be given for DAG G. Furthermore, all resources in R employ a *First-Come-First-Served (FCFS)* scheduling policy, where tasks are queued for execution based on the time they are enabled. Fixed-order schedules on (a subset of) tasks in G can then be modeled by adding sufficient scheduling dependencies to the DAG, while omitting (some of) these dependencies allows modeling of e.g. shared communication resources.

A task is enabled when all its predecessors have completed their execution. Without loss of generality, tasks without inputs are assumed to be enabled at $[0, 0]$. The relation between the enabled interval of a task $t \in T$ of G and its completion interval is:

$$En(t) = \begin{cases} [0,0] & \text{if } pred(t) = \varnothing \\ \max\limits_{t' \in pred(t)} C(t') & \text{otherwise} \end{cases}, \qquad (1)$$

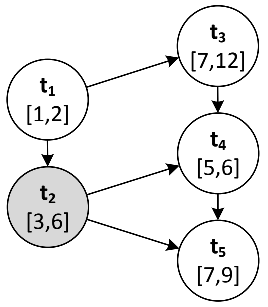
where $pred(t) = \{t' \in T \mid (t', t) \in D\}$ denotes the set of predecessors of t, and the max-operator on intervals is defined as $\max V = [max_{i \in V} L(i), max_{i \in V} U(i)]$ for all finite non-empty sets $V \subseteq I$.

When t is enabled, it enters the execution queue of its resource, where it needs to wait on the completion of all tasks that were queued earlier. When t eventually gets access to its resource, it starts executing and completes after some time in $E(t)$. During execution of t, no other task can claim the resource on which t is executing. The busy interval $B(t)$ of task t is the period between its enabling and its completion, and thus includes both the time that t spends waiting in the execution queue (which depends on the contention) as well as the time that t is executing. The completion interval of any task $t \in T$ in terms of its busy interval is then given by:

$$C(t) = En(t) + B(t), \qquad (2)$$

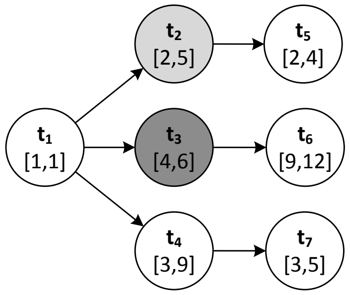
where $i_1 + i_2 = [L(i_1) + L(i_2), U(i_1) + U(i_2)]$ for all $i_1, i_2 \in I$.

In a concrete execution of G, each task has an execution time within the bounds of its execution interval. The execution times determine (up to non-determinism) the exact execution order of all tasks in the graph for that specific execution. In each concrete execution of G, tasks can have different execution times, so timing analysis of G must therefore result in labeling intervals that express conservative bounds on the timing properties of *any* possible concrete execution of G. The busy interval $B(t)$ of some task t is considered to be conservative if both $En(t)$ and $C(t)$ are conservative bounds for concrete executions.



task	En	C	B
t_1	[0, 0]	[1, 2]	[1, 2]
t_2	[1, 2]	[4, 8]	[3, 6]
t_3	[1, 2]	[8,14]	[7,12]
t_4	[8,14]	[13,20]	[5, 6]
t_5	[13,20]	[20,29]	[7, 9]

(a)



task	En	C	B
t_1	[0, 0]	[1, 1]	[1, 1]
t_2	[1, 1]	[3, 6]	[2, 5]
t_3	[1, 1]	[5, 7]	[4, 6]
t_4	[1, 1]	[4,10]	[3,9]
t_5	[3,6]	[5,31]	[2,25]
t_6	[5,7]	[14,31]	[9,24]
t_7	[4,10]	[7,31]	[3,21]

(b)

Fig. 1. Two example DAGs. In the left DAG G_{NC} there is no resource contention, since the dependencies in G_{NC} enforce a fixed-order schedule. In DAG G_C on the right, there is contention on the white resource.

If B is given, e.g. when there is no contention, the timing of a DAG is calculated by propagating the completion- and enabled intervals of Equations 1 and 2 through the nodes of the graph in topological order, similar to (Homogeneous) SDF-analysis techniques lifted to an interval-timed domain, as shown in the following example.
Figure 1(a) shows an example DAG G_{NC} with five tasks $t_1 \cdots t_5$ bound to two resources p_1(white) and p_2(light grey). The intervals inside the vertices denote the execution interval for each task. G_{NC} has a set of dependencies that enforce a fixed-order schedule ($t_1 \rightarrow t_2$ *and* $t_3 \rightarrow t_4 \rightarrow t_5$). Due to this schedule, any task can start executing immediately when it is enabled, so $B(t) = E(t)$ for all $t \in T$. Starting with task t_1, which is enabled at $En(t_1) = [0, 0]$, execution of t_1 completes at $[1, 2]$, and enables tasks t_2 and t_3 at $[1, 2]$, since both have only a single incoming dependency on t_1. Subsequently, executions of t_2 and t_3 complete at $[4, 8]$ and $[8, 14]$ respectively. Completion of t_3 enables task t_4, which has multiple incoming dependencies, so $En(t_4) = \max\{C(t_2), C(t_3)\} = [8, 14]$. Completion of t_4 at $[13, 20]$ enables t_5 at $[13, 20]$, which completes at $[20, 29]$. The timing labels of G_{NC} are summarized in the figure.

IV. Timing Analysis with Contention

For DAGs with contention, B is not given but needs to be computed. The additional task delay in B due to contention can be estimated by analyzing the relation between enabled intervals of different tasks on the same resource. These enabled intervals in turn depend on B. A natural approach to deal with this recursive dependency is to use a fixed-point iteration. We analyze DAG timing by iterating on B, starting with a B assuming no contention. In each iteration more contention is taken into account, until a fixed-point is reached. To provide some intuition on contention, we first show an example DAG showing contention. Then, in Subsection IV-A, we show how to conservatively estimate task completion intervals given some B. The fixed-point iteration is explained in Section IV-B.

Consider DAG G_C in Figure 1(b). It consists of seven tasks $t_1 \cdots t_7$ mapped to three resources $p1$(white), p_2(light grey) and p_3(dark grey), and has insufficient dependencies between the tasks on resource p_1 to enforce a fixed-order schedule. As a result, some tasks on p_1 will contend for resource access.

Due to the dependency between tasks t_1 and t_4, they do not contend for resource access. Also, t_2, t_3 and t_4 do not contend, since they are mapped to different resources. For these tasks, $B(t) = E(t)$ holds. Initially, task t_1 is the only enabled task, at $[0, 0]$. Completion of t_1 at $[1, 1]$ enables t_2, t_3 and t_4, which then complete at $[3, 6]$, $[4, 7]$ and $[4, 10]$ respectively.

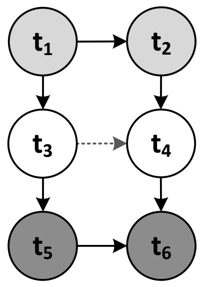
Now, tasks t_5, t_6 and t_7 are enabled at $[3, 6]$, $[5, 7]$ and $[4, 10]$. These tasks suffer from contention, so their busy time is to be determined. In case of resource contention, predecessor completion is not a sufficient precondition for the start of the execution of a contending task; completion of any other tasks that precede it in its execution queue is required as well. In G_C, t_4 is enabled *strictly* before t_5 and t_6, so in *any* concrete execution of G_C, t_4 precedes these tasks. Depending on their concrete enabling time, they may need to wait on completion of t_4 before starting their execution.

Besides the possible delay caused by waiting on completion of t_4, tasks t_5, t_6 and t_7 also contend with each other. Since they have overlapping enabled intervals and are mapped to the same resource, in concrete executions of G_C, these tasks are enabled at any time within the bounds of their enabled intervals. For different concrete executions of the graph, the queueing order of these tasks can be different. These queue orderings can result in different busy and completion intervals for these tasks and any other tasks that depend on them.

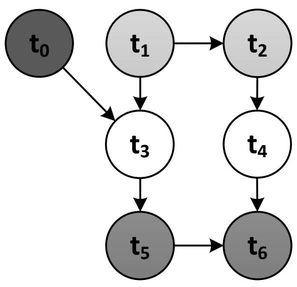
Exact analysis of the timing effects of these different queueing orders requires state-space exploration. In such an approach, analysis time and resource utilization show factorial growth in the number of overlapping tasks in the DAG, rendering an exact analysis intractable already for relatively small DAGs. To ensure scalability, instead of exactly analyzing all different task execution orders, we approximate the best-case and worst-case completion intervals, and combine these in a single closed interval that provides conservative bounds for any concrete execution of the graph.

A. Contention model

Given an initial B, the enabled intervals of the tasks in the DAG can be calculated by evaluating Equations 1 and 2 on tasks in the graph in topological order. If the enabled intervals of all tasks are known, the completion interval of a task t is estimated by analyzing the possible delay caused by tasks mapped to the same resource that can be queued in the execution queue of t's resource before the enabling of t.



(a)



(b)

Fig. 2. Two similar DAGs. In the DAG of 2(a), t_3 precedes t_4 in any execution of the DAG, even if their enabled intervals would overlap, since tasks in $pred(t_4)$ are dependent on *all* tasks in $pred(t_3)$. The DAG of 2(b) has no such precedence relation between t_3 and t_4.

There can be no contention between t and some other task t' if t and t' are dependent, i.e. a directed subset of D connects them, since then one is enabled after completion of the other. Tasks that are enabled strictly earlier than t precede t in any concrete execution, and thus affect t in both best-case and worst-case. Tasks with an enabled interval overlapping with that of t will precede t in only some concrete executions.

Let $ee(t)$ denote the set of tasks which are independent of t, mapped to the same resource and enabled strictly earlier than t, either based on a strictly smaller enabled interval, or because of the dependencies between predecessors of t and t' (see Figure 2). Similarly, $oe(t)$ denotes all tasks which are independent of t, mapped to the same resource, whose enabled interval overlaps with that of t and which are not in $ee(t)$.

The earliest possible completion of t occurs when it is enabled as early as possible, and the start of its execution is delayed as little as possible. This is the case when t is enabled at $L(En(t))$, all tasks in $ee(t)$ complete as soon as possible, and all tasks in $oe(t)$ (except for t itself, which executes at its best-case execution time) are enabled later then t. So, in best-case, t can start executing after its best-case enabling and the best-case completion of the last completing task in $ee(t)$.

The latest possible completion of t occurs when it is enabled as late as possible, and the start of its execution is delayed as much as possible. Given t's worst-case enabling, t is delayed most if all tasks in $ee(t)$ complete at the upper bound of their completion interval, and all tasks in $oe(t)$ execute at the upper bound of their execution interval, while they are enabled just before the upper bound of the enabled interval of t.

Consider the partial DAG and a representation of its execution queue in Figure 3. It consists of 3 independent tasks t_1, t_2 and t_3 mapped to the same resource. The enabled interval of t_1 overlaps with that of t_2 and t_3, and t_2 is enabled strictly before t_3. According to aforementioned completion time estimation, $U(C(t_2)) = U(En(t_2)) + U(E(t_2)) + U(E(t_1))$. Task t_3 must wait for completion of t_2 before it can start its execution, and can be preceded by t_1, so then $U(C(t_3)) = U(C(t_2)) + U(E(t_3)) + U(E(t_1))$. However, since $U(E(t_1))$ is already taken into account in $U(C(t_2))$, it is accounted for twice in $U(C(t_3))$ (case 1 in the figure). To correct this, in the

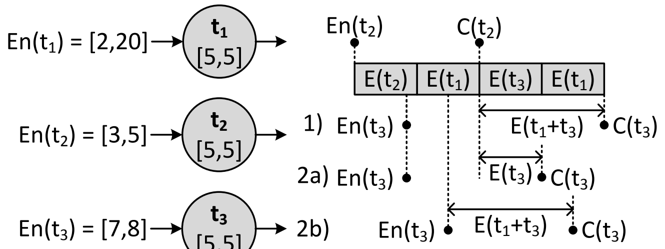


Fig. 3. Example of part of a DAG where the execution time of t_1 is contributing twice to the completion time of t_3. A representation of the execution queue for worst-case timing is shown on the right.

estimation of $U(C(t_3))$, only the execution time of tasks in $oe(t_3)$ that are not in $oe(t_2)$ are to be added (case 2a in the figure). However, if $U(En(t_3)))$ is such that $U(C(t_2)) - U(En(t_2)) \leq U(E(t_1))$, not adding t_1 would lead to an estimate that is not conservative, so in that case $U(C(t_2))$ is estimated with $U(En(t_2)) + U(E(t_1)) + U(E(t_2)) + U(E(t_3))$ (case 2b in the figure). So in general, the corrected worst-case completion time estimate is the maximum of case 2a and 2b in the Figure 3.

With this contention model, the completion interval of a task t given some B is then given by:

$$\begin{aligned} &C(B)(t) = \max(\xi, \zeta), \text{ where} \\ &\xi = En(B)(t') + B(t') + \left(\sum_{t'' \in oe(t) \setminus oe(t')} E(t'') \right) \cup E(t) \\ &\text{with } t' \text{ the last completing task in } ee(t) \\ &\zeta = En(B)(t) + \left(\sum_{t'' \in oe(t)} E(t'') \right) \cup E(t). \end{aligned} \tag{3}$$

The union operator on sets of intervals is defined as $\cup V = [min_{i \in V} L(i), max_{i \in V} U(i)]$ for all finite non-empty $V \subseteq I$.

B. Fixed-point iteration

In this section, we will formulate an iteration on busy intervals, and prove that a fixed point will always be found in a finite number of steps. We start by defining a partial ordering on the interval labelings of Section III.

Definition 2. (POSET ON I^T) *Define relation $\sqsubseteq \subseteq I^T \times I^T$ as follows. For $l_1, l_2 \in I^T$ let $l_1 \sqsubseteq l_2$ if and only if $l_1(t) \subseteq l_2(t)$ for all $t \in T$. It is easy to verify that $(I^T, \sqsubseteq)$ is a poset.*

The iteration will compute busy intervals for each task taking into account the contention on the shared resources. To this end we will define a finite complete lattice on the mappings from tasks to busy intervals. This lattice is contained in poset $(I^T, \sqsubseteq)$ and is defined by explicitly defining a bottom and top element. The bottom element represents the case in which tasks do not contend, while the top element assumes maximal contention.

Definition 3. (FINITE BUSY INTERVAL LATTICE IN I^T) *We define bottom $\bot \in I^T$ and top $\top \in I^T$ by $\bot(t) = E(t)$ and $\top(t) = (\sum_{t' \in \{t'' \in T | M(t'') = M(t)\}}) \cup E(t)$ for all $t \in T$. The* *set $B_L \subseteq I^T$ is defined as $B_L = \{B : T \to I \mid \bot \sqsubseteq B \sqsubseteq \top\}$. It is easy to prove that $(B_L, \sqsubseteq)$ is a finite lattice.*

Any $B \in B_L$ maps tasks in T to their busy interval. Given a busy interval labeling $B^i \in B_L$, a new busy interval labeling $B^{i+1} \in B_L$ can be computed by the following progressive function $F_B(B) : B_L \to B_L$:

$$B^{i+1}(t) = \big(C(B^i)(t) - En(B^i)(t)\big) \cup B^i(t), \qquad (4)$$

where $i_1 - i_2 = [L(i_1) - L(i_2), U(i_1) - U(i_2)]$ for all $i1, i2 \in I$ with $L(i_1) \geq L(i_2)$ and $U(i_1) \geq U(i_2)$. Given B^i, with Equations 1 and 2 $En(B^i)$ is computed. Then, $C(B^i)$ can be computed with Equation 3, taking into account the contention based on $En(B^i)$. A new busy interval is then obtained by calculating $C(B^i) - En(B^i)$ and taking the union with B^i to ensure progressiveness, which is required to guarantee convergence.

Starting with an initial $B(t) = E(t)$ for all $t \in T$ ($\bot$ of B_L), with Equation 4, lattice B_L is traversed from the bottom upwards. At each iteration, busy intervals can grow by taking more contention into account, until a fixed-point is reached. The following theorem shows that the fixed-point algorithm converges in a finite number of iterations.

Theorem 1. *F_B has a fixed-point in B_L given by FIX $F_B = \sqcup\{F_B^n(\bot) \mid n \geq 0\}$, and there is an $m \in \mathbb{N}$ such that $F_B^m(\bot) = FIX\ F$.*

Proof. Trivially, $B \sqsubseteq F_B$ holds, since F_B is constructed by taking the union with B. Since B_L is a lattice, and thus a chain-complete partial order, with the Bourbaki-Witt theorem, F_B has a fixed-point. Since $\hat{B}$ is finite, this fixed-point will be reached in a finite number of steps. □

The following theorem shows that any fixed-point of F_B is conservative.

Theorem 2. *If B is not conservative, then it is not a fixed-point of F_B.*

Proof. Assume that B is not conservative in the i^{th} iteration. Then there exists a first faulty task t_f with either a faulty concrete completion time $C^c(t_f) > U(C(B^i)(t_f))$ or a faulty concrete enabled time $En^c(t_f) > U(En(B^i)(t_f))$. Since it is the first faulty task, the concrete execution of all earlier tasks are within their respective intervals. Since $En^c(t_f)$ is derived from completion times of predecessors which are within their intervals, $En^c(t_f) \leq U(En(B^i)(t_f))$. Hence, $C^c(t_f) > U(C(B^i)(t_f))$. We show that this will be fixed in the $i+1^{th}$ iteration such that $C^c(t_f) \leq U(C(B^{i+1})(t_f))$. Referring to Equation 4, we distinguish two cases in the concrete execution. The first case is when some tasks in $ee(B^i)(t_f)$ are still executing after $En^c(t_f)$. In the concrete execution, there is the last completing task in $ee(B^i)(t_f)$ following without any gaps by 0 or more tasks that are not in $ee(B^i)(t_f)$ ending with t_f. Since $En^c(t_f) \leq U(En(B^i)(t_f))$, these tasks are included in $eo(B^i)(t_f))$ in the evaluation of ξ given in Equation 4. Hence for this case $C^c(t_f) \leq U(C(B^i)(t_f))$. The second case is when t_f is enabled after completion of all tasks in $ee(B^i)(t_f)$. In the concrete execution, there are tasks not in $ee(B^i)(t_f)$ and enabled earlier, ending with t_f. These tasks are included in $eo(B^i)(t_f))$ in the evaluation of ζ given in Equation 4. The remaining execution of these tasks is added to $En^c(t_f)$ which is also within its interval bounds. This leads to $C^c(t_f) \leq U(C(B^i)(t_f))$ in both the cases. Since $C^c(t_f) > U(C(B^i)(t_f))$, transitively $U(C(B^i)(t_f)) \leq U(C(B^{i+1})(t_f))$. Thus, B^i is not a fix-point of F_B. □

Finally, Theorem 3 provides bounds on the fixed-point found by the algorithm.

Theorem 3. *For some DAG $G = (T, D)$ and $t \in T$, let $B_{bounds}(t)$ be defined by $[L(E(t), U(\sum_{t' \in indep(t)} E(t'))]$, with $indep(t) = \{t' \in T \mid M(t') = M(t) \wedge \{(t,t'),(t',t)\} \cap D_{TC} = \varnothing\}$ and $G_{TC} = (T, D_{TC})$ the transitive closure of G. If $B(t)$ is bounded by $B_{bounds}(t)$, then $F_B(B)$ is bounded by $B_{bounds}(t)$ as well.*

Proof. $L(F_B(B)(t)) = L(E(t))$ holds trivially due to the union operation in the definition of B in Equation 4. The upper bound considers two cases. First case is when $U(\xi) > U(\zeta)$ and $U(F_B(B)(t)) = U\left(En(B)(t') + B(t')\right) - U\left(En(B)(t)\right) + U\left(\sum_{t'' \in eo(B)(t) \setminus eo(B)(t')} E(t'')\right)$ where t' is the last completing task in $ee(B)(t)$. The term $U(En(B)(t') + B(t')) - U(En(B)(t))$ is only computed using tasks in $indep(t)$. This is because $U(En(B)(t'))$ is derived from tasks that t' depends on, and $U(B(t'))$ is derived from tasks not in $U(En(B)(t'))$ but in $eo(B)(t')$. Hence, $U(En(B)(t') + B(t'))$ is not computed using duplicates of the execution time of any task. $U(En(B)(t))$ is computed using all tasks having a dependency to t. As a result, the difference $U(En(B)(t') + B(t')) - U(En(B)(t))$ is not computed using any tasks with dependencies to t. It is also not computed using any tasks dependent on t since those have not been enabled yet to contribute to any of the three constituent terms. Tasks in $eo(B)(t)$ can neither have a dependency to t since they are completed before $U(En(B)(t))$, or have dependencies from t since those are enabled only after the completion of t. The term $U\left(\sum_{t'' \in eo(B)(t) \setminus eo(B)(t')} E(t'')\right)$ adds tasks that are in $eo(B)(t)$ but not in $eo(B)(t')$ thus avoiding duplicates. Therefore, $U(F_B(B)(t))$ is computed using tasks in $indep(t)$, without duplicates. In other words $U(F_B(B)(t))$ is bounded by $U(B_{bounds}(t))$. Second case is when $U(\xi) < U(\zeta)$ and $U(F_B(B)(t) = U\left(\sum_{t'' \in eo(B)(t)} E(t'')\right)$ since $U\left(En(B)(t) - En(B)(t)\right) = 0$. As already explained, tasks in $eo(B)(t)$ are in $indep(t)$, so also in this case $U(F_B(B)(t))$ is bounded by $U(B_{bounds}(t))$. □

V. Experimental Evaluation

In this section, the presented approach is applied to a realistic industrial-scale analysis problem, and compared to two other approaches: SDF analysis without shared resources

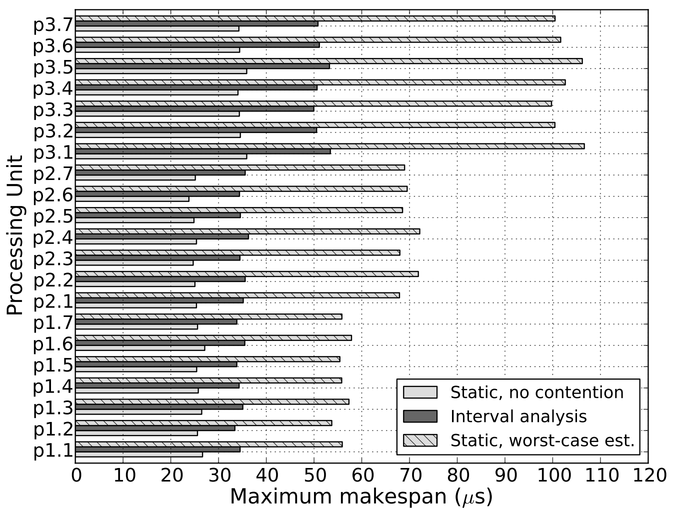


Fig. 4. Makespans of a DAG modeling an industrial-sized control application in a wafer scanner, mapped to 3 octo-core processors.

by neglecting any contention on shared resources, and by worst-case contention analysis.

The analyzed DAG is a 6 degree of freedom digital control application that controls an imaging subsystem in a commercial wafer scanner. It consists of 2285 tasks mapped to a platform with three octo-core processors. The 8th core on each processor is reserved for other processing purposes. The model is calibrated with time measurements obtained by measuring block execution timing in isolation on the machine.

The DAG contains a set of dependencies that enforce a fixed-order schedule on each processor resource. It has lots of dependencies between tasks mapped to different cores; to model core-to-core (c2c) communications, which occur through shared L3 cache, 5377 c2c tasks are automatically added to the DAG, and mapped to 3 (FCFS) resources, one for each processor. Each c2c task has an assumed execution interval of $[5, 5]$ns. The decorated DAG is analyzed using interval analysis, and compared to a standard SDF analysis.

Since SDF analysis requires monotonic models, any contention between c2c tasks in the DAG can be only analyzed statically, i.e. either under the assumption that there is no contention at all, or under assumption of worst-case contention. In the first case, c2c tasks keep their assumed 5ns execution time. In the second case, the worst-case contention is determined by counting the number of tasks mapped to each shared resource, and then for each task t, subtracting the number of tasks on the same resource with which t has a dependency relation and the number of tasks that will precede t based on a dependency relation between predecessors (see Figure 2(a)). This yields the number of potentially interfering tasks for each task on a shared resource. The execution time of that task is scaled accordingly. In both static cases, the DAG is analyzed with standard max-plus algebra that is used in SDF analysis.

Figure 4 shows the maximum makespan (the worst-case completion of the last scheduled task) for each processing unit, obtained by interval analysis, static SDF analysis neglecting contention, and static SDF analysis assuming worst-case contention. The model is analyzed in less than 10 seconds. The analysis results show that, on average, the worst-case makespan calculated with interval analysis is 46% lower compared to static worst-case analysis. On average, the makespans obtained with our analysis are 40.2% above the makespans where no contention is assumed. Part of this difference is caused by contention and part by over-estimation; exactly in which proportion is not easy to pinpoint.

This case shows that our analysis can easily handle large-scale models in the order of thousands of contending tasks, and shows a significant improvement in the provided worst-case bounds compared to static worst-case analysis.

VI. Conclusion

We proposed a new analysis method to obtain bounds on worst-case timing properties of acyclic task graphs mapped to platforms containing shared resources with a FCFS scheduling policy, where variation on task execution times is allowed. With an industrial case study we show that our approach can easily analyze models with thousands of tasks in less than 10 seconds. The calculated worst-case bounds obtained with interval analysis show an average improvement of 46% compared to bounds calculated by static worst-case analysis.

References

[1] R. Frijns, A. Kamp, S. Stuijk, J. Voeten, M. Bontekoe, K. Gemei, and H. Corporaal, “Dataflow-based multi-asip platform approach for digital control applications,” in *Euromicro conf. on Digital System Design, DSD*, 2013.

[2] S. Adyanthaya, M. Geilen, A. Basten, R. Schiffelers, B. Theelen, and J. Voeten, “Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems,” in *Euromicro conf. on Digital System Design, DSD*, 2013.

[3] R. Makowitz and C. Temple, “Flexray - a communication network for automotive control systems,” in *Factory Communication Systems, 2006 IEEE International Workshop on*, 2006, pp. 207 –212.

[4] S. Fuller, *RapidIO: The embedded system interconnect*. Wiley, 2005.

[5] E. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[6] M. Hendriks and M. Verhoef, “Timed automata based analysis of embedded system architectures,” in *International Parallel and Distributed Processing Symposium, IPDPS*, 2006.

[7] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[8] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, “Iteration-based trade-off analysis of resource-aware sdf,” in *Euromicro Conference on Digital System Design, DSD*, 2011, pp. 567–574.

[9] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell,” *Int’l J. on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[10] J. Xing, B. Theelen, R. Langerak, J. Pol, J. Tretmans, and J. Voeten, “Uppaal in practice: Quantitative verification of a rapidio network,” in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. LNCS. Springer, 2010, vol. 6416, pp. 160–174.

[11] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, “Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems,” in *Proc. of the 10th ACM Int’l Conf. on Embedded Software, EMSOFT*, 2012, pp. 63–72.

[12] L. Thiele, S. Chakraborty, and M. Naedele, “Real-time calculus for scheduling hard real-time systems,” in *IEEE Int’l Symp. on Circuits and Systems, ISCAS*, vol. 4, 2000, pp. 101–104.

[13] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, “System architecture evaluation using modular performance analysis: A case study,” *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 6, pp. 649–667, Oct 2006.