Reconstructing Critical Paths from Execution Traces

Martijn Hendriks
Embedded Systems Institute
Eindhoven, The Netherlands
Email: martijn.hendriks@esi.nl

Frits W. Vaandrager
Radboud University Nijmegen
Nijmegen, The Netherlands
Email: frits.vaandrager@cs.ru.nl

***Abstract*—We consider the problem of constructing critical paths from incomplete information. In general, a directed acyclic graph of tasks with their execution times (i.e., a *task graph*) is necessary to extract critical paths. We assume, however, that only the set of tasks, and their start and end times are known, e.g., an execution trace in the form of a Gantt chart. This information can be extracted from real machines or from the output of analysis tools, whereas extraction of the exact task graph often is problematic due to imperative modeling formalisms and complicated platform semantics (resource allocation, varying execution speeds). We show that, based on start and end times only, an over-approximation of the critical paths of an unknown task graph can be extracted nevertheless. Furthermore, this approach is generalized to deal with "noisy" execution traces of real machines in which control overhead is present. Finally, we discuss various methods to deal with false positives, and apply our approach to a complex industrial case study.**

***Index Terms*—Embedded systems, critical path analysis, task graph.**

I. Introduction

Critical path analysis originates from the project scheduling domain in which it is used to create project schedules that minimize the project duration [2], [3]. A project is represented by a directed acyclic graph of *tasks* that need to be executed for the project. The edges between tasks express *precedence constraints*. Tasks not related by precedence constraints can be executed in parallel. Furthermore, every task has a known *execution time* associated with it. A task can start at the moment that all its predecessors have finished. Starting a task as early as possible yields the schedule with the minimal makespan (assuming that a task can always obtain the resources it needs) [4]. The critical path method provides an algorithm to compute the so-called *float* of each task. This is the amount with which the execution time of a task can be increased without increasing the makespan. A task is *critical* if and only if it has 0 float.

Example 1: Figure 1 shows a small task graph. The critical path algorithm as defined in [2] annotates each task in a first pass with its earliest start time. This gives the minimal makespan of the task graph. For instance, the earliest start time of A equals 0 because it has no predecessors. The earliest start time of B equals the earliest start time of A plus the execution time of A. The earliest start time of E equals the maximum of (i) the earliest start time of D plus 1, and (ii) the earliest start time of C plus 5. The minimal makespan equals 8. Using the minimal makespan, a second backwards pass through the task graph is made to compute the latest starting times. For instance, the latest starting time of F equals 7 because in that case the total makespan still equals 8. The critical paths consist of the tasks which have 0 float, i.e., those tasks whose earliest and latest starting time are equal. (Note that there can be multiple critical paths.) In this example, the path $A \to C \to E$ is critical.

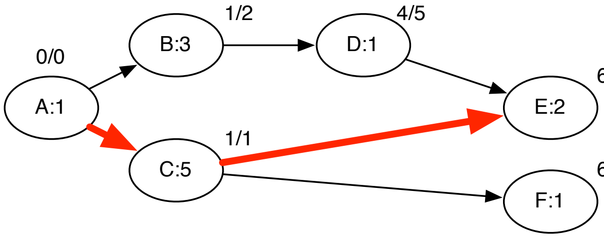


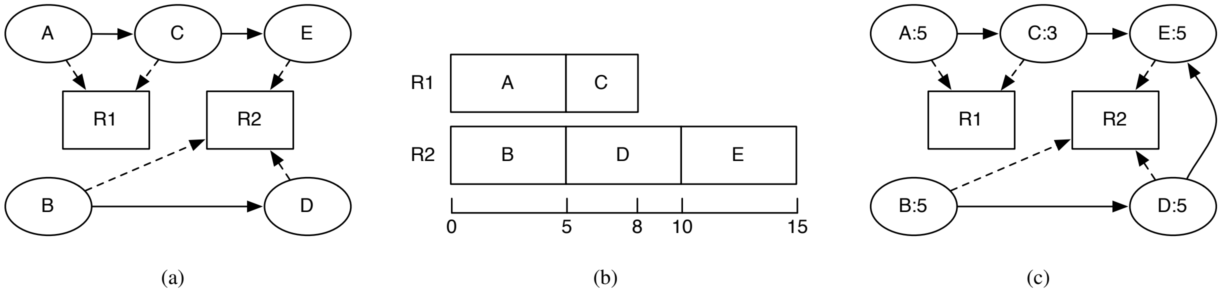
Fig. 1. Critical path analysis of a small task graph. Execution times are shown after the task names. The task are annotated with a/b where a is the earliest start and b is the latest start. The only critical path is $A \to C \to E$.

Critical paths provide important information on the system's bottlenecks. Therefore, being able to show the critical paths in an execution trace is a valuable means to diagnose performance problems.

A precondition of critical path analysis is that a task graph is at hand. This is not always the case. Even in a model-based setting, the availability of a system model does not automatically mean that task graphs are readily at hand. This is explained below. Execution environments such as real systems or simulation engines, however, generally can produce execution traces that contain (i) the start time of the tasks and (ii) the execution times of the tasks. These can be represented in a graphical way using, e.g., a Gantt chart [5]. This situation is illustrated by the following example.

Example 2: Figure 2(a) shows a typical representation of a system model. The application is represented by a set of dependent tasks A, B, C, D and E, which are mapped to two resources, $R1$ and $R2$. Typically, the execution times of the tasks are derived by the execution engine and also depend on the mapping and platform properties. Figure 2(b) shows the output of the execution engine as a Gantt chart. Figure 2(c)

This work was carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program. An earler version of this paper without proofs appeared as [1].



(a) (b) (c)

Fig. 2. A simple Y-chart based system model in which 5 tasks are mapped to two resources. Figure (a) shows the input for the execution environment, (b) shows the output of the execution engine as a Gantt chart, and (c) shows the task graph that the execution engine constructed internally. Note that it resolved the resource conflict on $R2$ by an additional precedence $D \to E$.

shows the task graph that the execution engine constructed internally. The tasks in Figure 2(c) are labeled with their execution time, and there is an additional precedence, $D \to E$, which is the result of platform semantics: tasks D and E cannot run in parallel because they both need resource $R2$.

Example 2 touched the reasons why a model-based environment does not automatically give straightforward access to task graphs. Although system models typically are close to the task graphs that are needed to extract critical paths, they usually do not contain enough information to do this because the platform semantics is missing:

1) The exact execution time of tasks is not known because this may depend on the run-time state of the system. For example, tasks that use a hard disk or data bus may slow each other down if they run concurrently.
2) The model may contain conditional execution rules that depend on the run-time state of the system. Caching rules have this effect, for instance.
3) The model does not contain precedences between tasks that are the consequence of their execution on the same non-preemptive resource. If memory is scarce, for instance, then it may happen that a task must wait for another task to release its memory.

As a consequence, the task graph needed for critical path analysis can only be constructed by the execution environment. We regard the execution environment as a black box and the task graph is thus not available.

A. Contribution

We present an algorithm to reconstruct an approximation of a task graph from a system's execution trace which only needs to contain start times of tasks and their execution times. Application of the classical critical path algorithm to this task graph yields an *over-approximation* of the set of critical tasks. We extend our algorithm to deal with traces from real machines, which are not perfect in the sense that the timing behavior contains control overhead that prevents a direct application of our first algorithm. Finally, we discuss three approaches to the problem of false positives. Our algorithms, which easily scale to hundreds of thousands of tasks, have been added to the OCTOPUS toolset [6], and have been applied to the design space exploration of industrial printing systems.

Existing literature describes various approaches in different domains that reconstruct task graphs from execution traces of the system (see next section). Our main contribution is a *general* and *formalized* approach to this problem that has been successfully applied to industrial problems and which – as far as we know – has not been presented before.

B. Related work

Critical path analysis originated in the context of project planning and is used to create project schedules that minimize the project duration [2], [3]. Critical path analysis did originally not consider resource constraints although it has been acknowledged almost immediately that these constraints are important [7]. Several methods exist that take resources into account such as resource leveling [8] and the critical chain method [9]. Often, however, finding a schedule with a minimum makespan for projects with resource constraints is an NP hard scheduling problem [4]. In contrast to the application of critical path analysis in the project management area, our (and other) work uses critical path analysis *after* the system executed in order to analyze and improve it.

Within the embedded systems domain critical path analysis is used in scheduling or mapping heuristics [10], [11], [12]. The existence of a task graph is assumed in [10], [11] and thus differs in this respect from the problem of our work. The work in [12] probably is most closely related to our work. The authors reconstruct a task graph from a schedule and their approach is similar to ours. However, construction of the critical path is not the main subject of that work and is presented in an ad-hoc and informal way. Moreover their work does neither address the issue of false positives, nor the issue of optimizations that are required for scalability. Critical path analysis is also used in the profiling of distributed applications [13], [14], [15], [16]. The task graph is constructed from actual executions of the system, for instance, by intercepting system calls of the application under consideration. At least partial knowledge of the system task graph is assumed in these works. The present paper provides an alternative and formalized approach that only needs start and end times of tasks and therefore is widely applicable.

Our techniques are related to process or workflow mining (see, e.g., [17] for a survey). Van der Aalst and Van Dongen present a workflow mining technique in [18] that includes timing information. This is used to measure the performance in the workflow model. The nature of the constructed models is different from the models in our work. Furthermore, the mining techniques usually work on sets of traces whereas our technique is typically applicable to a single trace.

C. Outline

Section II presents basic definitions of critical path analysis and the algorithm that constructs an over-approximation of the critical paths from an execution trace that only contains start and execution times of tasks. Unfortunately, this algorithm may not be directly applicable to traces from real machines. Section III presents a modification of the algorithm to overcome this problem. The problem of false positives (due to the over-approximation) is discussed in Section IV. Section V presents an industrial case study. Finally, conclusions are presented in section VI.

II. Critical Path Analysis

This section contains the mathematical model used for critical path analysis. It starts with a subsection with definitions, lemmas and proofs which have already appeared in some form somewhere in the extensive literature that exists on this subject, see for instance [2], [4]. The other subsections contains our main contribution: an algorithm to reconstruct critical paths from execution traces without prior knowledge of the precedence relation.

A. Preliminaries

A task graph is a directed acyclic graph in which every task has a duration or execution time. Note that we allow a zero execution time for a technical reason which will become clear later.

Definition 1 (Task Graph): A *task graph* is a tuple $\mathcal{G} = (T, \dashrightarrow, d)$ with T a finite set of tasks, $\dashrightarrow \subseteq T \times T$ an acyclic precedence relation, and $d : T \to \{x \in \mathbb{R} \mid x \geq 0\}$.

If we assume that parallel tasks do not influence each other, then the fastest way to execute a task graph is to execute the tasks as soon as they are enabled (i.e., as soon as their predecessors have finished). In scheduling theory the problem to compute such a schedule is known as the parallel machine problem with infinitely many machines [4]. In practice, an execution environment will often construct a task graph internally, either implicitly or explicitly, in order to adhere to platform semantics, and output the *greedy* or *non-delay interpretation* of that task graph (defined formally in Def. 4 below). The task graph in Figure 2(c), for instance, has the schedule shown in Figure 2(b) as non-delay interpretation.

The situation thus is that there are task graphs, but they are invisible since they only exist (implicitly) inside the execution environment, which is a black box. We only have access to the non-delay interpretation of the task graph in the form of a start times and execution times of tasks.

The remainder of this subsection presents basic definitions and lemmas regarding critical path analysis of task graphs, see for instance [2], [4].

Definition 2 (Critical Path): Let $\mathcal{G} = (T, \dashrightarrow, d)$ be a task graph. A *path* of $\mathcal{G}$ is a sequence $\pi = t_1 \cdots t_n$ of tasks in T such that, for all $i < n$, $t_i \dashrightarrow t_{i+1}$. We say that π is a path from t_1 to t_n. The *duration* of path π, denoted $d(\pi)$, is defined to be $d(t_1) + d(t_2) + \cdots + d(t_n)$. If $n = 0$, that is, π is the empty path, then $d(\pi) = 0$. A path is *critical* if its duration is maximal amongst the set of all paths of $\mathcal{G}$.

Next, the concept of a sink task is introduced, which makes the formalizations easier in some places. A sink task has a zero execution time, which motivates the decision to allow tasks with zero execution time in Def. 1.

Definition 3 (Sink task): Let $\mathcal{G} = (T, \dashrightarrow, d)$ be a task graph. We say that a task $\mathsf{sink} \in T$ is a *sink task* of $\mathcal{G}$ if $d(\mathsf{sink}) = 0$ and, for each $t \in T \setminus \{\mathsf{sink}\}$, there exists a path from t to sink.

Note that if $\mathcal{G}$ does not have a sink task, we can always add such a task sink with precedences $t \dashrightarrow \mathsf{sink}$, for any $t \in T$. Then π is a critical path of T iff $\pi\ \mathsf{sink}$ is a critical path of the extended task graph. Furthermore, there can never be more than one sink task since that would contradict the acyclic nature of task graphs.

Algorithm 1 computes the earliest and latest starting times of tasks such that the minimal latency is achieved. It is a slightly rephrased version of the procedure described in [2] and [4]. Example 1 shows an example. We use the convention that the maximum over an empty set of numbers equals 0.

Algorithm 1 Critical path analysis of task graph $(T, \dashrightarrow, d)$.

```
Require: a topological ordering $t_1, t_2, \ldots, t_n$ of $T$ according
    to $\dashrightarrow$, and that there is a sink task, i.e., $t_n$.
for $i = 1$ to $n$ do
    $start^-(t_i) = max(\{start^-(t) + d(t) \mid t \dashrightarrow t_i\})$
end for
$start^+(t_n) = start^-(t_n)$
for $i = n - 1$ to $1$ do
    $start^+(t_i) = min(\{start^+(t) - d(t_i) \mid t_i \dashrightarrow t\})$
end for
Ensure: $start^-$ and $start^+$ have been defined for all tasks
```

The constructed $start^-$ function gives the non-delay interpretation of the task graph which is straightforwardly representable as a Gantt chart.

Definition 4 (Non-delay interpretation): Let $\mathcal{G} = (T, \dashrightarrow, d)$ be a task graph. The non-delay interpretation of $\mathcal{G}$ is the tuple $(T, start^-, d)$, where $start^-$ is obtained through Algorithm 1.

The next four lemmas establish some basic technical properties of the $start^-$ and $start^+$ functions.

Lemma 1: Let $t_i \in T$. Then a path π to t_i exists with $start^-(t_i) + d(t_i) = d(\pi)$. Furthermore, there is no path to t_i with a larger duration.

Proof: By induction on i. ■

Lemma 2: $\mathit{start}^-(\mathsf{sink})$ is the duration of the critical paths of $\mathcal{G}$.

Proof: By Lemma 1 a path π exists to sink such that $\mathit{start}^-(\mathsf{sink}) = d(\pi)$, and there is no path to sink with a larger duration. However, by definition of a sink task, any critical path can be extended to a path (which hence also is critical) to sink. Hence π is critical. ■

Lemma 3: Let $t_i \in T$. Then there exists a path π from t_i to sink such that $\mathit{start}^+(t_i) + d(\pi) = \mathit{start}^+(\mathsf{sink})$. Furthermore, there is no path from t_i with a larger duration.

Proof: By induction on $n - i$. ■

Lemma 4: $\mathit{start}^-(\mathsf{sink}) = \mathit{start}^+(\mathsf{sink})$

Proof: Immediate from the definition of the algorithm and the fact that sink is a sink state and therefore the last element in the topological order. ■

The next proposition establishes that choice of the topological order, which is an input for the algorithm, does not matter: in all cases *start*$^-$ and *start*$^+$ are identical.

Proposition 1: For a given task graph, the values of *start*$^-$ and *start*$^+$ do not depend on the actual topological ordering of the tasks.

Proof: By Lemma 1, $\mathit{start}^-(t)$ does not depend on the selected topological ordering, for any $t \in T$. In particular, $\mathit{start}^-(\mathsf{sink})$ does not depend on the topological ordering and hence, by Lemma 4, also $\mathit{start}^+(\mathsf{sink})$ does not depend on it. Now we use Lemma 3 to conclude that $\mathit{start}^+(t)$ does not depend on the selected topological ordering, for any $t \in T$. ■

The *float* of a task t_i is defined as $\mathit{start}^+(t_i) - \mathit{start}^-(t_i)$. In the literature, criticality of tasks is defined both in terms of float and in terms of being part of the longest path. We use the latter definition, and the next proposition asserts the equivalence.

Proposition 2: A task is on a critical path if and only if it has zero float[1].

Proof: Let $t_i \in T$ be a task. By Lemma 1, there exists a path π to t_i satisfying $\mathit{start}^-(t_i) + d(t_i) = d(\pi)$. By Lemma 3, there exists a path π' from t_i to sink satisfying $\mathit{start}^+(t_i) + d(\pi') = \mathit{start}^+(\mathsf{sink})$. By concatenating π and π', and removing the duplicate occurrence of t_i, we obtain a path π'' on which t_i occurs with duration $d(\pi) + d(\pi') - d(t_i)$. By Lemmas 1 and 3, there is no path on which t_i occurs with a larger duration. We infer

$$\begin{aligned} d(\pi'') &= d(\pi) + d(\pi') - d(t_i) \\ &= (\mathit{start}^-(t_i) + d(t_i)) + \\ &\quad (\mathit{start}^+(\mathsf{sink}) - \mathit{start}^+(t_i)) - d(t_i) \\ &= \mathit{start}^+(\mathsf{sink}) + \mathit{start}^-(t_i) - \mathit{start}^+(t_i) \end{aligned}$$

Assume that t_i has zero float, that is, $\mathit{start}^+(t_i) = \mathit{start}^-(t_i)$. Then $d(\pi'') = \mathit{start}^+(\mathsf{sink})$ and hence, by Lemmas 2 and 4, path π'' is critical.

Now assume that t_i occurs on a critical path. Then, since there is no path on which t_i occurs with a duration larger than π'', π'' is critical. Using again Lemmas 2 and 4, $d(\pi'') = \mathit{start}^+(\mathsf{sink})$. Hence $\mathit{start}^+(t_i) = \mathit{start}^-(t_i)$, that is, t_i has zero float. ■

The concept of an artificial sink task is needed for formalizations concerning Algorithm 1. In the remainder of this paper we assume that no artificial sink task has been added to the task graphs.

B. Approximation of the task graph

As explained in the introduction, execution environments apply platform semantics to obtain a task graph. This semantics often includes mutual exclusion type rules which form implicit precedence rules. However, their output is not that task graph, but rather an execution trace. We further assume that the execution environment adheres to the non-delay interpretation of this internal task graph when it produces the execution trace. This is to say that the tasks are started on their earliest starting times. Thus, the non-delay interpretation (T, start^-, d) of the task graph is available from the output of the execution engine. However, extraction of the critical paths requires the precedence relation $\rightarrow$ of the task graph. Algorithm 2 extracts a set of precedence edges from the start and end times of tasks which is sufficient to recreate the critical paths of the underlying but unknown task graph.

Algorithm 2 Approximation of a task graph.

```
Require: A non-delay interpretation $(T, \mathit{start}^-, d)$.
1: $\mapsto = \emptyset$
2: for all $t \in T$ do
3:    for all $t' \in T$ do
4:       if $t \neq t' \wedge \mathit{start}^-(t) + d(t) = \mathit{start}^-(t')$ then
5:          $\mapsto = \mapsto \cup \{(t, t')\}$
6:       end if
7:    end for
8: end for
```

The condition in line 4 of Algorithm 2 only holds for pairs of non-equal tasks. Therefore, it is clear that there will be no self-loops in the computed precedence relation. Furthermore, there is a precedence from a task t to another task t' if and only if the end of t coincides with the start of t'. The following properties thus hold:

$$t = t' \implies \neg(t \mapsto t') \quad (1)$$

$$t \neq t' \implies (t \mapsto t' \iff \mathit{start}^-(t) + d(t) = \mathit{start}^-(t')) \quad (2)$$

In order to apply the critical path algorithm, however, the resulting structure $(T, \mapsto, d)$ must be a task graph, i.e., $\mapsto$ must be acyclic.

Proposition 3: The $\mapsto$ relation as created by Algorithm 2 is acyclic if $d(t) > 0$ for all $t \in T$.

[1] This proposition is well-known, see for instance, [19] where a proof is sketched for the equivalent model of PERT charts.

Proof: Suppose that there is a cycle, say the sequence $t_1 \mapsto t_2 \mapsto \cdots \mapsto t_n \mapsto t_1$. Then the following equations hold: $\mathit{start}^-(t_1) + d(t_1) = \mathit{start}^-(t_2)$, . . . , $\mathit{start}^-(t_n) + d(t_n) = \mathit{start}^-(t_1)$. Repeated substitutions then give that $d(t_1)+d(t_2)+\cdots+d(t_n) = 0$. Our assumption that $d(t) > 0$ for all $t \in T$ then results in a contradiction. ■

Algorithm 2 thus produces a directed acyclic graph if the tasks have a non-zero execution time. The $\mapsto$ relation that is constructed, however, is in general neither a subset nor a superset of $\rightarrow$. This is shown by the next example.

Example 3: Consider the system in Example 2. The execution environment does not output a task graph such as shown in Fig. 3(a), but rather the non-delay interpretation of this task graph, shown in Fig. 3(b) as a Gantt chart. Fig. 3(c) shows the $\mapsto$ relation as extracted from the Gantt chart in Fig. 3(b) by Algorithm 2 (the critical paths are marked by fat red precedences). Note that there are spurious precedences such as $A \mapsto D$ and $B \mapsto C$ which can result in spurious critical paths. There is also a missing precedence: $C \mapsto E$. Note that the precedence $D \mapsto E$ is not spurious as it results from sharing a resource. Of the two critical paths only $B \mapsto D \mapsto E$ is a real critical path, and $A \mapsto D \mapsto C$ is the result of the spurious precedence $A \mapsto D$.

Although the $\mapsto$ relation may be quite different from the precedence relation of the original task graph, it is conservative with respect to critical paths. This is formally stated in Theorem 1 below. In order to prove this theorem, we first need to establish a technical lemma.

Lemma 5: Let $\pi = u_1 \cdots u_m$ be a critical path of a task graph $(T, \rightarrow, d)$. Then $\mathit{start}^-(u_1) = 0$ and $\mathit{start}^-(u_{i+1}) = \mathit{start}^-(u_i) + d(u_i)$, for $1 \leq i < m$.

Proof: Suppose that $u \mapsto u_i$, for some task u and index i. Then, by Lemma 1, there exists a path π' to u such that $\mathit{start}^-(u) + d(u) = d(\pi')$. Since path π is critical, $d(\pi') \leq d(u_1) + \cdots + d(u_{i-1})$ (otherwise we could concatenate π' and $u_i \cdots u_m$, to obtain a path with a larger duration than π, thus contradicting the assumption that π is critical). Hence,

$$\mathit{start}^-(u) + d(u) \quad \leq \quad d(u_1) + \cdots + d(u_{i-1}) \qquad (3)$$

We now prove that $\mathit{start}^-(u_i) = d(u_1) + \cdots + d(u_{i-1})$ by induction on i.

For the base case $i = 1$, suppose $u \mapsto u_1$ for some task u. Then, by inequality (3), $\mathit{start}^-(u) + d(u) = 0$. Hence, by definition of Algorithm 2, $\mathit{start}^-(u_1) = 0$.

Now suppose that $i > 1$. By induction hypothesis and inequality (3) we have, for each u with $u \mapsto u_i$, $\mathit{start}^-(u) + d(u) \leq \mathit{start}^-(u_{i-1}) + d(u_{i-1})$. Hence, by definition of Algorithm 2, $\mathit{start}^-(u_i) = \mathit{start}^-(u_{i-1}) + d(u_{i-1}) = d(u_1) + \cdots + d(u_{i-1})$.

From these equalities, the lemma follows directly. ■

Theorem 1: Let $\mathcal{G}$ be a task graph, let $\mathcal{I}$ be its non-delay interpretation, and let $\mathcal{G}'$ be the output of Algorithm 2 when run with $\mathcal{I}$. A critical path of $\mathcal{G}$ is also a critical path of $\mathcal{G}'$.

Proof: Consider a task graph $\mathcal{G} = (T, \rightarrow, d)$ with its non-delay interpretation $\mathcal{I} = (T, \mathit{start}^-, d)$ and with a critical path $\pi = t_1 t_2 \cdots t_m$. Let $\mapsto$ be computed as in Algorithm 2 from $\mathcal{I}$. By Lemma 5 and property (2), the path π is also present in $\mapsto$. Next, we prove by contradiction that no longer path in $\mapsto$ exists which makes π critical in $\mapsto$. Consider a longer path $\pi' = t'_1 t'_2 \cdots t'_n$. Because it has been assumed that π' is longer than π it also holds that $\mathit{start}^-(t'_n) + d(t'_n) > \mathit{start}^-(t_m) + d(t_m)$. This means that in the task graph task t'_n ends after task t_m and according to Lemma 1 there is a path in the task graph to task t'_n with a length greater than $d(\pi)$. This contradicts that π is a critical path. ■

Example 3 showed that there can indeed be spurious critical paths in the approximation. Clearly, if there is only a single critical path in $(T, \mapsto, d)$, then this critical path is not spurious because a task graph always has at least one critical path. Furthermore, if there are multiple critical paths that all share a common sub-path, then it is certain that this sub-path is part of any real critical path.

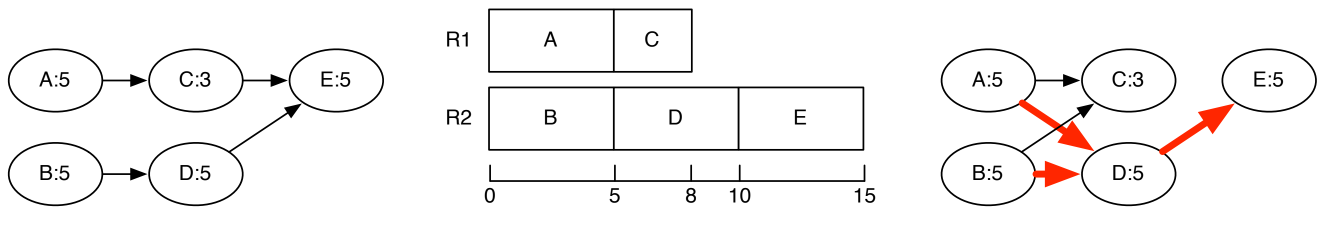
C. Optimization of the algorithm

Algorithm 2 requires a quadratic number of operations w.r.t. the size of the task set. Hence this algorithm does not scale well to large sets of tasks. (In the case studies we did, traces with hundreds of thousands of tasks need to be analyzed). However, if the tasks are ordered according to their starting times (which often is a natural ordering of events), then this can be used to optimize the algorithm. Algorithm 3 is the optimized version of Algorithm 2.

Algorithm 3 Optimized approximation of a task graph.

```
Require: A non-delay interpretation $(T, \mathit{start}^-, d)$ and a sequence of all tasks $t_1, t_2, \ldots, t_n$ such that $i < j \implies \mathit{start}^-(t_i) \leq \mathit{start}^-(t_j)$.
 1: $\mapsto = \emptyset$
 2: $i = 1$
 3: while $i < n$ do
 4:   $j = i + 1$
 5:   $\mathit{next} =$ true
 6:   while $j \leq n \wedge \mathit{next}$ do
 7:     if $\mathit{start}^-(t_i) + d(t_i) = \mathit{start}^-(t_j)$ then
 8:       $\mapsto = \mapsto \cup \{(t_i, t_j)\}$
 9:     else if $\mathit{start}^-(t_i) + d(t_i) < \mathit{start}^-(t_j)$ then
10:       $\mathit{next} =$ false
11:     end if
12:     $j = j + 1$
13:   end while
14:   $i = i + 1$
15: end while
```

The inner loop in lines 6 – 13 only inspects tasks which are started at the same time or later than the subject task t_i. Lines 9 and 10 implement a heuristic to stop inspecting tasks that are started strictly later than the subject task t_i ends. Although theoretically this algorithm still inspects $|T|^2$ combinations of



(a)

(b)

(c)

Fig. 3. Figure (a) shows a task graph, Figure (b) shows the non-delay interpretation of this task graph as a Gantt chart, and (c) shows the $\mapsto$ relation as extracted from this Gantt chart by Algorithm 2.

tasks, it works much better in practice than always comparing all combinations of tasks.

III. Dealing with Execution Traces of Real Systems

Algorithm 2 presented in the previous subsection approximates the task graph from its non-delay interpretation in order to enable the critical path analysis of Algorithm 1. The property of the non-delay interpretation that there is no idle time between successive tasks on the critical path is exploited for this. This works fine in a model-based setting, because then the supervisory control can be assumed to be infinitely fast in the models. If real system traces are considered, however, there will typicaly be some idle time between successive tasks because of the control overhead. Furthermore, not all tasks may be represented in the execution trace. Algorithm 2 fails for such traces because then the condition in line 4 is never satisfied for tasks that are related by a precedence constraint. Note that the $\mapsto$ relation is not necessarily empty after the algorithm ran, because tasks can still be related by timing coincidence.

The possible non-zero idle time between tasks that in reality are related, is due to tasks (e.g., control tasks) that are not represented in the execution trace. The precedence structure with respect to these invisible tasks thus needs to be guessed. The approach that we chose is a generalization of Algorithm 2 with a timing threshold to relate tasks. If some task starts within ϵ time units after the end of another task, then we assume that they are related and insert a new intermediate task as a placeholder for the missing task. Furthermore, two precedences are added to link these tasks. For this purpose, the existence of a set $U = \{u_1, u_2, \ldots\}$ of control-overhead tasks such that $U \cap T = \emptyset$ is assumed. Algorithm 4 shows the approach.

This algorithm extends Algorithm 2 with an additional case that implements the sketched timing threshold approach in lines 7 – 12. Lines 14 – 19 insert control overhead tasks before tasks that have no predecessors yet start later than time 0. Note that for $\epsilon = 0$ this algorithm is identical to Algorithm 2. Furthermore, the optimization that has been applied to Algorithm 2 in Section II-C can also be applied to Algorithm 4. The algorithm produces an acyclic precedence relation if the task durations are greater than 0.

Algorithm 4 ϵ-Approximation of a task graph.

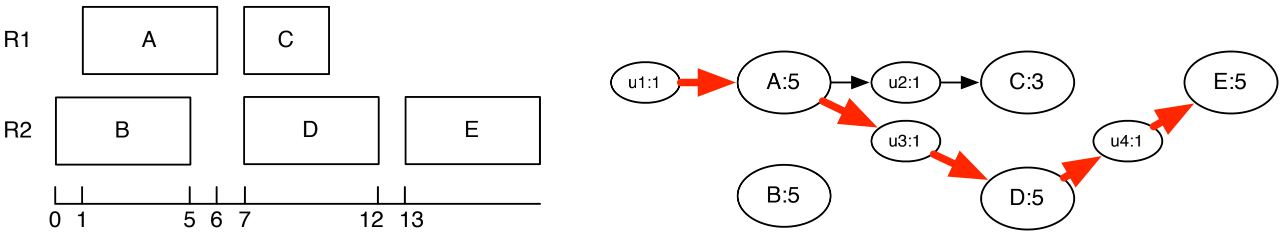
```
Require: A non-delay interpretation $(T, start^{-}, d)$, a set of
    control tasks $U = u_1, u_2, \ldots$ such that $T \cap U = \emptyset$, and a
    real-valued constant $\epsilon > 0$.
 1: $\mapsto = \emptyset$
 2: $i = 1$
 3: for all $t \in T \setminus U$ do
 4:   for all $t' \in T \setminus U$ do
 5:     if $t \neq t' \wedge start^{-}(t) + d(t) = start^{-}(t')$ then
 6:       $\mapsto = \mapsto \cup \{(t, t')\}$
 7:     else if $0 < start^{-}(t') - (start^{-}(t) + d(t)) \leq \epsilon$ then
 8:       $T = T \cup \{u_i\}$
 9:       $\mapsto = \mapsto \cup \{(t, u_i), (u_i, t')\}$
10:       $d(u_i) = start^{-}(t') - (start^{-}(t) + d(t))$
11:       $i = i + 1$
12:     end if
13:   end for
14:   if $0 < start^{-}(t) \leq \epsilon$ then
15:     $T = T \cup \{u_i\}$
16:     $\mapsto = \mapsto \cup \{(u_i, t)\}$
17:     $d(u_i) = start^{-}(t)$
18:     $i = i + 1$
19:   end if
20: end for
```

Proposition 4: The $\mapsto$ relation as created by Algorithm 4 is acyclic if $d(t) > 0$ for all $t \in T$.

Proof: By contradiction. Assume $t_1, t_2, \ldots, t_n, t_1$ is a cycle. Note that this cycle does not contain tasks from U added by lines 14 – 19 because such tasks have no predecessors. The cycle can, however, contain tasks from U added in lines 8 – 11. These tasks, however, are added in such a way that they exactly fill the time gap between two tasks from T. We may therefore use exactly the same argument as in the proof of Proposition 3 to derive a contradiction. ■

Obviously, in order to apply the algorithm to a concrete execution trace, the ϵ must be fixed. If the value of ϵ is not known and underestimated, then wrong results might follow. This is illustrated by the following example.

Example 4: Consider the system in Figure 2(a) and suppose that the control overhead is such that successive tasks have at



(a)

(b)

Fig. 4. Figure (a) shows a noisy Gantt chart of the system in Figure 2(a). Figure (b) shows the result of Algorithm 4 for $\epsilon = 1$. Note that it does not contain the critical path $B \rightarrow D \rightarrow E$ of the underlying task graph.

most 2 timeunits of idle time between them. Figure 4(a) shows a possible execution trace of this system. Note that the tasks A, C, and D, E have 1 timeunit of idle time between them, and that the tasks B, D have 2 timeunits of idle time between them. Also, A starts not immediately due to the controller overhead. Application of Algorithm 4 with $\epsilon = 1$ gives the task graph shown in Figure 4(b). Application of Algorithm 1 gives a critical path that involves task A instead of task B which is wrong according to the original system. If we use $\epsilon = 2$ then the missing precedence $B \rightarrow D$ is detected.

Fixing the right value of ϵ is of crucial importance for the correctness of the output of Algorithm 4. If it is too small, then erroneous output can result. Theorem 2 below states that if the value of ϵ is equal to or larger than the maximal distance between successive tasks, then the output is correct, i.e., an application of critical path analysis will give an over-approximation of the critical paths. Unfortunately, the work presented in this paper does not give a method to determine a safe value for ϵ when given an execution trace only. An indication of the appropriateness of the value of ϵ is whether the non-delay interpretation of the approximated task graph is the same as the *start*$^-$ mapping of the input execution trace. However, even this is not satisfactory as is shown in Figure 4(b). The approximation has a non-delay interpretation which matches the Gantt chart in Figure 4(a), but the real critical path $B \rightarrow D \rightarrow E$ is not present in the approximation.

Theorem 2: Let $\mathcal{G} = (T, \rightarrow, d)$ be a task graph and let $\mathcal{I}$ be its non-delay interpretation. Let $\mathcal{I}'$ be obtained from $\mathcal{I}$ by removing a subset S of tasks from T and from the *start*$^-$ and d functions. Let $\epsilon = max(\{d(\pi) \mid \pi$ is a path in $S\})$, and let $\mathcal{G}'$ be the output of Algorithm 4 when run with $\mathcal{I}'$ and ϵ. If every critical path in $\mathcal{G}$ ends in a task not from S, then a critical path in $\mathcal{G}$ has an equivalent critical path in $\mathcal{G}'$ which are equal when projected to $T \setminus S$.

Proof: We show that a critical path $\pi = t_1 t_2 \cdots t_m$ of $\mathcal{G}$ has an equivalent path π' in $\mathcal{G}'$ in which tasks from S are replaced by tasks from U such that $d(\pi) = d(\pi')$. If this has been established, then an equivalent argument as in the proof of Theorem 1 can be used to show that the assumption that π' is not critical leads to a contradiction.

We use induction on the number of tasks which are not elements of S in π. The base case is the case for 1 such a task, because we assumed that a critical path always ends with a task which no element of S. In this case, $t_1 t_2 \cdots t_{m-1}$ are elements of S. By assumption, the duration of this sequence of tasks is smaller than or equal to ϵ. Thus, $0 < start^-(t_m) \leq \epsilon$ and the condition on line 14 of the algorithm is satisfied. Thus, the path $u_1 t_m$ mimicks π.

Now suppose that we can construct an equivalent path of π in $\mathcal{G}'$ if π has n tasks which are not elements of S. The proof that we now can construct an equivalent path of π in $\mathcal{G}'$ if π has $n+1$ tasks follows the same line of reasoning as the base case, but now using lines 7 – 12 of the algorithm to connect the n-th and $n+1$-th tasks which are not elements of S. ■

Algorithm 4 can be used to compute critical path of noisy execution traces if two conditions are met. First, a safe upper bound on the possible time gaps between dependent tasks that appear in the non-delay interpretation is known in order to determine a safe value for ϵ. Second, the last task of a critical path of the system is also present in the non-delay interpretation. These conditions can typically be met for execution traces of systems where control overhead is small and not shown in the execution trace.

IV. Spurious critical paths

Spurious critical paths distort the critical path view and make interpretation more difficult. There are several ways to deal with this.

A. A theoretical approach

If every path through a task graph has a unique duration, then there will not be spurious critical paths.

Theorem 3: Let $\mathcal{G}$ be a task graph, let $\mathcal{I}$ be its non-delay interpretation, and let $\mathcal{G}'$ be the output of Algorithm 2 when run with $\mathcal{I}$. Supose $d(\pi) \neq d(\pi')$ for all paths π and π' of $\mathcal{G}$. Then $\mathcal{G}'$ has the same critical paths as $\mathcal{G}$.

Proof: By Theorem 1, every critical path of $\mathcal{G}$ is also a critical path of $\mathcal{G}'$. Let π be a critical path of $\mathcal{G}'$. It suffices to prove that π is also a critical path of $\mathcal{G}$.

If we assume that every path in $\mathcal{G}$ has a unique length, then in $\mathcal{G}$ it holds that $start^-(t') = start^-(t) + d(t) \implies t \mapsto t'$. This can be understood by the observation that since path lengths are unique, by Lemma 1 no other task than t ends at time $start^-(t) + d(t)$. Thus by line 2 of Algorithm 1 we have that

$t \to t'$. Algorithm 2 thus only adds edges to $\mathcal{G}'$ that are also edges of $\mathcal{G}$. Thus π is also a path of $\mathcal{G}$. Now suppose that π is not critical in $\mathcal{G}$. Then there exists a critical path π' of $\mathcal{G}$ with $d(\pi') > d(\pi)$. By Theorem 1, π' is also critical in $\mathcal{G}'$. But this contradicts the assumption that π is critical in $\mathcal{G}'$. Hence π is critical in $\mathcal{G}$. ■

An execution engine might be able to ensure that every path has a unique duration by adding a very small and unique value to the execution time of each task. For instance, suppose that every task has an integer duration, then the execution time of a task t can be increased by a decimal fraction $2^{-b(t)}$, where $b : T \to \mathbb{N}$ gives a unique index to each task: $t \neq t' \implies b(t) \neq b(t')$. This approach is not very practical as a bit is needed for every task. For instance, if we have a system with 10,000 tasks, then an execution engine needs to represent the time with at least 10,000 bits.

B. Counterexample-guided abstraction refinement

The property that increasing the duration of tasks on the critical path increases the total latency can be used to detect spurious critical paths. Suppose that Algorithm 2 adds two predecessors to a task t, say t_1 and t_2. It is possible that this is due to timing coincidence. In such a case increasing the execution time of t_1 a bit either results in a precedence $t_1 \mapsto t$ or into a precedence $t_2 \mapsto t$. In the first case, we must also check whether $t_2 \mapsto t$ is a valid precedence. This “testing” of precedences (counterexample-guided abstraction refinement) through subtle modification of the task execution times can result in the proper set of precedences. This method presumes that the application can be modified and that new execution traces can be generated. Our implementation does not automate this approach.

C. Manual elimination

When looking at critical paths of actual systems it often is clear what the spurious parts are. Users can point out spurious precedences which can then be hidden from the critical path view. Knowledge of the application is essential. For instance, if two tasks do not share a common resource and they are not directly related through data dependencies then there will be no precedence relation. Manual elimination can also be the result of the counter example guided abstraction refinement suggested above.

V. Case study: Datapath performance

This section presents a case study of the datapath of a prototype high-end Océ color copier. The datapath is the digital image processing heart of the machine, and it supports three main use cases:

- *SCAN:* The scanner hardware scans images from paper. The digital images are processed and sent over the network to their destination (e.g., an email account).
- *COPY:* The scanner hardware scans images from paper, which then are processed and printed by the printer hardware.
- *PRINT:* A PS or PDF file is received from the network, processed and then printed by the printer hardware.

A *job* consists of one of these use cases and a number of additional settings. These include, for instance, paper size (A3, A4, letter, etc.), number of images per paper sheet (imposition) for copy and print jobs, and export format (e.g., PDF or JPEG) for scan jobs. Note that a single A4 page at 1200 dpi, which is a typical resolution used in printing today, gives a bitmap of approximately 400 MB in the RGB color model, and 531 MB in the CMYK color model. In order to reduce the amount of data that needs to be processed, there are various places in the datapath where compression and decompression occurs. Not every page is the same, however, and therefore the compression ratios differ per page; also within a job. Furthermore, the image data is split into smaller chunks at several places in order to increase pipelining and thereby the throughput of the datapath.

Fig. 5 shows a high-level overview of the datapath of the prototype. The datapath application is split into four parts: the scan, export, raster image processing (RIP) and print paths, which are linked for scan, copy or print functionality. The datapath is mapped to a general purpose PC platform. The colored and numbered stars indicate which components are used by the four sub-paths. Note that various processing steps can compete for access to the CPU or disk at the same time, which effectively slows them all down. The stars also suggest that the main memory could be a source of interaction. This is not the case, however. Many tasks use buffers in main memory, but these are allocated statically.

The performance requirements on the datapath focus mostly on the most important uses cases, e.g., printing a job of A4 pages with typical content. In these cases, the datapath *should not be the limiting factor*. Instead, the scanner or printer hardware should be limiting.

The datapath has been modeled with the Octopus toolset [6]. Modeling the large variety of input possibilities resulted in a complex model, because in many cases the inputs are handled in slightly different ways. For instance, both the imposition functionality (same-up, 2-up, 4-up, 6-up, 8-up, 9-up and 16-up) and paper size (A3, A4, letter, ...) impact the sizes and number of the data chunks that are processed by various steps. Because of the effects mentioned in the introduction, only execution traces but no task graphs are available from the model execution engine. Using our techniques, however, we were still able to apply critical path analysis to traces of typical jobs (thousands of pages), which consist of hundreds of thousands of tasks. This has proven to be helpful for, among others, a buffer minimization problem described below.

Memory is a scarce resource in the datapath and therefore minimization of buffer space that is used for inter-task communication is required while retaining maximal utilization of the scan and print hardware. Initial experiments showed that the smallest possible buffers in the scan path are sufficient for the most important scan jobs. A copy job (that also uses the scan path), however, showed reduced throughput. A part of the execution trace is shown in Fig. 6. Note that both the print and

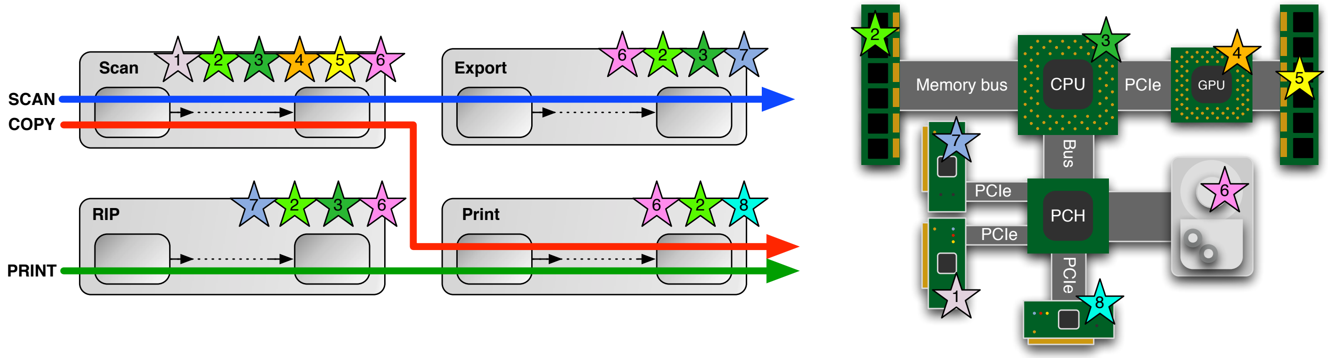


Fig. 5. Schematic representation of a data path and its mapping to a general purpose PC platform.

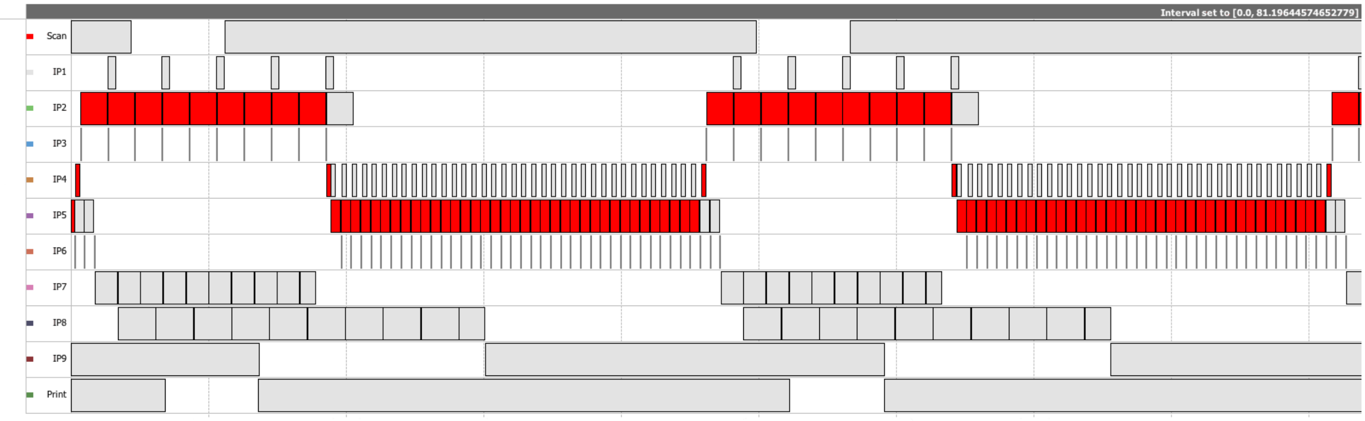


Fig. 6. A part of a trace of a COPY job which shows under-utilization of the scanner and printer hardware.

scan tasks are not consecutive. This signifies under-utilization of the hardware which should not happen. The critical tasks have been marked red using our techniques. There are two potential ways to solve the under-utilization of the scanner and printer: (i) make the critical tasks faster, or (ii) change system structure or parameters. We focussed on the latter. In the application model, IP2 and IP3 are data dependent and communicate through a shared buffer. The critical path shows that the 10th critical instances of IP2 (in the middle) must wait for IP3 to finish (the coloring of IP3 is not visible due to its small execution time). This is caused by a full buffer. Increasing the buffer size indeed solves the problem: IP2 instances are removed from the critical path and the gaps between the sets of consecutive IP2 instances become smaller because IP2 instances have more buffer space available to store their results. The result is that the gaps between the scan and print tasks disappear. (In fact also the buffer between IP3 and IP4 could be enlarged to achieve the same effect.) Critical path analysis allowed us to quickly zoom into the cause of the performance problem.

The above example of buffer minimization shows that combination of a critical path with the application structure can be used to discover critical resources. This is closely related to the work in [20], which applies bottleneck analysis to guide design space exploration in the context of resource-aware synchronous dataflow graphs.

VI. Conclusions

Critical path analysis is a useful tool for pinpointing performance bottlenecks with respect to timing related properties. The prerequisites are a task graph which models precedence constraints between tasks and an execution time for each task. Often, however, such a task graph is not available because the execution engine implements semantics with respect to resource constraints (limited memory prevents a task from running) and dynamic behavior (e.g., a task's execution time depends on resource load). An execution engine thus implicitly creates a task graph but outputs the fastest way to execute that task graph in the form of an execution trace. We construct an over-approximation of the critical paths in the unknown task graph from such an execution trace. This algorithm has been further generalized to deal with noisy execution traces from real machines. The algorithms have been added to the Octopus toolset [6] and scale to hundreds of thousands of tasks.

References

[1] M. Hendriks and F. Vaandrager, "Reconstructing critical paths from execution traces," in *Proceedings 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC'12),* 5-7 December 2012, Paphos, Cyprus. IEEE Computer Society, 2012, pp. 524–531. [Online]. Available: http://www.mbsd.cs.ru.nl/publications/papers/fvaan/HV12/

[2] J. E. Kelley and M. R. Walker, "Critical-path planning and scheduling," in *Eastern joint IRE-AIEE-ACM computer conference*. ACM, 1959.

[3] K. G. Lockyer, *Introduction to Critical Path Analysis*. Pitman Publishing Co., 1964.

[4] M. Pinedo, *Scheduling: Theory, Algorithms and Systems*, 2nd ed. Prentice Hall, 2002.

[5] W. Clark and H. L. Gantt, *The Gantt chart, a working tool of management*. Ronald Press, 1922.

[6] T. Basten et al., "Model-driven design-space exploration for embedded systems: The Octopus toolset," in *ISoLA 2010*, ser. LNCS, vol. 6415. Springer, 2010.

[7] J. D. Wiest, "Some properties of schedules for large projects with limited resources," *Operations Research*, vol. 12, 1964.

[8] A. Kastor and K. Sirakoulis, "The effectiveness of resource levelling tools for resource constraint project scheduling problem," *Int. Journal of Project Management*, vol. 27, 2009.

[9] E. Goldratt, *Critical Chain*. North River Press, 1997.

[10] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, 1990.

[11] P. Bjorn-Jorgensen and J. Madsen, "Critical path driven cosynthesis for heterogeneous target architectures," in *5th Int. Workshop on Hardware/Software Co-Design*. IEEE CS, 1997.

[12] J. Luo and N. K. Jha, "Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems," in *ASP-DAC*. IEEE CS, 2002.

[13] C.-Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *8th Int. Conf. on Distributed Computing Systems*. IEEE, 1988.

[14] J. K. Hollingsworth, "An online computation of critical path profiling," in *1st ACM SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, 1996.

[15] P. Barford and M. Crovella, "Critical path analysis of TCP transactions," *SIGCOMM Computer Communication Review*, vol. 30, 2000.

[16] M. Schulz, "Extracting critical path graphs from MPI applications," in *Int. Conference on Cluster Computing*. IEEE, 2005.

[17] W. M. P van der Aalst et al., "Workflow mining: A survey of issues and approaches," *Data & Knowledge Engineering*, vol. 47, 2003.

[18] W. M. P. v. d. Aalst and B. F. v. Dongen, "Discovering workflow performance models from timed logs," in *1st Int. Conf. on Engineering and Deployment of Cooperative Information Systems*. Springer, 2002.

[19] F. A. Lootsma, *Fuzzy logic for planning and decision making*. Springer, 1997.

[20] Y. Yang et al., "Automated bottleneck-driven design-space exploration of media processing systems," in *Design, Automation and Test in Europe*. IEEE, 2010.