Noname manuscript No.
(will be inserted by the editor)

Analyzing Execution Traces – Critical-Path Analysis and Distance Analysis

Martijn Hendriks · Jacques Verriet · Twan Basten · Bart Theelen · Marco Brassé · Lou Somers

the date of receipt and acceptance should be inserted later

Abstract System designers make trade-offs between metrics of interest such as execution time, functional quality and cost to create a properly balanced system. Execution traces, which are sequences of timestamped start and end events of system tasks, are a general and powerful means to understand the system behavior that gives rise to these trade-offs. Such traces can be produced by, e.g., executable models or prototype systems. Their interpretation, however, often is non-trivial. We present two automated analysis techniques that work on execution traces to help the system designer with interpretation. First, critical-path analysis can be used to answer the typical "what is the bottleneck" question, and we extend earlier work of [16] with a technique that uses application information to refine the analysis. Second, we define a pseudo-metric on execution traces, which is useful for calibration and validation purposes, and which can be used to visualize the differences between traces. Both techniques are based on a common graph representation of execution traces. We have implemented our techniques in the Trace visualization tool [12], and have applied them in a case study from the digital printing domain.

Keywords model-based design · execution trace · critical path · metric · visualization

Martijn Hendriks · Jacques Verriet · Twan Basten · Bart Theelen
Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

Twan Basten · Lou Somers
Eindhoven University of Technology, Eindhoven, The Netherlands

Marco Brassé · Lou Somers
Océ Technologies B.V., Venlo, The Netherlands

1 Introduction

1.1 A workflow around execution traces

Software-intensive embedded systems are often subject to the well-known trade-off between *good*, *cheap*, and *fast*: you can pick two, but having all three is impossible. A *good* and *cheap* system will not be *fast*. System design thus involves balancing system-level qualities to achieve an optimal mix. This, however, is a very challenging task because it often is unclear how design choices impact dynamic system-level qualities such as throughput or energy consumption. The predominant way of working nowadays is to build executable models and prototypes of the system to experiment with different designs and to make choices based on the results of the experiments. The model-based or prototype-based experiments during the design process typically measure the system-level qualities of interest, e.g., average throughput for a given set of inputs and varying design parameters (e.g., memory size). These kinds of experiments are done to obtain hard numbers, but often also to validate trends in the design space. An example of such a trend is "more memory will increase the throughput". This is usually, but not always, the case.

There, however, are a number of situations in which the designers want to look into the detailed behavior of the model or prototype, e.g., when the system-level behavior shows unexpected results, for model calibration and validation purposes, or to diagnose and solve performance problems. The detailed behavior can be studied through *execution traces*, which are sequences of timestamped start and end events of system tasks. Execution traces can be generated from models, by a wide variety of modeling and analysis tools, and from prototypes, through instrumentation of software. There

are many visualization tools available for this kind of data, e.g., the TRACE tool [12] which we use in this work because it is a configurable tool which we can easily extend.

The interpretation of system behavior through visualized execution traces is a powerful method for gaining insight and for analyzing performance-related problems. Interpretation is non-trivial, however, because of at least two reasons. First, execution traces typically are very large; tens of thousands of task executions is not exceptional. Looking into all details of the trace would be very time-consuming. Second, interpretation of local behavior can be complicated due to the sheer complexity of the system dynamics, and it often is hard to see the global consequences of local behavior. To aid the interpretation, we present two analysis techniques that work on execution traces and that can be applied to a wide range of systems and in combination with a wide variety of modeling and analysis tools. First, critical-path analysis can be used to answer the omnipresent "what is the bottleneck?" question. Second, trace distance analysis can be used for calibration and validation purposes, and to visualize the difference between execution traces. This is useful, e.g., after a change in the design, such as a different scheduling policy or a larger memory, etc. Both the critical-path analysis and the trace distance are based on a common graph representation of execution traces.

1.2 Motivating Example

The following example (taken from [7]) shows a synthetic example system that manifests many of the more complex issues that are present in actual software-intensive embedded systems. We use this example to motivate and illustrate our techniques.

Example 1 Figure 1 shows an example system consisting of seven (software) tasks, A – G, that are mapped to a hardware platform consisting of a CPU, a GPU, an FPGA and two memory modules. The system processes a stream of input objects of *low*, *normal* or *high* complexity. Task F is only executed for high-complexity objects. The arrows between the task nodes indicate data dependencies. The execution time for each task is shown in the task node. The arrows from the task nodes to the resources indicate the mapping. For the memory modules, these arrows are annotated with the memory requirements of the task for processing. The arrows (annotated with variables B1 – B5) from the data dependency arrows to the memories indicate the amount of memory necessary for the data communication between processing resources. These variables all

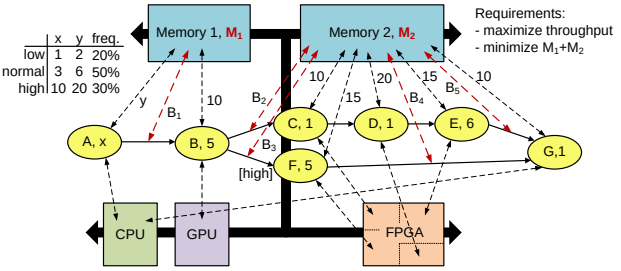
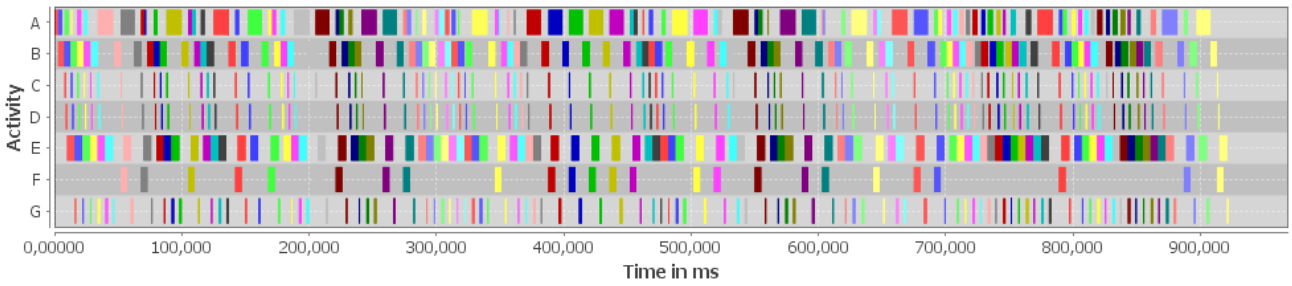


Fig. 1: A small example system (taken from [7]).

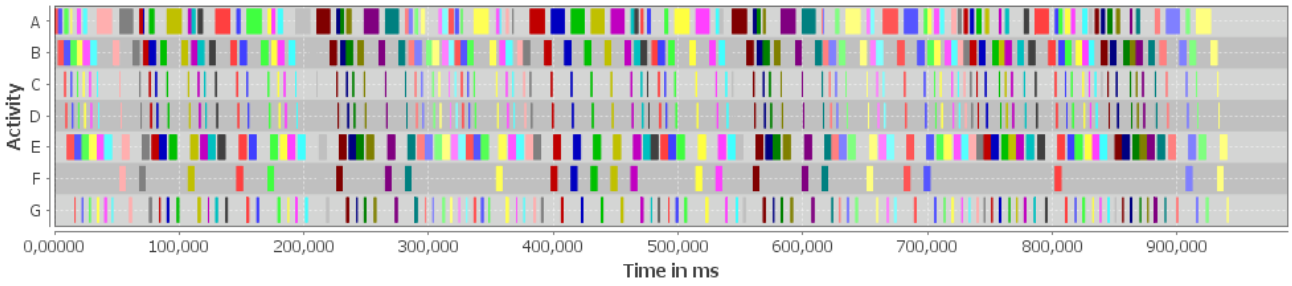
have value 10 in this example. E.g., task B takes 5 time units on the GPU and needs 10 units of M1 for internal processing, and 10 units of M2 for storing the result that is passed to C (indicated by arc B2). If a complex object is processed, then it also needs 10 units of M2 for the result that is passed to F (indicated by arc B3). The resource requirements for task A depend on the object type, and are listed in the table. The occurence frequencies for the different types in the input object stream are also given in the table. The analysis question for this system is to clarify the trade-off between memory sizes and the expected throughput. It is very hard to get insight in this issue by static analysis only. Because the number of possible inputs is very large, it is not feasible to compute static schedules beforehand. Instead, local rules govern the execution, such as the memory allocation strategy in the memories (fragmentation can occur), and the CPU scheduling in which G preempts A. The total system behavior therefore is *emergent*. To gain insight in the dynamic behavior and the trade-off between memory size and throughput, we have modeled this system in the OCTOSIM environment [15] and have run a number of simulations with different sizes of the memory units.[1] Figure 2 shows two execution traces in the TRACE tool for processing 100 objects. The system in Fig. 2a has 80 units of M2, and the system in Fig. 2b is the same (and processes the same input) except that it has 90 units of M2. For this particular input, having more memory decreases the performance, which is rather unexpected.

The non-monotonous behavior of the execution time with respect to the size of the M2 memory in Ex. 1 above is a good demonstration of the complex behavior that can arise for even simple systems, and which makes it hard to estimate the impact of design choices such as the memory sizes. Analysis of this behavior can be aided by visualization of the execution traces, such as in Figs. 2a and 2b. These example traces, however,

[1] We do not precisely specify the execution semantics of the system as this is out of scope. What matters are the execution traces that are produced by the system.



(a) 80 units of M2: execution time of 922 time units.



(b) 90 units of M2: execution time of 941 time units.

Fig. 2: Execution traces of the example system with varying amounts of RAM.

make it clear that we need automated help for interpretation and diagnostics. For instance, manually finding the differences to diagnose the unexpected performance decrease is tedious. For larger traces that are very similar, a manual search is not feasible.

1.3 Contribution

Our contribution consists of the following parts. First, we propose the already mentioned graph-based representation of execution traces, which is a generalization of the one proposed in [16]. This representation is based on the temporal ordering of the tasks and serves as a basis for both the critical-path analysis and the trace distance computation. Our method of representation is computationally efficient: it takes time $\mathcal{O}(n \cdot w)$ to construct a graph from an execution trace, where n is the number of events in the trace and w quantifies the parallelism in the system that has produced the trace.[2] Second, we extend the critical-path analysis techniques from [16] with a feature that uses application precedence constraints (which usually are known) to find resources in the platform that cause blocking. Third, we define a metric for traces based on their graph representations. The metric lies between approaches based on string edit distances that totally discount time [24,28,35] and approaches that try to precisely capture timing differences [27,23]. The running time of our distance algorithm is not sensitive to the actual distance between two traces and in that respect differs from existing, efficient approaches based on string edit distances [28,35]. Instead, the running time depends on the size of the trace, and the amount of parallelism (clarified in Sec. 5). This suits our domain, as the amount of parallelism in a trace is typically limited by the number of resources in the system, and can often be considered constant. This distance function is consequently used to visualize the differences between traces. Fourth, we show how our techniques, which have been implemented in the Octopus toolset [7] and the Trace tool [12], can be applied in an industrial case study.

[2] Sometimes w can be derived by static analysis of the model. For instance, it could be bound by the cardinality of a maximal antichain in the application task graph or by the number of resources in the platform. In general, however, w can only be determined by examination of the execution trace.

1.4 Related work

Critical-path analysis. Critical-path analysis is a technique from the project planning domain that originally was used to minimize the duration of projects by analysis of the activity dependency graph of the project [22,25,33]. Subsequent work describes methods that also take resources into account, such as resource leveling [21] and the critical chain method [14]. It is, however, an

NP-hard problem to minimize the makespan of projects with resource constraints [31]. In our approach, critical-path analysis is applied after the system has executed for the purpose of analysis and improvement, *using only an execution trace as input*. This thus is different from the way in which critical-path analysis is used in the project scheduling domain. Related work from the embedded and distributed systems domains mostly assumes (partial) knowledge of the task graph, e.g., [34, 8,36,18,6,32,9]. This is an assumption that we do not make. An exception is the work in [26], in which a task graph of a schedule is created in a way that is similar to ours. The main difference is that we present a soundness proof of the approach, which establishes a relation between the system that produced the execution trace and the critical paths that we extract from the trace. Our techniques to build models from execution traces is also related to process or workflow mining (see, e.g., [2] for a survey). For instance, in [1] a technique is presented that includes timing information for performance analysis. The nature of the constructed models is different, however, and their techniques usually work with a set of execution traces. Our technique, in constrast, works on a single trace. We extend our earlier work ([16]) with a technique to determine *resource bottlenecks*. This is strongly related to the bottleneck analysis in Synchronous Dataflow as presented in [37].

Trace distance analysis. The quantitative similarity of timed systems has been studied in [17]. The authors provide algorithms to compute the distance between systems that have been modeled by timed automata. The work in [19] takes a similar approach using a different modeling formalism. Both approaches define a metric on timed event traces that requires that the events in both sequences are equal (i.e., only the timing of the events may differ) in order to get a finite distance. This is a too strong requirement for our purpose. In the case study (see Sec. 7) for instance, we have traces from a prototype that we need to compare with traces from a model for calibration and validation purposes. The event sets of the prototype traces do not match the event sets of the model traces because of a modeling mismatch due to abstractions that have been applied in the modeling process.

Traces can be interpreted as strings (when the timestamps are removed), on which edit distances can be applied. For instance, the Levenshtein distance [24] computes the number of insertions, deletions and substitutions that are needed to transform one string into the other. A dynamic programming approach takes time complexity $\mathcal{O}(n \cdot m)$, where n and m are the lengths of the respective strings. Improvements of the algorithm are described in, e.g., [28,35]. These algorithms have running times of $\mathcal{O}(n \cdot d)$ and $\mathcal{O}(n \cdot p)$ respectively, where d is the distance between the strings, and p is a parameter related to the distance and the string lengths. A string-based metric should take care of events with the same timestamp by, e.g., ordering them according to some total order before applying the distance computation. Otherwise, such a metric assigns a positive distance to traces that we would consider equal. Such an additional ordering step introduces another aspect with respect to computational complexity. The main differences between these string-based approaches and our work are the following. First, the complexity of our algorithm does not depend on the distance of the event traces. Second, we consider ordering on task level instead of event level, which makes the distance less fine-grained. For instance, let $(a_1, 0)(b_1, 1)(a_2, 4)(b_2, 5)$ and $(a_1, 0)(b_1, 1)(b_2, 5)(a_2, 7)$ be timestamped event sequences which model the start and end of tasks a and b. A string-based approach would map these events to $a_1b_1a_2b_2$, and $a_1b_1b_2a_2$ respectively. A string-based metric that works at the event level would thus give a non-zero distance, but our task-level pseudo-metric indicates that the distance between the event traces equals 0 because the tasks overlap in both cases.

A distance for event traces is introduced by Mannila and Ronkainen [27]. Their work is based on a weighted edit distance using weights that are based on event occurrences and the timestamps of the events, thus taking care of events that happen at the same time. The time complexity of the proposed algorithm, however, is quadratic in the length of the input sequences, which renders it impractical for large traces. A similarity measure on timestamped business events is presented by Obweger et al. [29]. The approach is configurable (driven by the interest of the analyst) but has an exponential execution time. A distance metric for execution traces specific for multimedia applications is presented by Kengne et al. [23]. This domain-specific metric is used to diagnose domain-specific problems such as audio and video that are not properly synchronized. The authors define three distance measures based on typical multimedia anomalies. These metrics are not generally applicable. Two of them are computable in linear time. The third is based on [27] and has a quadratic execution time.

Graph edit distance provides a robust and flexible way to solve approximate graph matching problems. It is widely used in image retrieval and pattern recognition applications. It is also the basis of our work. An overview of graph edit distance work is presented by Gao et al. [13]. The idea is to transform a (labeled) graph $G_1 = (V_1, E_1, \ell_1)$ into a graph $G_2 = (V_2, E_2, \ell_2)$ by a sequence of operations that include insertion, dele-

tion and substitution. Substitution changes the label of a graph. Finding the minimal sequence of operations to transform G_1 into G_2 is NP-hard [38]. In our context, however, using substitution is not appropriate: task A in one trace cannot be used to cover for a task B in another trace. We therefore define an adapted graph edit distance that uses only insertion and deletion. We use this distance to define a task-level pseudo-distance on execution traces, and to visualize the differences using the Trace tool.

1.5 Outline

Section 2 recalls some mathematical prerequisites. Then, Sec. 3 explains how we create graph representations of execution traces, and shows an efficient algorithm. Section 4 extends our previous work [16] on critical-path analysis with some new features that aid the interpretation. The distance on executions is defined in Sec. 5. Section 6 introduces the Trace tool and explains its core concepts, and how the critical-path analysis and trace distance functionalities are implemented. Next, Sec. 7 shows several applications that are based on traces from industrial case studies. Finally, we present our conclusions in Sec. 8. Proofs for the propositions and theorems that appear below can be found in Online Resource 1.

2 Preliminaries

We let $\mathbb{R}$ denote the set of real numbers, and $\mathbb{R}^+$ the set of positive real numbers including zero. The function *max* selects the maximum from a finite set of positive real numbers, and $max(\emptyset) = 0$.

Below follow some preliminaries on graphs which closely follow the definitions of [3]. A *directed graph* G is a tuple (V, E), where V is a set of vertices, and $E \subseteq V \times V$ is a set of arcs. We say that $v \to v'$ if and only if $(v, v') \in E$. A sequence $v_1, v_2, \ldots, v_n$ is a directed path of G if and only if $v_i \to v_{i+1}$ for all $1 \leq i < n$. There is a *directed path* from v to v', denoted by $v \to^+ v'$, if and only if there exists a directed path $v_1, v_2, \ldots, v_n$ such that $v = v_1$ and $v' = v_n$. If $v \to^+ v'$, then we call v' a *successor* of v and v a *predecessor* of v'. Moreover, if $v \to v'$, then v' is called an *immediate successor* of v and v an *immediate predecessor* of v'.

A graph is *cyclic* if and only if it has a directed path that starts and ends in the same vertex and passes through at least one other vertex, and *acyclic* otherwise.

A graph is *transitive* if and only if $v \to^+ v'$ implies that $v \to v'$. The *transitive closure* of $G = (V, E)$, denoted by G^+, is the graph (V, E'), such that E' is the smallest subset of $V \times V$ that contains E and is transitive. The *transitive reduction* of $G = (V, E)$, denoted by G^-, is the graph (V, E') such that $(V, E')^+ = G^+$ and for every $E'' \subset E'$ holds that $(V, E'')^+ \neq G^+$ [3]. Intuitively, the transitive closure has an arc from v to v' if there is a directed path from v to v' in the original graph. In the transitive reduction, however, all such redundant arcs are removed. An arc (v, v') is *redundant* if there is a path from v to v' via another vertex v''. A graph without redundant arcs therefore is transitively reduced. Consider, for instance, the graph $G = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$. Then $G^+ = (V, E \cup \{(v_1, v_3), (v_1, v_4)\})$ and $G^- = (V, E \setminus \{(v_2, v_4)\})$.

A vertex v of a graph (V, E) is a *source* if it has no incoming arcs, and a *sink* if it has no outgoing arcs.

Let $G = (V, E)$ be a directed acyclic graph. The *width* of G, denoted by $w(G)$, is defined as the cardinality of a maximum anti-chain in G. An *anti-chain* of G is a set of vertices $U \subseteq V$ such that for all vertices v, v' in U holds that $v \not\to^+ v'$.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. We say that G_1 is a subgraph of G_2, denoted by $G_1 \subseteq G_2$, if and only if $V_1 \subseteq V2$ and $E_1 \subseteq E_2$.

A topological ordering of a directed graph (V, E) is a total order $\leq$ on V such that $(v, v') \in E$ implies that $v \leq v'$.

A *metric*, or *distance*, on a set M is a function $d : M \times M \to \mathbb{R}^+$ such that for all $x, y, z \in M$ (1) $d(x, y) \geq 0$, (2) $d(x, y) = 0$ if and only if $x = y$, (3) $d(x, y) = d(y, x)$, and (4) $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality). If the second property is replaced by $d(x, x) = 0$, then d is called a *pseudo-metric*. In that case the distance between two different elements of M is allowed to be 0 (see, e.g., [4]).

3 Reconstructing task graphs

3.1 Application graphs, task graphs and executions

The Y-chart is a well-known method to decompose systems into an *application*, a *platform* and a *mapping* of the application to the platform [5]. Figure 1 shows an example of a typical Y-chart based decomposition. It shows seven tasks with data dependencies. When executed on n objects, this leads to n executions of each of the tasks excluding task F; the number of executions of F depends on the input stream objects. The application part thus often boils down to the definition of a set of task executions and precedence constraints between these task executions. This gives us the following definition:

Definition 1 (Application graph) A tuple $(T, \mapsto)$ with a set of task executions T, $\mapsto \subseteq T \times T$, and $\mapsto$ acyclic is called an *application graph*.

The arcs between task executions express the precedence constraints (also called *dependencies*), and task executions not related by precedence constraints can be executed in parallel. We introduce application graphs as an auxiliary notion. In practice, applications are specified in a wide variety of graph-based formalisms, which implicitly define an application graph as introduced above. When an application is mapped to a certain platform and is executed, the effect is that start and end timestamps are assigned to each task execution in the application graph. These timestamps are such that all operational rules of the platform and all dependencies of the application are respected. This brings us to the following definition:

Definition 2 (Task graph) A tuple $(T, \rightarrow, d)$ with a set of task executions T, $\rightarrow \subseteq T \times T$ and $\rightarrow$ acyclic, and $d : T \rightarrow \mathbb{R}^+ \setminus \{0\}$, is called a *task graph*.

A basic assumption throughout this paper is that tasks have a strictly positive execution time. If $\pi = v_1, v_2, \ldots, v_n$ is a path of a task graph $(T, \rightarrow, d)$, then the *duration* of this path, denoted by $duration(\pi)$, equals $d(v_1) + d(v_2) + \cdots + d(v_n)$.

We take the view that the execution platform (being a model or a physical prototype), when given an application graph to execute, can "add" task executions. An example of this is tasks for control overhead that are not explicitly part of the application model. Furthermore, the execution platform in general also adds dependencies between task executions, e.g., to resolve resource conflicts. More formally, consider an application graph $A = (T_1, \mapsto)$ and a task graph $B = (T_2, \rightarrow, d)$. We say that B is a task graph of A if and only if $T_1 \subseteq T_2$ and $u \mapsto v$ implies that $u \rightarrow^+ v$.

Task graphs characterize the execution of an application on a certain platform. Unfortunately we often do not have access to the actual task graph object because it is internal to and often on-the-fly created by the execution platform. (Note that we usually do have access to the application graph because that is a design-time artifact.) Instead, what we can observe is the *execution* of the task graph, which boils down to the start and end timestamps of the task executions. An execution can capture parallelism of task executions (i.e., task executions that are active simultaneously), but does not show the dependencies between task executions, which are present explicitly in the task graph. We assume that the execution engine executes the task graph that it constructs internally in the most efficient way.[3] This is done by starting a task execution at the moment that all its predecessors have finished [31], and is reflected in the definition below.

Definition 3 (Execution) Let $G = (T, \rightarrow, d)$ be a task graph. An *execution* of G, denoted by $exec(G)$, is a tuple $(start, end)$ where $start, end : T \rightarrow \mathbb{R}$ such that

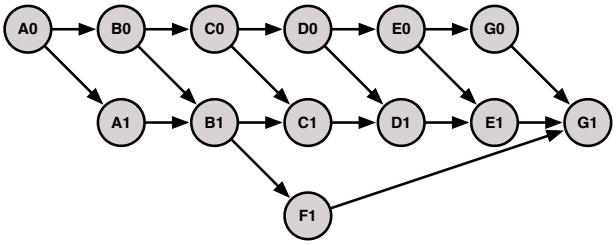
- $start(t) = max(\{end(t') \mid t' \rightarrow t\})$, and
- $end(t) = start(t) + d(t)$.

Note that having both d and end contain redundancy, which, however, provides notational convenience.

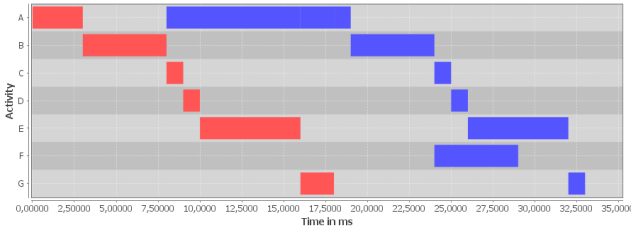
Example 2 Consider the example system in Fig. 1 and let us assume that it needs to process two objects: one of normal complexity and one of a high complexity. Figure 3a shows the application graph. We consider two platforms with different memory sizes, and the executions of the application on these different platforms are shown in Figs. 3b and 3c. Note that in Fig. 3b A1 has to wait for B0 because there is not enough M1 memory for them to run in parallel. This constraint is not present in the application graph, but it is present in the task graph that underlies the execution in Fig. 3b. Both A and G are mapped to the CPU and have the same priority in our example. Therefore, when G runs together with A, both are slowed down. This explains the fact that execution of G0 in Fig. 3b takes twice as long as in Fig. 3c.

The analysis methods that we present in the remainder of this paper are based on executions rather than on task graphs. The reason is that executions are relatively easy to obtain from simulations or physical prototypes. This makes our techniques broadly applicable and allows us to compare execution traces generated by different artifacts (such as simulators and prototypes). We have introduced the concept of task graphs, because they are the connection between application graphs (design-time artifacts) and executions (run-time artifacts). As such, we also establish formal relations between the results of our techniques and the underlying, but unavailable, task graphs. At the core of our techniques is a graph representation derived from executions, which is the subject of the following section.

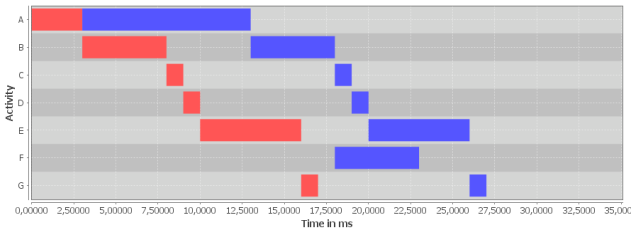
[3] This assumption is not limiting because the task graph can have an arbitrary set of additional task executions that can be inserted between the task executions of the application graph. It therefore can delay application task executions arbitrarily.



(a) Application graph for processing two objects.



(b) Execution for a platform with M1 = 50, M2 = 60.



(c) Execution for a platform with M1 = 150, M2 = 160.

Fig. 3: An application graph of the example system for processing two objects, and two executions on different platforms (the color indicates the object).

3.2 Approximating task graphs from executions

In order to reconstruct the original dependencies between tasks[4] from the execution only, we make use of the key assumption that the tasks are executed by the system *as soon as they are enabled* modulo some overhead ϵ. For instance, if t provides data for t' and this is the only dependency, then the time distance between the end of t and the start of t' in the execution is less than or equal to ϵ time units. This assumption allows us to define dependencies between tasks from the execution in the form of the relation $close_\epsilon : T \times T$ which is parameterized by an $\epsilon \in \mathbb{R}^+ \cup \{\infty\}$:

$$close_\epsilon(t, t') \triangleq end(t) \leq start(t') \wedge start(t') - end(t) \leq \epsilon$$

Thus, $(T, close_\epsilon)$ is a graph representation of an execution. The reconstructed graph is acyclic, because the $close_\epsilon$ relation only relates a task t with a task t' if t ends before t' starts, or if t ends at the moment that t' starts. The second case can cause no cycles because we assumed that tasks have a strictly positive execution time. Note that the $close_\epsilon$ relation can induce redundant arcs if $\epsilon > 0$. In order to have a compact representation, which is beneficial for the efficiency of our algorithms, we define the ϵ-approximated task graph of an execution as the transitive reduction of the graph consisting of the tasks which are related through the *close* relation defined above.

Definition 4 (ϵ-Approximated task graph) Let f be an execution of a task graph $(T, \rightarrow, d)$, and let $\epsilon \in \mathbb{R}^+ \cup \{\infty\}$. The *$\epsilon$-approximated task graph* of f, denoted by $\mathcal{G}_\epsilon(f)$, is defined as $((T, close_\epsilon)^-, d)$.

An execution has a unique approximated task graph, because the transitive reduction of a directed acyclic graph is unique [3]. Unfortunately, the approximation is generally neither sound nor complete when compared to the real task graph:

- *False positives:* The fact that $(t, t') \in close_\epsilon$ can be mere coincidence: a timing artifact.
- *False negatives:* A real dependency (t, t') can be missed if ϵ is set too small.

Note that false negatives can have the effect that a task graph consists of disconnected components.

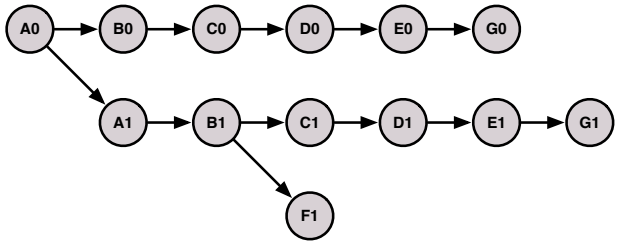
The ϵ parameter can be used to balance the ratio of false positives and false negatives. Intuitively, $\mathcal{G}_0(f)$ has the least false positives because the timing window for relating tasks is 0. On the other hand, $\mathcal{G}_\infty(f)$ has the most false positives because it relates a task t with any other task t' that starts after the former has finished. The situation for false negatives is mirrored: $\mathcal{G}_0(f)$ has the most and $\mathcal{G}_\infty(f)$ has the least. This intuition is formalized by the following theorem:

Theorem 1 *Let f be an execution, and let $\epsilon_1, \epsilon_2 \in \mathbb{R}^+ \cup \{\infty\}$ such that $\epsilon_1 \leq \epsilon_2$. Then $\mathcal{G}_{\epsilon_1}(f) \subseteq \mathcal{G}_{\epsilon_2}(f)$.*

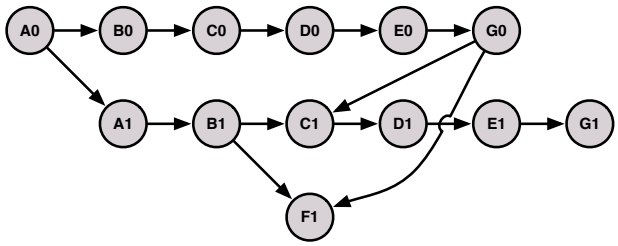
Proof See Online Resource 1.

Example 3 Figures 4a–4c show the task graph representations for $\epsilon = 0$, $\epsilon = 1$ and $\epsilon = \infty$ of the execution in Fig. 3c. In the cases $\epsilon = 0$ and $\epsilon = 1$, the application arc from F1 to G1 is not present, because these tasks are separated by a time gap larger than 1. This is an example of a *false negative*, because it is present in the application graph (and therefore the task graph), shown in Fig. 3a. The root cause for this false negative is the dependency of G1 on E1, which delays the execution of G1. In case $\epsilon = 1$, additional arcs G0 to C1 and G0 to F1 are present, because the gaps equal 1. These are not data dependencies (because they are

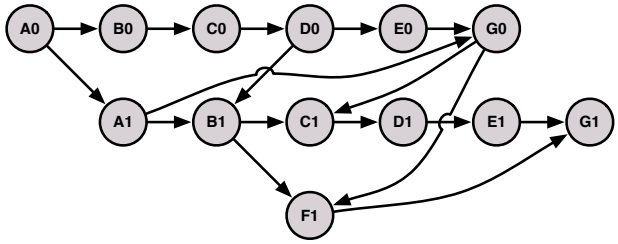
[4] From now on we use *task* instead of *task execution* for brevity.



(a) $\mathcal{G}_0(f)$



(b) $\mathcal{G}_1(f)$



(c) $\mathcal{G}_\infty(f)$

Fig. 4: Various task graphs for an execution f of Fig. 3c.

not in Fig. 3a), but they could be caused by the M2 resource which is used by all tasks. If this is not the case, then these arcs are *false positives*. Establishing whether these additional dependencies are true dependencies or timing coincidences leading to false positives is hard, because in general the time-dependent state of the execution platform needs to be taken into account.

The following theorem states that false negatives are absent in the ∞-approximation of the task graph, i.e., it contains all precedences of the real task graph (and probably many more).

Theorem 2 *Let $G = (T, \rightarrow, d)$ be a task graph, let f be an execution of G, and let $\mathcal{G}_\infty(f) = (T, \rightarrow_\infty, d)$. If $t \rightarrow^+ t'$, then $t \rightarrow^+_\infty t'$.*

Proof See Online Resource 1.

We use approximated task graphs for our analysis even though we know that these can contain false positives and false negatives. We discuss the implications in Sec. 4 and Sec. 5 that define the analysis techniques. Below, we present an efficient algorithm to compute the approximated task graph from a representation of an execution.

3.3 Efficient task graph construction

Executions of systems or system models are typically represented by sequences of timestamped start and end events of tasks. This is the reason why we define our graph construction algorithm in terms of such sequences.

Definition 5 (Execution trace) Consider an execution $f = (start, end)$. An *execution trace* (or *trace* for short) of f is a sequence of all events of f, defined as $\{(v, start(v), \mathbf{s}) \mid v \in T\} \cup \{(v, end(v), \mathbf{e}) \mid v \in T\}$ that is ordered according to the timestamp of the event, and – in case of events with an equal timestamp – according to the event type: **e** events before **s** events.

Note that an execution can potentially be represented by many traces that differ in the order of events that have an equal timestamp.

The assumption on the time ordering is naturally accommodated by systems that generate traces such as real systems or simulation models, because these typically timestamp and log events when they occur. The requirement that events with an equal timestamp are ordered by their type is only needed in order to simplify the presentation of our algorithm. In general, components that process the events of a trace can ensure this second requirement on-the-fly, e.g., by caching events in two separate sets (one for start events and one for end events). These are then flushed to the main algorithm in the proper order when an event with a later timestamp arrives, or if the end of the trace is reached. The proper order is: first those from the end set, then those from the start set. The order of events within the sets does not matter.

Above we have assumed that an execution is represented by a trace, and therefore, we have designed our algorithm that constructs the ϵ-approximated task graph to take such traces as input instead of executions. We have also noted that an execution may have multiple traces that represent it and that differ with respect to the ordering of events with an equal timestamp. The task graph construction abstracts from these differences in that it maps all traces of an execution to a single task graph.

Given a trace f that represents an execution, we can construct the ϵ-approximated task graph $\mathcal{G}_\epsilon(f) = (T, \rightarrow, d)$ of that execution using Alg. 1. The algorithm iterates over the events in the trace and builds the task

Algorithm 1 Computation of the ϵ-approximated task graph from a trace $(v_1, t_1, e_1) \cdots (v_n, t_n, e_n)$ that represents an execution f.

```
1: T_0 = ∅, →_0 = ∅, d_0 = ∅ // task graph under construction
2: O_0 = ∅, F_0 = ∅, D_0 = ∅ // helper sets
3: for i = 1 to n do
4:    // processing event (v_i, t_i, e_i)
5:    if e_i = s then
6:       T_i = T_{i-1} ∪ {v_i}
7:       →_i = →_{i-1} ∪ {(v_k, v_i) | v_k ∈ F_{i-1} ∧ t_i − t_k ≤ ε}
8:       d_i = d_{i-1} ∪ {v_i ↦ t_i} // temporary; finalized line 15
9:       O_i = O_{i-1} ∪ {v_i}
10:      F_i = F_{i-1}
11:      D_i = D_{i-1}
12:   else
13:      T_i = T_{i-1}
14:      →_i = →_{i-1}
15:      d_i = (d_{i-1} \ {v_i ↦ d_{i-1}(v_i)}) ∪ {v_i ↦ (t_i − d_{i-1}(v_i))}
16:      O_i = O_{i-1} \ {v_i}
17:      F_i = {v_i} ∪ (F_{i-1} \ {w ∈ F_{i-1} | w →_{i-1} v_i})
18:      D_i = D_{i-1} ∪ {w ∈ F_{i-1} | w →_{i-1} v_i}
19:   end if
20: end for
21: return (T_n, →_n, d_n)
```

graph on-the-fly. The algorithm uses a number of task sets. The set O_i contains so-called *open* tasks for which only the start event has been processed. The set F_i contains so-called *finished* tasks for which also the end event has been processed, and which are without successors in the arc relation $\rightarrow_i$. The set D_i contains so-called *done* tasks from which the end event has been processed and which are predecessors of tasks in F_i. If a start event has been processed, the corresponding task is added to T_i, and arcs from the current tasks without successor (F_i) to the new task are added (lines 6 and 7) if they are ϵ close. This ensures that the tasks are ordered with respect to their temporal properties. Furthermore, line 8 adds the start time of the task v_i to the mapping d_i. When the end event for a task v is processed, task v is moved from O_i to F_i, and all tasks in F_i that are predecessors of v are moved from F_i to D_i. This ensures that the minimum set of arcs is used to encode the temporal relation of the tasks, i.e., the transitive reduction. In line 15, the execution duration function is updated with the proper value using the stored value that has been set in line 8 as the start time of the task.

Proposition 1 *Algorithm 1 adds the tasks (in line 6) according to a topological order of the resulting arc relation.*

Proof See Online Resource 1.

The following theorem states that Alg. 1 computes the ϵ-approximated task graph of an execution that is represented by an execution trace.

Theorem 3 *Let f be an execution, and let τ be a trace of f. Algorithm 1 computes $\mathcal{G}_\epsilon(f)$ from τ.*

Proof See Online Resource 1.

3.4 Complexity of task graph construction

We discuss the complexity of the algorithm in terms of set operations: set addition, set removal and the set inclusion check. When the sets are implemented with hash tables that use a proper hash function, the average time complexity for these operations is constant [10].

The width of a task graph quantifies the amount of parallelism in the system that has produced the trace. It is equal to the maximum number of tasks that are executed at the same time. In our setting – system-level models of embedded systems – the parallelism is often limited and depends on the number of resources in the platform. We use this to define the complexity of the task graph construction.

Theorem 4 *Consider a trace τ of an execution f. Algorithm 1 uses $\mathcal{O}(length(\tau) \cdot w(\mathcal{G}_\epsilon(f)))$ set operations.*

Proof See Online Resource 1.

4 Critical-path analysis

4.1 Critical-path analysis for perfect executions

Critical-path analysis is a technique that computes the longest paths in a task graph. This gives an indication of the bottlenecks, which is useful information if the performance of the system needs to be improved. We have explained earlier that we generally do not assume access to real task graphs because these are constructed implicitly. Instead, we use the approximate task graph, derived from the execution (via Alg. 1), for our critical-path analysis. This section assumes that all tasks of the task graph are represented in the execution trace: we have *perfect* execution traces. In Sec. 4.2 we deal with the case in which not all tasks are represented in the execution trace.

Algorithm 2 shows the critical-path algorithm, which is a slightly rephrased version from [22]. It computes the minimum and maximum starting times of tasks, $start^-$ and $start^+$ respectively. The *float* of a task t is defined as $start^+(t) - start^-(t)$. The float of a task indicates how much the execution time of the task can increase without increasing the total execution time of the task graph. A task is *critical* if and only if it has zero float. The set of critical tasks is equivalent to the set of tasks that are on the longest paths in the task graph (see,

Algorithm 2 Critical-path analysis of a task graph $(T, \rightarrow, d)$.

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

e.g., [16]). Note that the minimum starting times of the tasks, given by the $start^-$ function, equals the *start* function of the execution of a task graph (see Def. 3).

The algorithm assumes that there is a unique sink task, and that we have a topological ordering of the tasks. Because a task graph is acyclic, such a unique sink task has the property that all other tasks lead to it. If it does not already exist, it can be added straightforwardly to a task graph in order to satisfy the condition by extending the arc relation with arcs from all sink tasks to the newly added artifical task. Furthermore, Prop. 1 ensures that Alg. 1 also gives a topological ordering of the tasks. Therefore, we can apply Alg. 2 directly to the output of Alg. 1. We need the following two lemmas to establish a formal relation between the computed critical path and the underlying task graph.

The following theorem states that running the critical-path analysis algorithm on the 0-approximated task graph that is obtained from an execution trace gives an *over-approximation* of the set of critical tasks of the underlying task graph.

Theorem 5 *A task that is marked critical (i.e, it has zero float) by Alg. 2 when run on task graph G, is also marked critical by Alg. 2 when run on $\mathcal{G}_0(exec(G))$.*

Proof See Online Resource 1.

The approach can produce spurious critical paths due to timing coincidences. We have identified two possible ways to deal with this. First, there is an approach that theoretically solves the issue and that can be approximated relatively easily in practice when models are used to generate execution traces. It is based on the following theorem.

Theorem 6 *Let G be a task graph. If for any two paths π and π' in G $duration(\pi) \neq duration(\pi')$, then the sets of critical tasks of G and $\mathcal{G}_0(exec(G))$ are equal.*

Proof See Online Resource 1.

Thus, if we are able to guarantee that no two paths in our task graph have equal duration, then our technique is exact. If our execution traces are produced by a model over which we have control, then we can try to achieve this by adding a very small random number to the execution time of each task which has no effect on the global system properties.[5] This effectively reduces the chances of spurious critical paths in our experience.

A second approach to deal with spurious critical paths is by counter-example guided refinement and manual checking. Consider a situation in which a critical task t has two critical and direct predecessors t_1 and t_2. It could be the case that one of these is a false positive with respect to criticality. An experiment in which we have slightly increased the duration of t_1 removes one of the precedences and therefore the criticality of either t_1 or t_2. This testing of precedences can lead to an exact set of critical paths, but requires modification and re-execution of the system.

Example 4 The 0-approximated task graph for the executions in Fig. 3b is shown in Fig. 5. The tasks have been annotated with their execution times, and the critical paths as computed by Alg. 2 have been highlighted. Theorem 5 tells us that a critical task in the real task

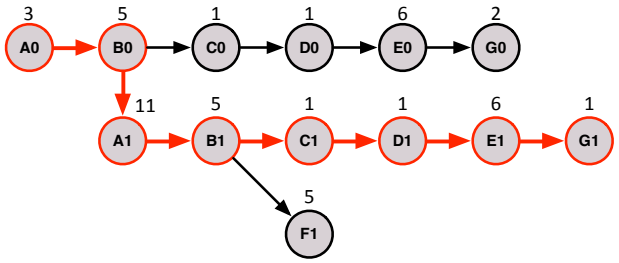


Fig. 5: Critical paths for the execution in Fig. 3b.

graph is also shown to be critical in the approximated task graph. The reverse implication is not true, i.e., there can be false positives in the approximated task graph. This can happen in situations where false positives are created in the arc relation due to timing coincidences. This is not the case in this example with only two objects, but if more objects are processed then false positives will show (e.g., when computing the critical paths for the execution traces in Fig. 2). In that case,

[5] Of course, it is very hard to guarantee beforehand that adding small random numbers to the execution time of tasks does not significantly change the system behavior. Significant changes by very small variations in the model, however, are a sign of limited robustness, which should be subject of analysis anyhow.

Th. 6 gives us a means to deal with this. Indeed, adding very small random numbers to the execution times of tasks solves this issue.

The 0-approximated task graph that is used in Th. 5 is fragile in the sense that the timestamps in the execution trace need to be *perfect*. Even the smallest gap between the end of some task and the start of another task implies that the tasks have no (direct) dependency in the 0-approximated task graph. This situation is fine for executions that are produced by, e.g., simulation models because in these environments the modeler has full control over the simulated time. If traces of real systems are to be analyzed, however, it usually is the case that the timestamping (e.g., through system logging) is not so perfect due to all kinds of overhead effects. The next section discusses this situation.

4.2 Dealing with executions of real systems

Traces of real systems usually do not have perfectly aligned timestamps for dependent tasks due to, e.g., timing overhead of control actions. In Sec. 3 we have taken a step to formalize this by allowing the execution engine (being a prototype or simulator) to add tasks to the application graph to get to the actual task graph that is executed. Definition 3, however, still assigns a start and end timestamp to every task of the task graph. To deal with executions with timing gaps, we now assume that we only have a subset of the execution start and end timestamps: only those of the tasks instances of the application graph, and not those of the task instances of the so-called overhead tasks. More formally, let $G = (T, \rightarrow, d)$ be a task graph, let $U \subseteq T$, and let $f = (start, end)$ be the execution of G. The execution without tasks in U is denoted by $f \ominus U$, and the graph consisting of U and arcs between tasks of U is denoted by $(U, \rightarrow_U)$. We thus assume that we have an execution $f \ominus U$, which contains timing gaps of at most

$$m(G, U) = max\left(\{duration(\pi) | \pi \text{ is a path in } (U, \rightarrow_U)\}\right)$$

time units. We use the ϵ-approximated task graph with a properly selected value for ϵ for these situations. Theorem 1 suggests that this can reduce the number of false negatives at the cost of having more false positives. This indeed is the trade-off: a larger value for ϵ is needed to produce reliable results for traces with timing gaps, but this also increases the number of false positives and thereby the number of false positive critical tasks.

In order to create a proper graph structure, we create the m-approximation from $f \ominus U$ and consequently run an adjusted version of Alg. 2. In this algorithm, we compute $start^-$ and $start^+$ while taking the gaps into account. This algorithm virtually re-inserts tasks from U in order to properly compute the float of the tasks. We therefore define a gap function $\delta^f : T \times T \rightarrow \mathbb{R}$ from the execution $f = (start, end)$ as follows:

$$\delta^f(t, t') \triangleq start(t') - end(t)$$

Algorithm 3 Robust critical-path analysis of a task graph $(T, \rightarrow, d)$ and a gap function δ.

Require: A topological ordering $t_1, t_2, \ldots, t_n$ of T according to $\rightarrow$, and that there is a unique sink task, i.e., t_n.
1: **for** $i = 1$ to n **do**
2: $\quad start^-(t_i) = max(\{start^-(t) + d(t) + \delta(t, t_i) \,|\, t \rightarrow t_i\})$
3: **end for**
4: $start^+(t_n) = start^-(t_n)$
5: **for** $i = n - 1$ to 1 **do**
6: $\quad start^+(t_i) = min(\{start^+(t) - (d(t_i) + \delta(t_i, t)) \,|\, t_i \rightarrow t\})$
7: **end for**
Ensure: $start^-$ and $start^+$ have been defined for all tasks

Algorithm 3 shows the algorithm for robust critical-path analysis. Note that if $U = \emptyset$, then $m(G, U) = 0$, which implies we can use $\mathcal{G}_0(f)$. In that case, the gap function always gives 0, and as a consequence Alg. 3 reduces to Alg. 2.

Theorem 7 *Let $G = (T, \rightarrow, d)$ be a task graph, let $exec(G) = f$, and let $U \subseteq T$. If the source tasks and the unique sink task in G are not elements of U, then a task $t \in T \setminus U$ that is marked critical by Alg. 2 when run on G, is also marked critical by Alg. 3 when run on $\mathcal{G}_{m(G,U)}(f \ominus U)$ and the gap function δ^f.*

Proof See Online Resource 1.

Under a restriction with respect to source and sink tasks that need to be present in the execution, we thus can also deal with traces that have imperfect timestamping. This is relevant for execution traces from real systems which often are not perfect in the sense that their timestamps match. Unfortunately, it is not easy to estimate the duration of the longest path in the set of missing tasks U. This knowledge, however, is essential because a value that is too small can result in loss of the over-approximation property. In practice, a large and safe value for ϵ should thus be choosen, which, however, increases the chances of false positives.

4.3 Adding application information

The critical-path analysis can be extended by adding knowledge. This is inspired by the Y-chart modeling approach [5], which distinguishes an application, a platform and a mapping. For instance, Fig. 1 shows these three elements: the graph structure of the tasks A – G

is the application, the architecture of the resources is the platform, and the mapping is indicated by the arcs between application and platform. Usually, it is well known what the application tasks and dependencies are as these stem from well-understood design of a certain functionality. This is reflected in Def. 1 which defines an application graph $A = (T_1, \mapsto)$ (also see Fig. 3a), and Def. 2 which defines a task graph $G = (T_2, \rightarrow, d)$. These are related: $T_1 \subseteq T_2$ and $t \mapsto t'$ implies that $t \rightarrow^+ t'$. Application of critical-path analysis (via some trace of G) gives us a set $C \subseteq T_2$ of critical tasks.

It is interesting to know which of the critical tasks are *blocked* in the sense that they are waiting on platform resources and not on application progress (e.g., earlier tasks that must produce some result). This is formalized as follows: A task t is *blocked* if for all its critical predecessors t' holds that (t', t) is not an application dependency as given by $\mapsto$.

$$blocked(t) \triangleq t \in C \wedge \forall_{t' \in C} t' \rightarrow t \Rightarrow t' \not\mapsto t$$

Example 5 Consider the execution in Fig. 3b, and its critical path in Fig. 5. The application structure as shown in Fig. 3a has been used to analyze the blocked tasks; see Fig. 6. The analysis gives us that task A1 is blocked because it is critical, and its only critical predecessor B0 is not an appplication dependency as given in Fig. 3a. The intrepretation is that A1 must wait for B0 to release M1 memory. This thus hints that increasing the amount of M1 memory might improve the system performance in this case. This is confirmed by the execution in Fig. 3c in which M1 is larger and the performance indeed is better.

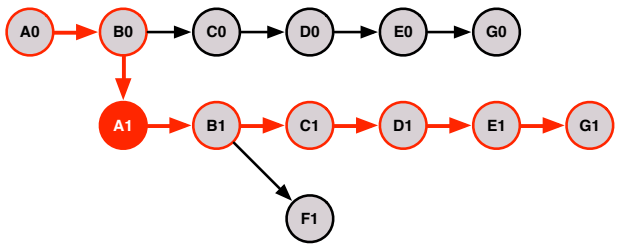


Fig. 6: The critical path for the trace from Fig. 3b, extended with *blocked* tasks (A1, highlighted in red).

4.4 Summary and complexity of critical-path analysis

Summarizing, application of critical-path analysis to an execution trace τ of an execution f requires the following steps:

1. Estimate the maximum gap size between tasks in τ. If τ has been created by a model, then 0 normally is appropriate. Underestimation can lead to invalid results, but overestimation can lead to useless results (because too many tasks are marked as critical).
2. Create $\mathcal{G}_\epsilon(f)$ from τ by applying Alg. 1. This algorithm also produces a topological ordering of the tasks on-the-fly (Prop. 1). Furthermore, by recording the start and end times and duration per task, and by keeping the direct successors and predecessors per task, we have an efficient data structure for the next step.
3. Apply Alg. 3 to $\mathcal{G}_\epsilon(f)$. This marks a subset of the tasks as critical.
4. If we have information on the application dependencies, iterate over the critical tasks and check for each of them whether it satisfies the *blocked* condition.

The computation complexity of step (1) is out of scope and assumed to be of constant time. The complexity of step 2, with respect to set operations, is $\mathcal{O}(|T| \cdot w(\mathcal{G}_\epsilon(f)))$ by Th. 4, and the additional work needed to construct the topological order on-the-fly and to record the direct successors and predecessors with each task does not add to that. Algorithm 3 also takes $\mathcal{O}(|T| \cdot w(\mathcal{G}_\epsilon(f)))$ set operations, because for each task we need to look at its direct successors and direct predecessors, and the cardinality of these sets also is bounded by the width. The same holds for the fourth step (which in fact can be done on-the-fly during step 3). This brings the total complexity to $\mathcal{O}(|T| \cdot w(\mathcal{G}_\epsilon(f)))$.

5 Trace distance analysis

5.1 Model calibration and validation

Earlier, we have introduced a graph representation of a trace. In this section, we define a (pseudo-)distance between two graphs, which induces a distance on traces and on executions. We use this distance for two purposes: automatic model calibration, and visualization of differences for, e.g., validation.

Calibration. Model-based methods usually contain a calibration step in which model parameters are tuned so that model results match a reference as closely as possible [30]. The distance between execution traces is the subject of the minimization process, and often system-level metrics of interest are used for this. It may happen, however, that these system-level metrics of interest, e.g., total execution time, do not differ significantly between traces. This is the case when model parameters affect parts of the system which are not critical with respect to the metric of interest. These model parameters

should, however, still be tuned appropriately as other use cases might render these system parts critical. In such cases, our distance, which takes the *structure* of the execution traces into account may be able to make a distinction which can be used for minimization.

Validation through visualization. Manual comparison of large execution traces is a very time consuming and error-prone task. The distance between execution traces that we define below can be used to highlight those tasks in the execution traces that contribute most to the difference. This is a useful technique to quickly find parts in traces where structural differences occur. This then can be used, e.g., for manual validation that a model properly models the internals of some system or to analyze the effect of design changes and parameter settings, such as a change of the memory size.

5.2 Graph edit distance

In this section we define a pseudo-distance on execution traces through a distance – a variation of *graph edit distance* – on their ∞-approximated task graphs.

There is a vast body of work on graph edit distances, see, e.g., [13] for a survey. One crucial difference with existing work is that we do not consider re-labeling of vertices, since a task in one trace should only be related to the same task in another trace. We therefore define the distance between two graphs as the minimum number of addition and removal operations on the sets of vertices and arcs that are necessary to transform one graph into the other. This makes the otherwise NP-hard problem much easier, as we show below.

The procedure to transform one graph $G_1 = (V_1, E_1)$ into another graph $G_2 = (V_2, E_2)$ is the following:

1. Remove arcs not in E_2. This takes $|E_1 \setminus E_2|$ removals.
2. Remove vertices not in V_2. This takes $|V_1 \setminus V_2|$ removals.
3. Add vertices not in V_1. This takes $|V_2 \setminus V_1|$ additions.
4. Add arcs not in E_1. This takes $|E_2 \setminus E_1|$ additions.

There is no procedure that transforms G_1 into G_2 and that takes fewer set additions and removals. We reach a contradiction by assuming that such a procedure exists, and by examining what this implies for, e.g., a vertex that is removed by our procedure but not by the new procedure.

The symmetric difference between sets A and B, denoted by $A \Delta B$, is defined as $A \Delta B = (A \setminus B) \cup (B \setminus A)$. The minimum number of addition and removal operations needed to transform G_1 into G_2 is therefore equal to the sum of the sizes of the symmetric differences of the vertex set and the arc set: $|V_1 \Delta V_2| + |E_1 \Delta E_2|$. The procedure to transform G_1 into G_2 is the basis for our definition of the distance between two graphs.

Definition 6 (Graph edit distance) Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The distance between G_1 and G_2, denoted by $d_{\text{ged}}(G_1, G_2)$ is defined as $|V_1 \Delta V_2| + |E_1 \Delta E_2|$.

Proposition 2 (Triangle inequality) *The graph edit distance satisfies the triangle inequality:* $d_{\text{ged}}(G_1, G_2) + d_{\text{ged}}(G_2, G_3) \geq d_{\text{ged}}(G_1, G_3)$.

Proof See Online Resource 1.

This brings us to the definition of a distance between executions. Theorem 2 says that the ∞-approximation is *safe* in the sense that it contains the reachability relation of the original task graph. This is the reason why we use this approximation in the following definition.

Definition 7 (Execution distance) The distance between executions f_1 and f_2, denoted by $d_{\text{ex}}(f_1, f_2)$, is defined as $d_{\text{ged}}(\mathcal{G}_\infty(f_1), \mathcal{G}_\infty(f_2))$.[6]

The execution distance can be computed for executions that are represented by traces using Alg. 1 and by computing the edit distance of the resulting graphs as defined in Def. 6.

Theorem 8 *Execution distance is a pseudo-metric.*

Proof See Online Resource 1.

The execution distance is not a metric, because there exist different executions that are at distance 0. A trivial example consists of two task graphs with a single task t that result in two different executions: f_1 with $start_1(t) = 0$ and $end_1(t) = 1$, and f_2 with $start_2(t) = 0$ and $end_2(t) = 2$. Clearly, $f_1 \neq f_2$, but $d_{\text{ex}}(f_1, f_2) = d_{\text{ged}}(\mathcal{G}_\infty(f_1), \mathcal{G}_\infty(f_2)) = d_{\text{ged}}((\{t\}, \emptyset), (\{t\}, \emptyset)) = 0$.

Our distance does not take timestamps explicitly into account like, e.g., [27,23] do, but only looks at the temporal order of events. Furthermore, interchanges of pairs of start events and pairs of end events are not noticed by our distance. Our distance works at the task level and can be interpreted as the structural difference between the partially ordered sets of task executions that define the system behavior.

5.3 Visualization of differences

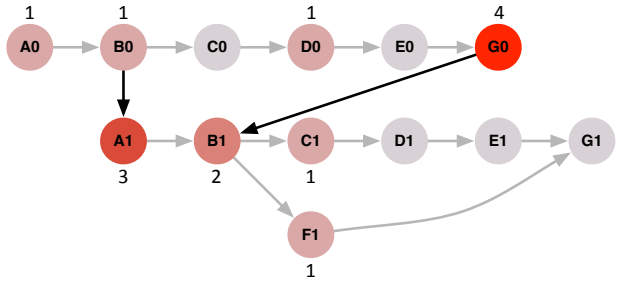
It is very hard to manually discover differences between large traces that are similar. Based on the definition of

[6] We liberally skip the duration part of the approximated task graph definition to avoid more notation.

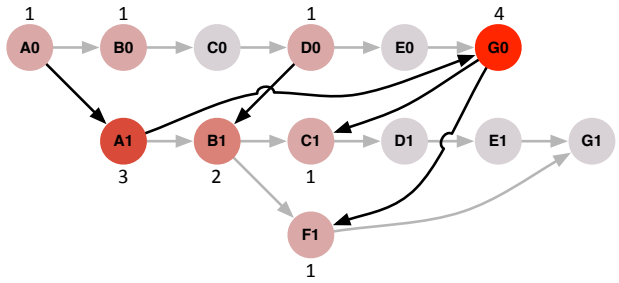
the distance between graphs above, we assign a certain positive *weight* to a vertex, which intuitively describes how much it adds to the distance. We can then color the vertices according to their weight. Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The weight of a vertex $v \in V_1 \cup V_2$ is defined as follows:

$$weight(v) \triangleq |\{u \in V_1 \Delta V_2 | u = v\}| + |\{(u, w) \in E_1 \Delta E_2 | u = v \vee w = v\}|$$

Example 6 Consider the traces shown in Fig. 3b and Fig. 3c respectively. Figures 7a and 7b show their ∞-approximated task graphs and the weights of the vertices. Note that both task graphs have the same set of vertices. The arcs of $\mathcal{G}_\infty(f_1)$ that are not in $\mathcal{G}_\infty(f_2)$ have been emphasized in Fig. 7a, and similarly, the arcs of $\mathcal{G}_\infty(f_2)$ that are not in $\mathcal{G}_\infty(f_1)$ have been emphasized in Fig. 7b. The weight of a vertex is equal to the number of times that it appears in such an emphasized arc, e.g., 4 for G0. This indicates that the differences around G0 are the largest. In Sec. 7 we show how this is visualized in the Trace tool.



(a) $\mathcal{G}_\infty(f_1)$



(b) $\mathcal{G}_\infty(f_2)$

Fig. 7: The ∞-approximated task graphs of the execution f_1 shown in Fig. 3b and f_2 shown in Fig. 3c, annotated with the weights of the vertices.

5.4 Complexity of execution distance

Computation of the graph edit distance as in Def. 6 requires a number of set inclusion checks that is linear in the number of the vertices and the arcs. The number of vertices is equal to the number of tasks in the execution. The sizes of the arc relations are in general quadratic in the number of tasks. However, because we are dealing with transitively reduced graphs, we are able to express the complexity of the total distance computation in terms of the number of events and the parallelism in the system. The total distance is not a factor.

Theorem 9 *The time complexity to compute the execution distance from two traces is $\mathcal{O}(|T| \cdot w)$, where T is the largest of the traces' task sets and w is the maximum of the widths of the two task graphs.*

Proof See Online Resource 1.

6 Implementation in Trace tool

6.1 Trace introduction

We have implemented the analysis techniques in Trace, which is a configurable visualization tool for quantitative data to support design-space exploration. Currently, Trace supports two forms of visualization:

1. Gantt chart visualization capable of showing (large sets of) activities on resources (and dependencies between them) as a function of the time, and
2. six visualization variants for multi-dimensional trade-off spaces, in which the results for each design alternative can be linked to the Gantt charts that explain the trade-offs.

In this paper, we focus on the Gantt chart visualization capabilities of Trace. This is supported with the traditional activity focused view where the vertical axis lists activities as shown, e.g., in Fig. 2. In addition, a resource focused view as commonly used in high-tech system design is supported for the same trace in which the vertical axis lists resources. Both views allow for manual comparisons of multiple traces by supporting a combined visualization into a single Gantt chart. Trace differs from other Gantt chart tools in that it is highly configurable to match a large variety of application domains.

Gantt chart visualization in Trace relies on the concepts of resources, claims and dependencies, which are visualized based on coloring, filtering and grouping functionalities that depend on the configuration concepts of attributes and contexts. Task executions as described in this paper map to activities of tasks due to

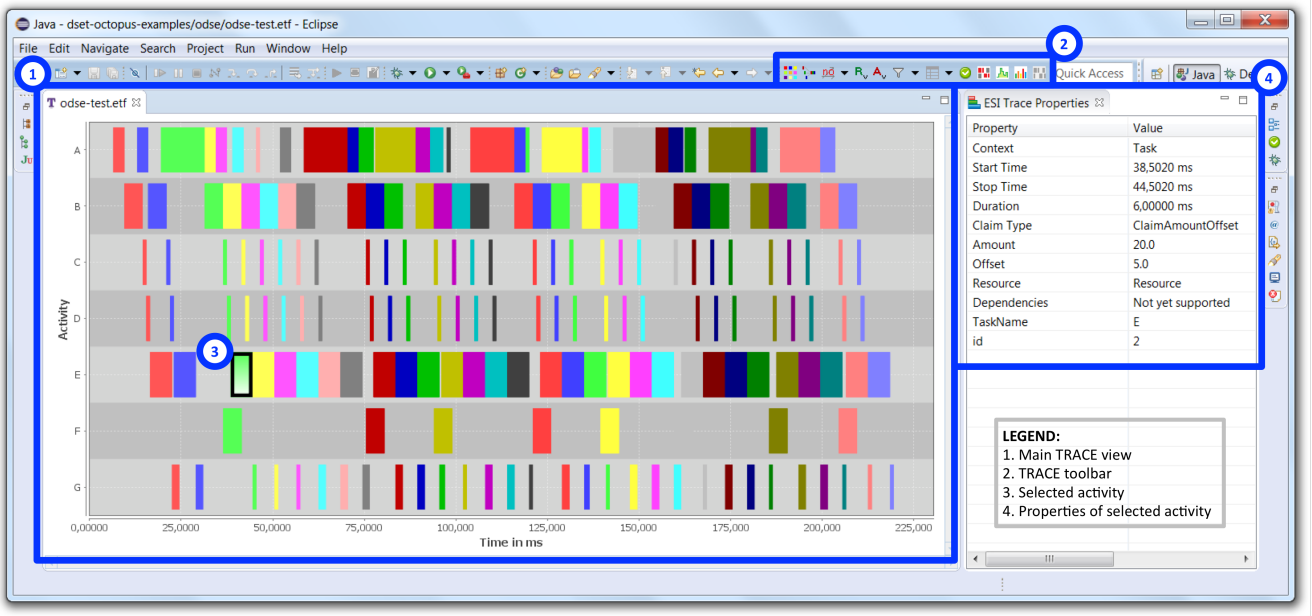


Fig. 8: A screenshot of the TRACE plug-in in the ECLIPSE IDE.

the availability of the resources they require. TRACE uses the concept of claims to describe undividable and unique accesses of a resource by an activity or task. To this end, claims contain start and end timestamps and a reference to a resource of three possible types:

1. Full claim resources are either occupied or not. A typical example is a processor core
2. Amount claim resources are occupied by a certain amount of their pre-specified capacity. An example is a storage resource for which insight in its occupancy over time is of interest
3. Amount claim with offset resources are the same as amount claim resources, where also an offset must be specified. This can for example be useful in case of evaluating the address range mapping of activities or tasks on a memory over time.

Given that a claim is an undividable primitive, an activity or task may imply many claims for the same resource (at different times). Claims are shown as (colored) blocks in the Gantt chart. TRACE also supports visualization of dependencies between claims as arrows in the Gantt chart (e.g., originating from the application graph if such information is known or from the approximated task graph).

Claims, resources and dependencies refer to values for the attributes in a certain context. A context is just a container for multiple attribute definitions. An attribute definition represents a configurable visualization property such as a descriptive name or criticality information of an activity or task. Examples of claim attributes for the professional high-end printing domain discussed in Sec. 7 include the job identity and page number. Note that these attributes do not exist in other domains.

Coloring of claim representations (the blocks) in a Gantt chart is based on equality of user-selectable attribute values. Filtering allows restricting the claims, resources or dependencies to those with user-selectable values for their attributes. In the resource view, the grouping functionality of TRACE merges the vertical rows for individual resources into combined rows. This can for instance be used to collapse all claims on the individual cores of a processor onto one row in the Gantt chart.

TRACE is implemented in JAVA and exploits the ECLIPSE framework [11] for its GUI functionality. It is freely downloadable from [12] as an ECLIPSE plug-in or as a stripped-down stand-alone application for various common platforms. Figure 8 shows a screenshot of the TRACE tooling. Trace files with the *etf* extension can be opened, viewed and analysed. The buttons at the right-hand side of the toolbar (left of the *Quick Access* field) give access to the functionality mentioned above, including critical-path analysis and distance analysis.

6.2 Critical-path analysis in TRACE

The critical-path analysis functionality in TRACE can be applied to any execution trace that can be loaded by the tool. It has the following parameters:

– *Merge claims.* Tasks that use multiple resources are modeled in Trace by multiple claims. Also effects like resource sharing in combination with pre-emption cause fragmentation of a task over multiple claim entries. The merge option merges all claims that belong to a single task by taking the minimal start time and the maximal end time of all these claims. This takes $\mathcal{O}(c^2)$ time, where c is the number of claims in the input trace.
– *Apply to filtered view.* The analysis can be applied to subset of claims that is currently in the filtered view.
– ϵ *value.* The ϵ parameter for task graph construction can be defined manually.
– *Use data dependencies.* It is possible to provide application level dependencies in order to identify resource blocking. The data dependencies are defined in a certain XML format.

6.3 Trace distance analysis in Trace

The distance between traces can be computed and visualized in two different modes:

– *Comparison mode.* In this mode, two traces are compared and annotated for visualization, and their distance is written to the console. This mode is useful for validation and spotting differences quickly.
– *Calibration mode.* In this mode multiple traces can be compared with a reference trace. The reference trace is not annotated, but the other traces are annotated for visualization. Furthermore, the distances of the traces to the reference trace are written to a console.

The distance computation needs the notion of equivalence between the claims of the traces. Our implementation bases equivalence of claims on the claim attribute values and on the attribute values of the resource of the claim.

7 Applications

7.1 Case study

The carrying example of this section is the datapath of a professional high-end digital color printer. The datapath is the embedded system inside the printer that is responsible for a large part of the image processing. Figure 9 shows a high-level overview of the three main use cases, and the four main functional components in the datapath. The SCAN functionality uses the scan and export paths to process input from the scan-engine and produce a file that, e.g., is copied to a removable USB device. The COPY functionality uses the scan and print paths to process input from the scan-engine and produce input for the print-engine. The PRINT functionality uses the rip and print paths to rasterize an input document received from the network and to send it to the print-engine. A *job* is a combination of input data (either data from the scan-engine or a document) and the path. We distinguish 4 basic paths: scan, export, rip, print, and the 3 combined paths SCAN (scan + export), PRINT (rip + print) and COPY (scan + print).

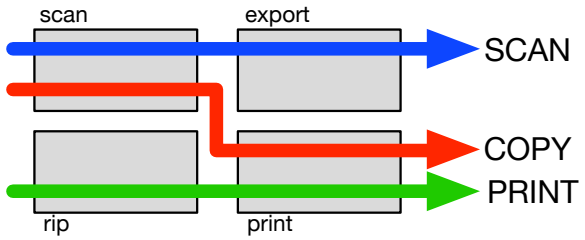


Fig. 9: High level view on the datapath.

The image processing within the datapath has a number of interesting features. First, the input, which is a stream of page data objects, is processed in a pipeline. This means that processing of page $i + 1$ may begin when the processing of page i has not finished (i.e., it is in a later pipeline stage). Second, the granularity of the data changes: some steps work on a full page, whereas other steps only work on parts of pages (these steps can also be pipelined). Third, the data communication between steps is implemented via bounded memory buffers: A full buffer can block the execution of tasks. Fourth, a number of tasks share resources such as the CPU and the harddisk. The pipelining can result in situations in which multiple tasks compete for these resources and thus interfere with each others execution. Fifth, some of the steps use compressed data. The amount of data therefore is highly dependent on the content: plain text compresses much better than a color photograph. These features make it very hard to predict the impact of design choices on the job execution time, which is one of the main system-level metrics of interest.

The four basic paths shown in Fig. 9 are mainly implemented in software that is mapped to a general-purpose PC platform running an off-the-shelf operating system. Besides the datapath software, the PC platform also runs the user interface software. The main chal-

lenge in the design of the datapath is a proper balance in the quality vs. speed vs. cost trade-off. The virtually infinite set of possible inputs make this even harder, as the input can impact both the quality and the speed.

7.2 Model-based support for product families

The system we consider is a prototype in a product family. Successors are usually created by an evolutionary process starting from an existing system. Our case study encompasses model-based support for the various steps in this evolutionary process with a focus on system performance. A single iteration of this continuous process is shown in Fig. 10. There are two loosely coupled development flows: the prototype development, and the model development. The starting point is an existing prototype (prototype X). This existing system has been used to create a model through a manual process, and this has resulted in a parameterized high-level discrete-event simulation model of the system with a focus on performance metrics: model A. The existing system is also used for calibration and validation purposes, resulting in a model A', in which proper parameter values have been defined. Section 7.3 explains the method that we have used to this end. This calibrated and validated model A' can then be used to optimize the existing design and to gain insight in intricate runtime behavior. An example of this is explained in Sec. 7.4. Furthermore, a new prototype Y, has been planned for the near future. Differences in the requirements of X and Y have resulted in a number of design questions, and the answers should make the evolution from X to Y smooth. In Sec. 7.5 we show how we can use model A' and its adaptation to envisoned changes, model B, to answer some of the design questions for the new prototype Y. Critical-path analysis and trace distance analysis play key roles in all these activities.

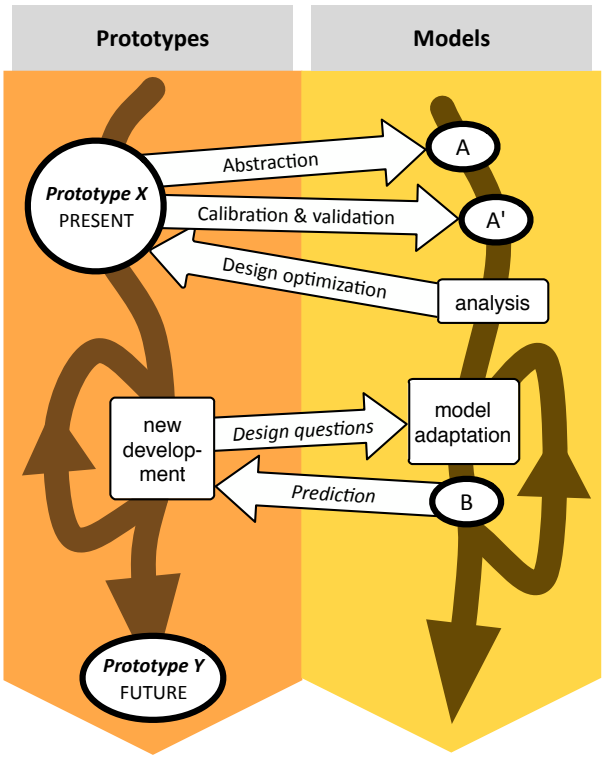


Fig. 10: Model-based support for design decisions in an evolutionary development process.

7.3 Abstraction, calibration and validation

7.3.1 Abstraction

We have built parameterized discrete-event simulation model parts of the scan, rip and print paths (see Fig. 9) using the OctoSim tool [15]. These model parts can be used separately, or can be composed to realize the COPY and PRINT paths. The model parts are similar to the model shown in Fig. 1 in that they contain an application graph consisting of image processing (IP) tasks which are mapped to a general purpose PC platform. All three model parts consist of linearly dependent tasks: sIP0 $\rightarrow$ sIP1 $\rightarrow \cdots \rightarrow$ sIP5 for scan, IP1 $\rightarrow \cdots \rightarrow$ IP5 for rip, and IP6 $\rightarrow \cdots \rightarrow$ IP9 for print. Furthermore, the rip path can be replicated in order to increase parallelism: pages of a single input document are then distributed over the parallel rip paths on demand. These model parts together form model A in Fig. 10.

The model has parameters for each task, and also for the OS and hardware components, which enable us to calibrate the behavior. The OctoSim model extrapolates in two ways: (i) it composes the individually calibrated tasks into the full application with its dependencies, and (ii) it adds platform resources with their scheduling and their constraints. The most important model parameters are:

- nominal task execution time,
- data compression ratios,
- buffer sizes,
- the number of parallel rip components, and
- harddisk scheduling behavior.

Note that the first two parameters are job specific (i.e., they depend on the image content), whereas the last three are determined by the datapath.

7.3.2 Calibration

We have taken two kinds of measurements in order to calibrate the model. First, we have added special logging statements to the datapath application code that log the start and end of image processing tasks. By executing scan and rip jobs on the existing prototype X with the extended datapath code, we obtain reference execution traces. The nominal execution times of tasks and the data compression ratios have been extracted from these traces.[7] Second, we have done dedicated experiments to investigate the harddisk behavior. These have shown that the OS favors writers over readers, and that approximately 50% of the specified maximum data transfer rate can be achieved when the load on the disk is high.

The measurements have enabled us to calibrate nominal task execution times, and data compression ratios in the PRINT path for the jobs, and also the harddisk behavior. This brings us to the buffer sizes, and the data compression in the scan path. Many of the model buffers map directly to real buffers and therefore their sizes are known from the design of prototype X. However, the model contains some approximations with respect to two buffers in the PRINT path, because explicit modeling of these buffers is too complex – if possible at all – since it involves an external component whose internals are largely unknown. Furthermore, the handling of compressed data on sub-image granularity in the scan path has also been approximated, since extracting the necessary information through system logging and adding that in the model would have been too complex. In order to choose proper values for the approximated model buffers and data compression ratios, we have used our trace distance to fit our model as good as possible to a number of reference traces from prototype X. We illustrate the added value of the trace distance analysis with the PRINT path calibration.

We have selected six representative jobs for use with the PRINT path, and we have used the measurements explained above to calibrate six instances of our model for these jobs (for the situation with no parallel rip). The six resulting models still have two free parameters regarding the sizes of the approximated buffers, $B2$ and $B3$. For each job we have computed model traces for 100 different configurations of $B2$ and $B3$: both buffer sizes are varied from 1 to 10. For each buffer configuration, we have determined the execution distance, d_{ex} as given in Def. 7, and the difference in total execution time with respect to the reference trace, denoted by Δ. We have collected the configurations that produce the minimum distance, and the results are shown in Tab. 1.

[7] Note that these execution traces are job dependent. Furthermore, we have ensured that the execution of the individual tasks is free from interference from other tasks in order to get the nominal execution time.

Table 1: The minimum configurations between the model and reference trace.

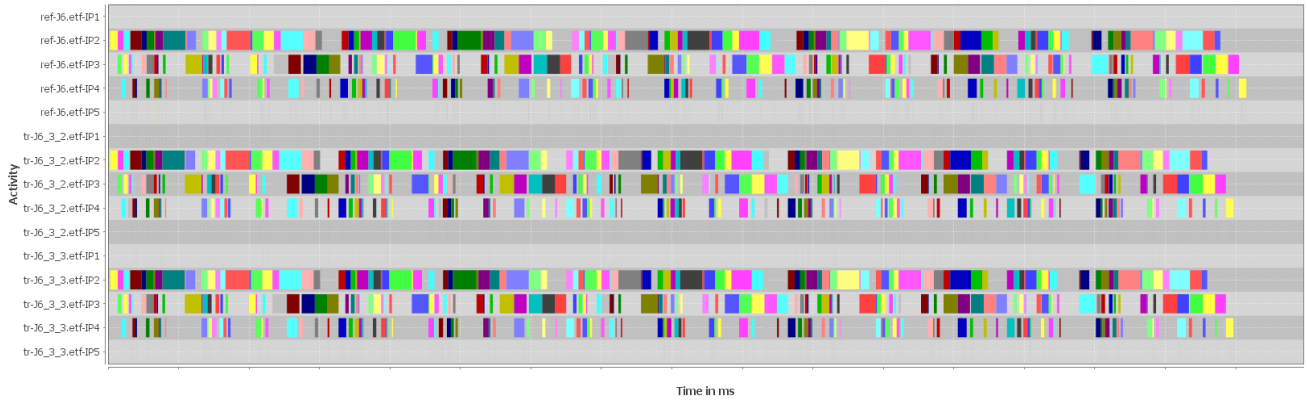
Job	Criterion	Minimal configurations	#
J1	d_{ex}	$B2 = 3,\ B3 \geq 5$	6
	Δ	$B2 \geq 4,\ B3 \geq 4$	49
J2	d_{ex}	$B2 = 3,\ B3 \geq 3$	8
	Δ	$B2 \geq 5,\ B3 = 2$	6
J3	d_{ex}	$B2 \geq 3,\ B3 \geq 3$	64
	Δ	$B2 \geq 3,\ B3 \geq 2$	72
J4	d_{ex}	$B2 = 3,\ B3 \geq 4$	7
	Δ	$B2 = 3,\ B3 = 3$	1
J5	d_{ex}	$B2 = 3,\ B3 \geq 4$	7
	Δ	$B2 = 3,\ B3 \geq 3$	8
J6	d_{ex}	$B2 = 3,\ B3 \geq 3$	8
	Δ	$B2 = 3,\ B3 = 2$	1

Table 2: Distances to the reference traces for minimum buffer configurations.

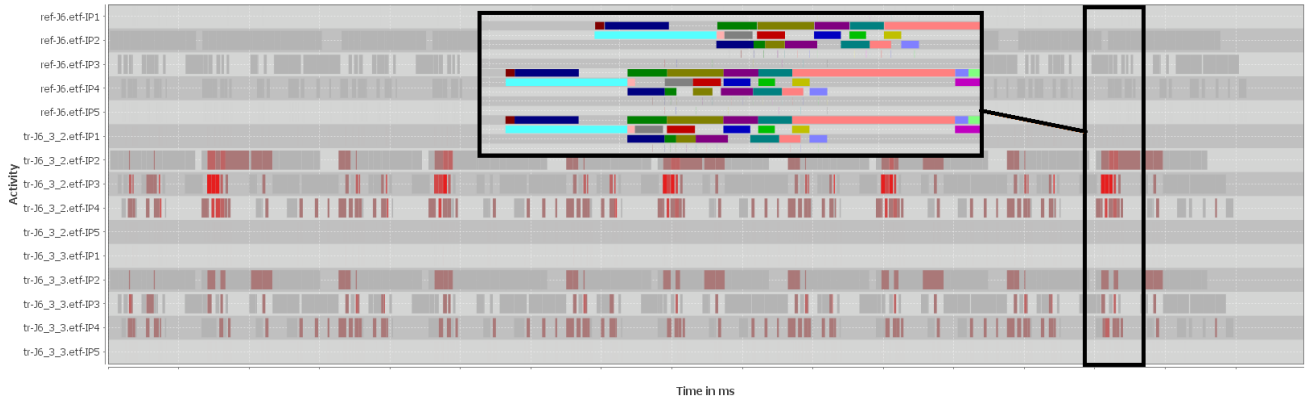
Job	Criterion	$B2$	$B3$	d_{ex}	Δ	Δ%
J1	d_{ex}	3	5	814	30	0.07
	Δ	4	4	1267	28	0.06
J2	d_{ex}	3	3	814	1223	0.57
	Δ	5	2	1133	119	0.06
J3	d_{ex}	3	3	762	1927	1.31
	Δ	3	2	869	1927	1.31
J4	d_{ex}	3	4	795	1595	0.97
	Δ	3	3	840	1464	0.89
J5	d_{ex}	3	4	781	1406	0.77
	Δ	3	3	842	1406	0.77
J6	d_{ex}	3	3	791	1852	1.15
	Δ	3	2	989	1848	1.14

The results in Tab. 1 show that there generally is not a unique minimum buffer configuration. Our explanation is that at a certain point, increasing a buffer has no effect on the behavior any more because the additional slots cannot be used due to other (timing) constraints in the system. The results also show that the optimization based on total execution time provides less consistent results over the jobs. E.g., the optimal buffer configurations with respect to total execution time Δ for J2 and J4 are totally disjoint. This is not the case for the buffer configurations that are obtained with the execution distance d_{ex}: the configurations $B2 = 3$ and $B3 \geq 5$ provide minimum distance for all jobs. We interpret this as a sign of robustness of our approach. The optimization is based on the internal behavior of the system and not only on the end result, which may falsely be pushed in the right direction by improper configuration assignments.

In Tab. 2 we have picked the "smallest" concrete configurations from the sets of minimum configurations



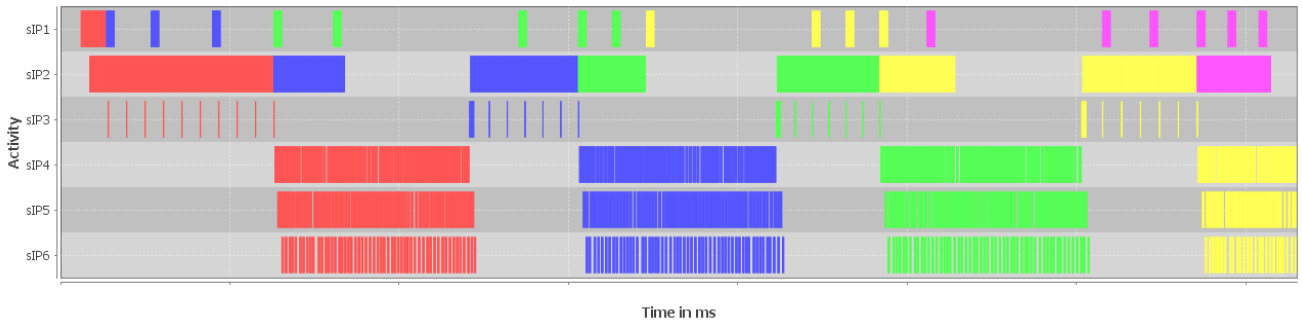
(a) A Trace activity view of a reference trace (ref-J6.*) and two model traces (tr-J6_3_2.* and tr-J6_3_3.*).



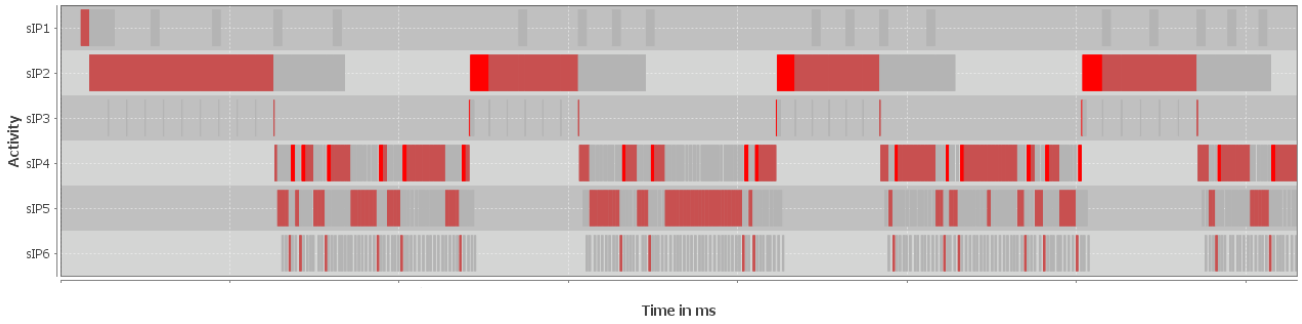
(b) A Trace activity view of the differences in the model traces.

Fig. 11: Trace comparison of two model traces to the reference trace of job J6. The model trace with B2=3 and B3=3 (the bottom trace in (a)) fits best.

shown in Tab. 1, and have listed the actual distances to the reference traces and the relative error with respect to total execution time ($\Delta\%$). The table shows that there is a trade-off between the distances when used for optimization, and that model fitting using execution time Δ as a criterion results in a different configuration than when execution distance d_{ex} is used. The relative error with respect to the total execution time, an important system-level metric, is good for both criteria. In order to understand the value that the execution distance d_{ex} adds, we have manually inspected the traces of the configurations listed in Tab. 2. Figures 11a and 11b give an indication of the situation for job J6. It is very hard to spot the differences between the reference trace and the two model traces in Fig. 11a. Our visualization based on trace distance, shown in Fig. 11b and explained in Sec. 5.3, however, highlights the tasks which play a role in the distance to the reference trace, and enables us to quickly zoom into the interesting areas. The zoomed part shown in Fig. 11b shows an interesting area with the usual colors. Careful analysis reveals that indeed the middle trace differs structurally more from the reference trace (top) than the other model trace (bottom). Our conclusion from the visual inspection is that model fitting based on execution distance d_{ex} gives, in this case, a better result than fitting based on latency, although the relative error $\Delta\%$ might be slightly larger. As a general rule, we can say that when the main system-level metrics, such as total execution time, are very close for different configurations (as is the case in our example), then the trace distance d_{ex} can be used to make the final decision: it is a complementary tool for calibration. Although using the distance is more complicated and more expensive than using the total execution time, this can pay off when the model is changed slightly and used for prediction (see Fig. 10). In that case, the critical parts of the application may change. Calibration based on total execution time only will have neglected these new critical parts earlier, because total execution time is determined only by the critical tasks. Structural differences in non-critical (with respect to total execution



(a) A Trace activity view of a scan path execution.



(b) A Trace activity view of (a) after critical-path analysis using application information.

Fig. 12: Identification of blocking resources. In trace (b) several instances of sIP2, sIP3 and sIP4 are highlighted because they have been blocked on resources.

time) parts are not taken into account when calibrating on total execution time only. They are taken into account by our trace distance.

7.3.3 Validation

For validation purposes we have executed the PRINT jobs on the prototype with 2 and 3 parallel rip components, which can be reflected directly in the model. This has resulted in a number of new reference traces in which tasks do interfere with each other when, e.g., multiple tasks run concurrently on a CPU core, or access the harddisk. We consequently have changed the appropriate model parameters of our calibrated models, and generated model traces. This exercises the two main extrapolation mechanisms in the OctoSim models: task interaction on the shared CPU and HD resources, and task composition and interaction on buffers.

Comparison of the model and reference traces has shown that our model quite accurately predicts the total execution time, which is the main metric of interest: the predicted execution time deviates less than 10% from reality for our test set. This kind of predictive value is good enough to investigate trends, gain insight and identify risks in development. We thus have validated our calibrated models and the OctoSim way of extrapolation, and can continue with the optimization and prediction steps as shown in Fig. 10.

7.4 Design optimization and insight creation

The current configuration of the buffers in the scan path allows enough pipelining in the sense that the output of the scan-engine can be consumed faster than the scan-engine can produce. There are, however, foreseeable situations in which this can be compromised, and it is therefore the question how much the buffer configuration of the scan path can be optimized further in order to counter these situations. We have used our model to investigate this.

Figure 12a shows the Trace activity view of the scan path. In order to be able to identify blocking resources, we have added application information, namely that sIP0→sIP1→ $\cdots$ →sIP6. Figure 12b shows the critical path, and also that a significant number of instances of sIP2, sIP3 and sIP4 are blocked.[8] This creates the pause in the sIP2 activity during the processing

[8] The fact that sIP3 instances are blocked is not visible in the figure due to the small execution time of sIP3. The console output of the critical-path analysis step in the Trace tool, however, contains this information.

of a single data object; this is clearly visible in Fig. 12a by the coloring. Activity sIP2 needs an output buffer C2 for its execution, and activity sIP3 also needs an output buffer C3 for its execution. From the coloring per image we can easily see that sIP3 of image i must wait for sIP4 of image $i-1$ to release a slot of buffer C3. Similarly, sIP2 (which works on parts of an image, called a block) of block j of image i must wait for sIP3 of an earlier block $< j$ of the same image i. The delay can thus be solved by increasing the number of slots of either buffer C2 or buffer C3. Hardware constraints have restricted us to buffer C2. We have increased the size of C2 in the model, and this has resulted in a maximum of 30% increase in datapath performance. There is a clear trade-off between the buffer space used and the performance: more buffer space increases the pipelining and thereby the performance. These results have been validated on the existing prototype.

We have also used model A' to gain insight in the scalability of the print path. As mentioned earlier, the steps IP1 – IP5 in the rip component can be replicated to increase the parallelism in the rip path. Because of pipelining effects, three parallel instances of the IP1 – IP5 set, result in at most 12 parallel tasks on the CPU (IP5 is not mapped to the CPU). The CPU, however, does not have that many cores and therefore these tasks potentially interfere with each other. We have found that the magnitude of the interference is dependent on the job structure. To be more precise, the ratios of the nominal execution time of IP2, IP3 and IP4 per page determine how well the system scales. If, for instance, IP2 takes much more time than IP3 and IP4 for all pages in a certain job, then replication of IP1 – IP5 indeed decreases the total execution time significantly. However, if IP2, IP3 and IP4 take approximately the same time, then replication is less helpful.

7.5 Prediction

Prototype Y will contain a print-engine that is faster than the print-engine in the existing prototype X. It is unclear whether the datapath, if re-used from prototype X, will become the bottleneck that impedes productivity. Therefore, we have done a model-based analysis of the new situation in which the print-engine of Y is attached to the datapath of X. We have used the six jobs mentioned above, for which we have calibrated and validated models based on prototype X. We have changed these models into models for prototype Y by changing a single parameter value: the speed of the print-engine. Consequently, we have evaluated two challenging scenarios: print-while-rip, and rip-while-print. The first scenario considers printing a job, while simultaneously another job is ripped. In the second scenario, we consider ripping a job, while simultaneously another job is printed. This reveals a bottleneck: the rip process can mostly not achieve the speed of the new print-engine. Figure 13a shows the Trace activity view of the rip-while-print of job J2. The view shows the rip of job J2 (IP0 – IP5), while simultaneously a number of copies of job J2 are being printed (IP6 – IP9). Using the filtering functionality of Trace, we can obtain the critical path of the rip only, which is shown in Fig. 13b. The critical activities are highlighted, and mainly involve parallel instances of IP2-2 and IP3-2, which are mapped to the CPU. The CPU bottleneck is further confirmed by a resource utilization analysis of the trace, which shows that for almost 80% of the time, the threads outnumber the CPU cores.

Further analysis of the other jobs support the observation that the CPU becomes the main bottleneck that impedes productivity. This information points to two possible ways to remove the bottleneck from the datapath: (i) optimize the algorithms in the critical steps IP2 and IP3, or (ii) replace the CPU with a faster model. We focus on the latter option and refine:

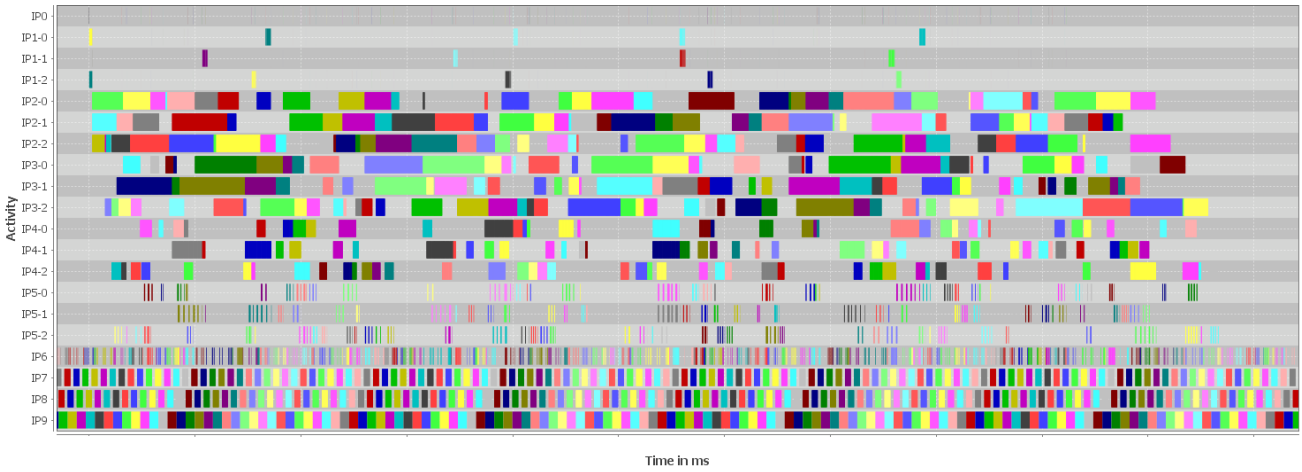
- use a similar model with a higher clock frequency
- use a model that has more cores

The first option assumes that the task execution times will decrease linearly with the increase in clock frequency. The second option is based on the observation that there are more tasks running on the CPU than there are cores for almost 80% of the time. Increasing the number of cores will reduce the interference between the tasks running on the CPU.

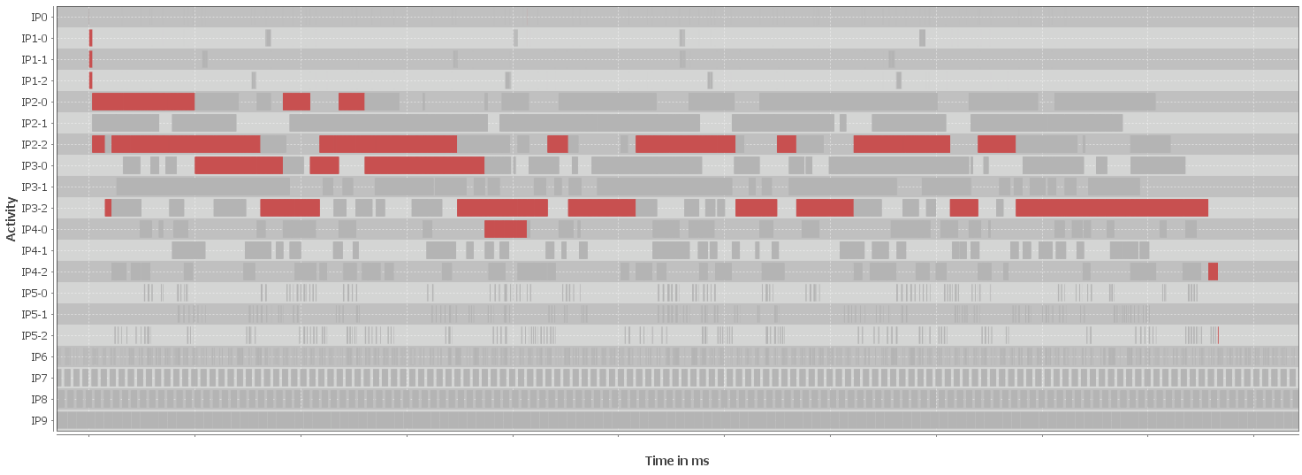
In order to make this more concrete, let us assume that the current CPU is an Intel Core i7-4790T with 4 cores that run at 2.7 GHz. For the first option we consider an Intel Core i7-4790 with 4 cores that run at 3.6 GHz. For the second option we consider an Intel Core i7-5820K with 6 cores that run at 3.3 GHz. The first two models are from the 4th generation Intel Core i7 family of desktop processors, and the third model is a high end desktop processor [20].

We assume that the nominal execution time scales linearly with the clock frequency. This means that the 4-core alternative is $\frac{3.6}{2.7} \approx 1.33$ times faster, and the 6-core alternative is $\frac{3.3}{2.7} \approx 1.22$ times faster. Making the speed and core-count changes in the model is straightforwardly done by changing two constants of the platform in the model.

Besides the identification of the CPU as the main bottleneck resource, we also observe that the model results suggest that often the HD is not fully used. In or-



(a) A Trace activity view of the rip-while-print of job J2.



(b) The critical path of the rip activities only.

Fig. 13: Identification of bottleneck activities and resources.

der to reduce the cost related to the bill-of-material, we therefore also want to consider a different HD for prototype Y, namely one that is cheaper and also slower than the one in prototype X. This gives us 6 design alternatives (3 choices for CPU, and 2 for HD) for prototype Y. We have evaluated the rip-while-print and print-while-rip scenarios for jobs J1 – J6 for all these design alternatives. Table 3 summarizes the experiments. Note that Y_1 has the same datapath as prototype X.

Our conclusion based on these experiments is that the faster print-engine in prototype Y poses a challenge for the current datapath. The results for Y_1 show that the CPU is the main bottleneck for this job set, and that improving the CPU hardware (Y_2 and Y_3) can alleviate this. The results in Tab. 3 for Y_4 – Y_6 show that a slower HD will result in a severe datapath bottleneck for the print case. It is interesting to see that a faster CPU can make things worse. Consider J6: In Y_4 the CPU is the bottleneck for the rip, and printing is at least at engine speed. In Y_5, we have alleviated the CPU bottleneck. The increased rip speed puts additional load on the disk, which then becomes the bottleneck for printing. This effect is even stronger for Y_6 which increases the rip speed of J6 even more. The automated critical-path analysis technique eases the interpretation of the already complicated execution traces (e.g., in Fig. 13a) and the identification of the bottlenecks. These results are based on model B which makes some rather crude assumptions. However, we believe that the conclusions on the dynamics and trends are valid, and increase the understanding w.r.t. the available design choices and risks.

Table 3: Model-based predictions for the current prototype X and six variants of the future prototype Y for the *rip-while-print* and *print-while-rip* scenarios. Bottlenecks and whether rip and print paths achieve at least engine speed of the respective prototypes are shown. The ✓signifies that engine speed is achieved, the − signifies that the engine speed is almost achieved (within 10%), and the × signifies that engine speed is not achieved (>10% lower).

				Main datapath bottleneck						rip / print @ engine speed					
	CPU	HD		J1	J2	J3	J4	J5	J6	J1	J2	J3	J4	J5	J6
X	4 cores, 2.7 GHz	Fast	rip	-	CPU	-	-	-	-	✓	−	✓	✓	✓	✓
			print	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
Y_1	4 cores, 2.7 GHz	Fast	rip	-	CPU	CPU	CPU	CPU	CPU	✓	×	−	×	×	×
			print	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
Y_2	4 cores, 3.6 GHz	Fast	rip	-	CPU	-	-	-	-	✓	×	✓	✓	✓	✓
			print	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
Y_3	6 cores, 3.3 GHz	Fast	rip	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
			print	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
Y_4	4 cores, 2.7 GHz	Slow	rip	-	CPU	CPU	CPU	CPU	CPU	✓	×	−	×	×	×
			print	HD	HD	HD	HD	HD	-	×	−	−	−	×	✓
Y_5	4 cores, 3.6 GHz	Slow	rip	-	CPU	-	-	-	-	✓	×	✓	✓	✓	✓
			print	HD	HD	HD	HD	HD	HD	×	×	×	−	×	−
Y_6	6 cores, 3.3 GHz	Slow	rip	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓
			print	HD	HD	HD	HD	HD	HD	×	×	×	×	×	×

8 Conclusions

We have defined two analysis techniques that work on *execution traces*, which can be produced, e.g., by executable models or real systems. The critical-path analysis technique can be used to identify activities and resources with the property that their improvement will contribute to a better performance (i.e., a lower total execution time of the system). It is a very helpful tool to answer the typical "what is the bottleneck?" question during system design activities. Furthermore, the trace distance analysis can be used for two purposes. First, it can be used for model calibration purposes and has the benefit that it considers the *structure* of the execution. This is complementary to calibration efforts using system-level metrics such as total execution time in the sense that it can be used to refine the calibration when the system-level metrics make no (or a very small) distinction. Second, trace distance can be used to visualize and highlight the differences between traces. Both critical-path analysis and trace distance analysis are based on a common graph representation of execution traces. They have low complexity and can be applied to large traces efficiently. We have implemented our techniques in the Octopus toolset and Trace visualization tool and have demonstrated the applicability and usefulness on a case study from the digital printing domain. We conclude that our techniques indeed ease the interpretation of execution traces and enable system designers to find bottlenecks in systems and to efficiently compare system behavior on a structural level for calibration and validation purposes.

Acknowledgments

We thank the anonymous reviewers for their valuable comments that helped us to improve the paper. The research is partially carried out as part of the Octo+ program under the responsibility of Embedded Systems Innovation by TNO (TNO-ESI) with Océ Technologies B.V. as the carrying industrial partner. The Octo+ research is supported by the Netherlands Ministry of Economic Affairs. The research reported on in this paper is also partially supported by the ARTEMIS joint undertaking under grant agreement no 621439 (ALMARVI).

References

1. W.M.P. van der Aalst and B.F. van Dongen. Discovering workflow performance models from timed logs. In *1st Int. Conf. on Engineering and Deployment of Cooperative Information Systems*. Springer, 2002.
2. W.M.P van der Aalst et al. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47, 2003.
3. A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1, 1972.
4. A.V. Arkhangelski and V.V. Fedorchuk. The basic concepts and constructions of general topology. In *General Topology I*, volume 17 of *Encyclopaedia of Mathematical Sciences*, pages 1–90. Springer Berlin Heidelberg, 1990.
5. F. Balarin et al. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.
6. P. Barford and M. Crovella. Critical path analysis of TCP transactions. *SIGCOMM Computer Communication Review*, 30, 2000.
7. T. Basten et al. Model-driven design-space exploration for embedded systems: The Octopus toolset. In *4th In-*

ternational Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2010), volume 6415 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2010.
8. 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.
9. D. Bohme, F. Wolf, B.R. de Supinski, M. Schulz, and M. Geimer. Scalable critical-path based performance analysis. In *26th International Parallel Distributed Processing Symposium (IPDPS)*. IEEE, 2012.
10. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
11. Eclipse Foundation. Eclipse website. `http://www.eclipse.org/`.
12. Embedded Systems Innovation by TNO. TRACE website. `http://trace.esi.nl`.
13. X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis & Applications*, 13(1):113–129, 2010.
14. E. Goldratt. *Critical Chain*. North River Press, 1997.
15. M. Hendriks, T. Basten, J. Verriet, M. Brassé, and L. Somers. A blueprint for system-level performance modeling of software-intensive embedded systems. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2014.
16. M. Hendriks and F.W. Vaandrager. Reconstructing critical paths from execution traces. In *International Conference on Embedded and Ubiquitous Computing (EUC 2012)*. IEEE Computer Society Press, 2012.
17. T.A. Henzinger, R. Majumdar, and V.S. Prabhu. Quantifying similarities between timed systems. In *Formal Modeling and Analysis of Timed Systems*, volume 3829 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005.
18. J.K. Hollingsworth. An online computation of critical path profiling. In *1st ACM SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, 1996.
19. J. Huang, J. Voeten, and M. Geilen. Real-time property preservation in concurrent real-time systems. In *10th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2004.
20. Intel. Intel ARK website. `http://ark.intel.com/`.
21. A. Kastor and K. Sirakoulis. The effectiveness of resource levelling tools for resource constraint project scheduling problem. *Int. Journal of Project Management*, 27, 2009.
22. J. E. Kelley and M. R. Walker. Critical-path planning and scheduling. In *Eastern joint IRE-AIEE-ACM computer conference*. ACM, 1959.
23. C.K. Kengne, N. Ibrahim, M.-C. Rousset, and M. Tchuente. Distance-based trace diagnosis for multimedia applications: Help me TED! In *7th International Conference on Semantic Computing (ICSC)*. IEEE, 2013.
24. V.I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10, 1966.
25. K. G. Lockyer. *Introduction to Critical Path Analysis*. Pitman Publishing Co., 1964.
26. 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.
27. H. Mannila and P. Ronkainen. Similarity of event sequences. In *4th International Workshop on Temporal Representation and Reasoning*. IEEE Computer Society Press, 1997.
28. N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18(2), 1982.
29. H. Obweger, M. Suntinger, J. Schiefer, and G. Raidl. Similarity searching in sequences of complex events. In *4th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 2010.
30. A.D. Pimentel, M. Thompson, S. Polstra, and C. Erbas. Calibration of abstract performance models for system-level design space exploration. *Journal of Signal Processing Systems*, 50(2), 2008.
31. M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 2nd edition, 2002.
32. M. Schulz. Extracting critical path graphs from MPI applications. In *Int. Conference on Cluster Computing*. IEEE, 2005.
33. J. D. Wiest. Some properties of schedules for large projects with limited resources. *Operations Research*, 12, 1964.
34. M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1, 1990.
35. S. Wu, U. Manber, G. Myers, and W. Miller. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35(6), 1990.
36. 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.
37. Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Automated bottleneck-driven design-space exploration of media processing systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1041–1046. IEEE, March 2010.
38. Z. Zeng, A.K.H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *Proceeding of the VLDB Endowment*, 2(1):25–36, aug 2009.