The Computation Time Process Model

Martin KORSGAARD [1] and Sverre HENDSETH

Department of Engineering Cybernetics, Norwegian University of Science and Technology

Abstract. In traditional real-time multiprocessor schedulability analysis it is required that all tasks are entirely serial. This implies that if a task is written in a parallel language such as occam, all parallelism in the task must be suppressed to enable schedulability analysis. Part of the reason for this restriction is the difficulty in analysing execution times of programs with a complex parallel structure. In this paper we introduce an abstract model for reasoning about the temporal properties of such programs. Within this model, we define what it means for a process to be easier to schedule than another, and the notion of upper bounds on execution times. Counterintuitive temporal behaviour is demonstrated to be inherent in all systems where processes are allowed an arbitrary parallel structure. For example, there exist processes that are guaranteed to complete on some schedule, that may not complete if executing less than the expected amount of computation. Not all processes exhibit such counterintuitive behaviour, and we identify a subset of processes that are well-behaved in this respect. The results from this paper is a necessary prerequisite for a complete schedulability analysis of systems with an arbitrary parallel structure.

Keywords. real-time programming, job-level parallelism, Toc, occam, CSP

Introduction

Multiprocessor systems are becoming more widespread in real-time applications, and much research has been done in recent years on schedulability analysis of these systems. However, analysis of multiprocessor systems does not normally allow parallelism within a task (often referred to as "job-level parallelism"); multiple processors are instead utilised by having multiple simultaneously active tasks. There are a number of drawbacks with this approach, for example that adding more processors to a system will never decrease the response-time of its highest priority task. Another drawback is that programs written in a parallel language such as occam cannot in general be used as tasks when schedulability analysis is required, without suppressing the parallelism that is often inherent in these programs.

This paper introduces an abstract process model for describing the computational requirements of processes with an arbitrary parallel structure. The model abstracts away any notion of what a process actually does, leaving only a measure of the parallel structure, and the computation time required to complete each part of the process. These processes are called *computation time processes* (CTPs), and the model is called the *computation time process model*.

In this model, we consider just the SEQ/PAR structure for parallelism: processes are explicitly specified to execute in parallel or in sequence, and a parallel process does not terminate until all the sub-processes terminate. The sub-processes themselves may also have a SEQ/PAR-structure. This is one of the structures of parallelism used in occam and its derivatives.

[1]Corresponding Author: *Martin Korsgaard, Department of Engineering Cybernetics, O.S Bragstads plass 2D, 7491 Trondheim, Norway.* E-mail: `martin.korsgaard@itk.ntnu.no`.

CTPs do not model communication, and it is therefore not possible to derive a CTP from a general program written in e.g. occam. CTPs should instead be considered building blocks that can be used as a basis for a complete timing analysis. However, every SEQ/PAR program without communication or synchronisation will have a temporal behaviour that can be modelled by a CTP.

For this paper we will analyse a single CTP in isolation. Still, it will always be assumed that the process is executing in some environment where processors are shared with other tasks. Some of these tasks will affect the number of processors available to the process under consideration, so the number of available processors will be considered time varying and non-deterministic. To allow schedulability analysis of a complete system one would need to analyse how different CTPs interact when executed on the same platform. The results in this paper is a necessary first step towards such an analysis.

The structure of this paper is as follows: Section 1 briefly discusses multiprocessor schedulability and process models for parallelism. The CTP model is defined in Section 2.

Two partial orders over CTPs are defined in Section 3. These orders give a precise definition to what it means for a process to be an upper bound of another with respect to worst-case computation time, and what it means for a process to be easier to schedule than another.

In Section 4 we demonstrate that some CTPs exhibit counterintuitive behaviour, and that this behaviour may occur in all systems where an arbitrary SEQ/PAR structure is allowed. For instance, even if schedulability analysis of a system finds that all tasks meet their deadlines, this may no longer be true if a task executes less than its expected amount of computation. In Section 5 we identify a subset of *well-behaved* CTPs that never exhibits this kind of behaviour.

1. Background

The term "job" is used to denote a finite computation released at some instant, with a deadline at some later instant. The term "task" is used for a repeated job. A task could be sporadic, meaning that it is triggered by an external event, or periodic, if it is released at fixed intervals. In either case, tasks are specified with a minimum inter-release time, denoted T, and a relative deadline, denoted D. The computational requirement of each job from a task is denoted C.

The goal of schedulability analysis is to take a set of tasks, specified by periods, deadlines and computational requirements, and find out whether all tasks meet their deadlines when executed on a given platform. For multiprocessor schedulability analysis, two pieces of information are needed for each task x:

1. An upper bound on the amount of computation of other tasks that interferes with x
2. A lower bound on the amount of interference required for x to miss its deadline.

If the first bound is lower than the second, then x will not miss its deadline. For example, a complete schedulability test may be formulated as follows [1]: a task set is schedulable on a multiprocessor composed by m identical processors if and only if for each task k

$$\sum_{i \neq k} \min\left(\overline{I}_{i,k}, D_k - C_k + 1\right) \leq m\left(D_k - C_k + 1\right) \tag{1}$$

where $\overline{I}_{i,k}$ is the maximum amount of computation of task i that may delay task k. The left-hand side of this equation is the total interference on task k; the right hand side is the total amount of interference required for task k to miss its deadline. This analysis does not support job-level parallelism.

Schedulability analysis of systems with job-level parallelism is not well developed, and most analyses have fundamental restrictions that limit their applicability, for example by assuming that all jobs can be fully parallelised [2,3], or that the number of processors given to a job is fixed during the execution of the job [4,5]. The analysis given in [6] supports analysis of jobs with a *fork-join* structure, but does not discuss the situation where parallel branches themselves are allowed to fork.

Toc [7] is a programming language that extends occam with a construct to specify deadlines. A uniprocessor schedulability analysis for Toc programs is given in [8]. Toc programs tend to have a large amount of parallelism with an arbitrary structure, and this is not supported by existing multiprocessor schedulability analysis.

In this paper we introduce the computation time process model for analysing timing of parallel programs with an arbitrary parallel structure. In developing the model we borrow some terms and syntax from the process algebra CSP [9,10]. CSP models the computational behaviour of communicating processes, but does not explicitly model time. An extension of CSP that includes timing semantics is called Timed CSP [11].

For all discussions regarding computing and time it is important to note that it is generally not possible to determine the computation time of a program in advance. For example, the computation time may be data-dependent, or the program may contain recursions that are hard to analyse (e.g. the halting problem). However, if schedulability analysis is to be employed, a safe upper bound on the *worst-case execution time* (WCET) must be known. A WCET bound can be found using computerised tools, provided that the programmer follows certain conventions such as refraining from using dynamic memory or unbounded loops [12].

It is also important to note that because WCETs are always upper bounds, schedulability analysis should never deem a system schedulable, if requiring less than the expected amount of computation can cause it to miss deadlines. A schedulability analysis is called *sustainable* [13] if systems deemed to be schedulable by the analysis remain schedulable when reducing the amount of computation in the system, or relaxing the timing requirements. A schedulability analysis is called *exact* if it provides sufficient and necessary conditions for schedulability; and *pessimistic* if it may reject some systems that are actually schedulable.

2. Computation Time Processes

This section will introduce the CTP model.

2.1. Definitions

The set of computation time processes will be denoted $\mathbb{P}$. A process $P \in \mathbb{P}$ may be one of the primitive processes **0** or **1**, or a combination of two other processes using either the sequence operator ";", or the parallel operator "$||$":

The zero process $\mathbf{0}$ denotes a process which requires no work. It will never consume CPU, and any process following $\mathbf{0}$ in sequence after may start immediately.

The unit process $\mathbf{1}$ denotes a process which requires one unit of computation. The value of one unit of computation with respect to real time represents the minimum quantification of time for the system. It can for example be thought of as the time of one CPU cycle.

The sequence process $P\,;Q$, where $P, Q \in \mathbb{P}$, denotes a process of two sub-processes where one has to terminate execution before the other can begin. It satisfies the following basic laws:

$$\mathbf{0}\,;P = P \qquad \text{(left-identity)}$$

$$P\,;\mathbf{0} = P \qquad \text{(right-identity)}$$

$$(P\,;Q)\,;R = P\,;(Q\,;R) \qquad \text{(associativity)}$$

The parallel process $P \,||\, Q$, where $P, Q \in \mathbb{P}$, represents two processes that may execute interleaved. It satisfies the following basic laws:

$$\mathbf{0} \,||\, P = P \qquad \text{(identity element)}$$
$$P \,||\, Q = Q \,||\, P \qquad \text{(commutativity)}$$
$$(P \,||\, Q) \,||\, R = P \,||(Q \,||\, R) \qquad \text{(associativity)}$$

2.2. Properties of Computation Time Processes

An important property of any computation time process is its *total amount of computation*. This will be described as a function $\mathcal{C}\colon \mathbb{P} \to \mathbb{N}$, defined by

$$\begin{aligned} \mathcal{C}(\mathbf{1}) &= 1 \\ \mathcal{C}(\mathbf{0}) &= 0 \\ \mathcal{C}(P\,;Q) &= \mathcal{C}(P) + \mathcal{C}(Q) \\ \mathcal{C}(P \,||\, Q) &= \mathcal{C}(P) + \mathcal{C}(Q) \end{aligned}$$

For a process P, if the number of processors available to P is always greater or equal to the number of processors P can utilise, then it will be the longest sequence in P that determines its execution time. The *length* of a process, $\mathcal{L}\colon \mathbb{P} \to \mathbb{N}$, describes this time:

$$\begin{aligned} \mathcal{L}(\mathbf{1}) &= 1 \\ \mathcal{L}(\mathbf{0}) &= 0 \\ \mathcal{L}(P\,;Q) &= \mathcal{L}(P) + \mathcal{L}(Q) \\ \mathcal{L}(P \,||\, Q) &= \max\{\mathcal{L}(P), \mathcal{L}(Q)\} \end{aligned}$$

The length of a process is thus the minimum execution time of a process.

Note that in both the above definitions (of $\mathcal{C}$ and $\mathcal{L}$) we are not including any time for computational overheads in managing the parallel processes. These could be accounted for by adding a sequence of unit processes before and after the parallel composition, but are not addressed here for simplicity.

Another important property is the *immediate height* of the process, $\mathcal{H}\colon \mathbb{P} \to \mathbb{N}$, defined as the number of currently active parallel branches in the process:

$$\begin{aligned} \mathcal{H}(\mathbf{1}) &= 1 \\ \mathcal{H}(\mathbf{0}) &= 0 \\ \mathcal{H}(P\,;Q) &= \begin{cases} \mathcal{H}(P) & \text{if } P \neq \mathbf{0} \\ \mathcal{H}(Q) & \text{if } P = \mathbf{0} \end{cases} \\ \mathcal{H}(P \,||\, Q) &= \mathcal{H}(P) + \mathcal{H}(Q) \end{aligned}$$

The height of a process is therefore the maximum number of processors the process can utilise for the first step of its computation. We need this information when scheduling a process to the number of processors available for that first step.

2.3. Computing a Single Step

We model time as discrete, so processes are executed in unit time steps. After each execution step, the resulting process is a function of the original process and the number of processors assigned to it for that step. If a process consists of multiple parallel branches and there are too few processors available to execute them all, then the scheduler must choose a subset of branches to execute. Thus, if nothing is known about the details of the scheduler itself, the result of an execution step may be considered non-deterministic. The only detail of the scheduler that will be assumed is that it is *work conserving*, meaning that it does not keep any processors idle if there is ready work to be done.

The function $\mathrm{step}\colon \mathbb{P} \times \mathbb{N} \to \{\mathbb{P}\}$ takes a process P and a number of processors m and returns the set of all possible processes that can result from executing a single step of P with m processors. An explicit formulation of the step function can be written as

$$
\begin{aligned}
\mathrm{step}(\mathbf{1}, m) &= \begin{cases} \{\mathbf{1}\} & \text{if } m = 0 \\ \{\mathbf{0}\} & \text{if } m \geq 1 \end{cases} \\
\mathrm{step}(\mathbf{0}, m) &= \{\mathbf{0}\} \\
\mathrm{step}((P\,;Q), m) &= \begin{cases} \{(P'\,;Q)\colon P' \in \mathrm{step}(P, m)\} & \text{if } P \neq \mathbf{0} \\ \mathrm{step}(Q, m) & \text{if } P = \mathbf{0} \end{cases} \\
\mathrm{step}((P \,||\, Q), m) &= \big\{(P' \,||\, Q')\colon P' \in \mathrm{step}(P, m_P), Q' \in \mathrm{step}(Q, m_Q), \\
&\qquad m_P \in [0, \mathcal{H}(P)], m_Q \in [0, \mathcal{H}(Q)], m_P + m_Q = \min\{\mathcal{H}(P) + \mathcal{H}(Q), m\}\big\}
\end{aligned} \tag{2}
$$

where the last equation states that a scheduler can distribute available processors to the parallel branches P and Q in several ways, as long as the amount of execution performed by the step is limited either by the number of parallel branches, or by the number of available processors.

If the resulting set from a step of a process has only one member, then the execution of the process was deterministic; if the resulting set has more than one member, then the result of execution depends on choices made by the scheduler. Two examples of using the step-function are given below:

Example 1. Let a process P be given by

$$P = \mathbf{1} \,||\, \mathbf{1}$$

In this case, $\mathcal{H}(P) = 2$, because for its first step, P may utilise two processors. Say 3 processors are available for P for its first step, i.e. $m = 3$. Then,

$$
\begin{aligned}
\mathrm{step}((\mathbf{1} \,||\, \mathbf{1}), 3) &= \big\{(P' \,||\, Q')\colon P' \in \mathrm{step}(\mathbf{1}, m_P), Q' \in \mathrm{step}(\mathbf{1}, m_Q), \\
&\qquad m_P \in [0, 1], m_Q \in [0, 1], m_P + m_Q = \min\{2, 3\}\big\} \\
&= \big\{(P' \,||\, Q')\colon P' \in \mathrm{step}(\mathbf{1}, 1), Q' \in \mathrm{step}(\mathbf{1}, 1)\big\} \\
&= \big\{(P' \,||\, Q')\colon P' \in \{\mathbf{0}\}, Q' \in \{\mathbf{0}\}\big\} \\
&= \{\mathbf{0} \,||\, \mathbf{0}\} \\
&= \{\mathbf{0}\}
\end{aligned}
$$

which means that P will complete all its computation after one time step if 3 processors are available for it to use (P would also have completed if given 2 processors).

Example 2. Let a process P be given by

$$P = \mathbf{1} \,||(\mathbf{1}\,; \mathbf{1})$$

Say P is given one processor, i.e. $m = 1$. Then,

$$\begin{aligned}\mathrm{step}((\mathbf{1}\,||(\mathbf{1}\,;\mathbf{1})),1) = \big\{(P'\,||\,Q')\colon\, & P' \in \mathrm{step}(\mathbf{1}, m_P), Q' \in \mathrm{step}((\mathbf{1}\,;\mathbf{1}), m_Q),\\ & m_P \in [0,1], m_Q \in [0,1], m_P + m_Q = \min\{2,1\}\big\}\end{aligned}$$

which means that we either have $m_P = 1, m_Q = 0$ or $m_P = 0, m_Q = 1$. Taking the union of both alternatives results in

$$\begin{aligned}&= \big\{(P'\,||\,Q')\colon P' \in \mathrm{step}(\mathbf{1}, 1), Q' \in \mathrm{step}((\mathbf{1}\,;\mathbf{1}), 0)\big\}\\ &\qquad \bigcup \big\{(P'\,||\,Q')\colon P' \in \mathrm{step}(\mathbf{1}, 0), Q' \in \mathrm{step}((\mathbf{1}\,;\mathbf{1}), 1)\big\}\\ &= \big\{(P'\,||\,Q')\colon P' \in \{\mathbf{0}\}, Q' \in \{\mathbf{1}\,;\mathbf{1}\}\big\}\\ &\qquad \bigcup \big\{(P'\,||\,Q')\colon P' \in \{\mathbf{1}\}, Q' \in \{\mathbf{1}\}\big\}\\ &= \big\{\mathbf{0}\,||(\mathbf{1}\,;\mathbf{1})\big\} \bigcup \big\{\mathbf{1}\,||\,\mathbf{1}\big\}\\ &= \big\{(\mathbf{1}\,;\mathbf{1}), (\mathbf{1}\,||\,\mathbf{1})\big\}\end{aligned}$$

The result set has more than one element. Therefore, the result of the first step of P is not deterministic, but depends on which branch is assigned the one available processor.

2.4. Schedules

A *schedule* is a sequence of the number of processors available to be assigned at consecutive points in time, modelled as a finite sequence of non-negative integers. The set of all schedules is denoted $\mathbb{S}$. We will use bracket notation for sequences, e.g. $\langle 1, 2, 3\rangle$ for 1 followed by 2 followed by 3; and $\langle\rangle$ for the empty sequence. The concatenation of two sequences is written $s_1 \frown s_2$, e.g.:

$$\langle 1, 2, 3\rangle \frown \langle 4, 5, 6\rangle = \langle 1, 2, 3, 4, 5, 6\rangle$$

The results of executing a process P on a schedule s will be expressed by the operator $\otimes\colon \mathbb{P} \times \mathbb{S} \to \{\mathbb{P}\}$, defined by

$$\begin{aligned}P \otimes \langle\rangle &= \{P\}\\ P \otimes (\langle m\rangle \frown s) &= \bigcup_{P' \in \mathrm{step}(P,m)} P' \otimes s\end{aligned} \tag{3}$$

We say that a process P *will complete* on schedule s if it is guaranteed to complete, i.e.

$$P \otimes s = \{\mathbf{0}\} \tag{4}$$

This is distinct from *may complete*, as in

$$\mathbf{0} \in P \otimes s \tag{5}$$

Example 3. Let P be a process, defined by

$$P = (\mathbf{1}\,;\mathbf{1})\,||\,\mathbf{1}\,||\,\mathbf{1} \tag{6}$$

Looking at P, one can see that there are three possible $\mathbf{1}$-processes that can be executed at the first step. This is equivalent with the fact that $\mathcal{H}(P) = 3$. Also, no matter how many processors that are given to P it will never complete in less than two steps due to the sequence process in the left branch. This is equivalent with $\mathcal{L}(P) = 2$. The total number of $\mathbf{1}$-processes in P is four, so $\mathcal{C}(P) = 4$.

Consider the schedule $s = \langle 2, 3 \rangle$, meaning that P is given two processors for its first step, and three processors for its second step. Will P complete? Beginning with $\mathrm{step}(P, 2)$, we get

$$\mathrm{step}(P, 2) = \{(\mathbf{1} \,;\mathbf{1}), (\mathbf{1} \,||\, \mathbf{1})\}$$

For each process P' of the resulting set, we apply $\mathrm{step}(P', 3)$:

$$\mathrm{step}((\mathbf{1} \,;\mathbf{1}), 3) = \{\mathbf{1}\}$$
$$\mathrm{step}((\mathbf{1} \,||\, \mathbf{1}), 3) = \{\mathbf{0}\}$$

Therefore,

$$P \otimes s = \{\mathbf{0}, \mathbf{1}\}$$

so P may or may not complete on the schedule.

3. Partial Orders on CTPs

In order to analyse CTPs, some means of comparing processes must be introduced. In this section, two partial orders will be defined that allow comparison between processes with respect to different properties. The first is the *upper bound order*, which relates to worst-case execution time. A process Q is said to be an upper bound of P if P has the same structure as Q, but with possibly some of the computation removed. The second is the *schedulability order*, relating to the ease of scheduling a process. A process P is said to be easier to schedule than Q, if Q always completing in a schedule implies that P will also always complete.

3.1. Upper Bound Order ($\sqsupseteq$)

Execution time estimates are always upper bounds, so there is always the chance that an execution of a program turns out to require less computation than a CTP based on execution time analysis of the program. If a process P represents a possible execution of a process Q, then Q is said to be an upper bound for P. This will be written $P \sqsupseteq Q$.

Definition 1 ($\sqsupseteq$). Q is an upper bound of P, written $P \sqsupseteq Q$ if P can be derived from Q by replacing any number of unit processes in Q with the zero process.

It follows per definition that

$$\begin{aligned} &\mathbf{0} \sqsupseteq \mathbf{1} && \\ &P' \,; Q' \sqsupseteq P \,; Q && \text{if } P' \sqsupseteq P \wedge Q' \sqsupseteq Q \\ &P' \,||\, Q' \sqsupseteq P \,||\, Q && \text{if } P' \sqsupseteq P \wedge Q' \sqsupseteq Q \end{aligned}$$

The upper bound relation satisfies all properties of a partial order:

$$P \sqsupseteq P \qquad \text{(reflexive)}$$
$$P \sqsupseteq Q \wedge Q \sqsupseteq P \implies P = Q \qquad \text{(anti-symmetric)}$$
$$P \sqsupseteq Q \wedge Q \sqsupseteq R \implies P \sqsupseteq R \qquad \text{(transitive)}$$

The relation is not a total order, in that there exist pairs of processes where neither is an upper bound of the other, for example $\mathbf{1} \,; \mathbf{1}$ and $\mathbf{1} \,||\, \mathbf{1}$. An important property is that executing a process can be seen as a special case of removing computation:

$$\forall P \in \mathbb{P} \colon \forall s \in \mathbb{S} \colon (P' \in P \otimes s \implies P' \sqsupseteq P) \tag{7}$$

3.2. Schedulability Order ($\leq$)

The schedulability order relates two processes P and Q in such a way that if one process is guaranteed to complete on a schedule, the other is also guaranteed to complete on that schedule.

Definition 2 ($\leq$)**.** A process P is *easier to schedule* than a process Q, written $P \leq Q$, if for all schedules s,

$$Q \otimes s = \{\mathbf{0}\} \implies P \otimes s = \{\mathbf{0}\} \tag{8}$$

This order is essentially an order of worst-case execution time, as it relates the worst-case completion of P to the worst-case completion of Q. If $P \leq Q$, and Q is known to complete on some schedule, then Q can be replaced with P, and P will also complete on that schedule. Trivially, we have that

$$\mathbf{0} \leq \mathbf{1}$$

because $\mathbf{0}$ is guaranteed to complete on any schedule. The $\leq$-relation also satisfies all properties of a partial order:

$$P \leq P \qquad \text{(reflexive)}$$
$$P \leq Q \wedge Q \leq P \implies P = Q \qquad \text{(anti-symmetric)}$$
$$P \leq Q \wedge Q \leq R \implies P \leq R \qquad \text{(transitive)}$$

Like the $\sqsupseteq$-relation, the $\leq$-relation is also only a partial order. For example, if

$$P = \mathbf{1} \,||\, \mathbf{1} \,||\, \mathbf{1} \qquad s_1 = \langle 3 \rangle$$
$$Q = \mathbf{1} \,;\, \mathbf{1} \qquad s_2 = \langle 1, 1 \rangle$$

Then P and Q are incomparable with respect to schedulability. P will always complete for s_1 but never for s_2, while Q will always complete for s_2 but never for s_1.

Unlike the $\sqsupseteq$-relation, the $\leq$-relation does not in general distribute over ; and $||$, but it does distribute for the right element of a sequence, as maintained by the following lemma:

Lemma 1. *For all processes P and Q, and all processes Q' where $Q' \leq Q$,*

$$P \,;\, Q' \leq P \,;\, Q \tag{9}$$

Proof. Proof by contradiction. Assuming that Eq. 9 does not hold, then there must exist a schedule s so that $(P\,;Q) \otimes s = \{\mathbf{0}\}$ and $(P\,;Q') \otimes s \neq \{\mathbf{0}\}$. For all executions of $P\,;Q$ on this schedule there exists some prefix of s so that

$$s = s_P {}^\frown s_Q.$$
$$Q \in (P\,;Q) \otimes s_P$$

As $P\,;Q$ will always complete on s it follows that $Q \otimes s_Q = \{\mathbf{0}\}$. However $Q' \leq Q$ implies $Q' \otimes s_q = \{\mathbf{0}\}$, so $P\,;Q'$ must also always complete, contradicting the assumption that $P\,;Q'$ did not complete on the schedule. □

4. Timing Anomalies

Say a process Q has been derived from execution time analysis of some program. Because the WCET analysis always yields worst-case timings, an actual instance of the program does not necessarily behave like Q, but is only guaranteed to behave like some process P for which Q is an upper bound. Consider the following statement:

$$P \sqsupseteq Q \overset{?}{\Longrightarrow} P \leq Q \tag{10}$$

If the above statement was true, then an execution of the program would always have behaved like Q or like some process easier to schedule than Q: If Q was schedulable, then the program would always be schedulable. However, and somewhat surprisingly, the above statement does generally not hold. A special case of this, which may seem even more counterintuitive, is illustrated by the following observation:

Observation 1. There exist processes P and Q, and a schedule s so that

$$(P \in Q \otimes s) \wedge (P \nleq Q) \tag{11}$$

that is, there may be situations where Q is unable to complete on a schedule *if and only if it has completed some execution prior to beginning on the schedule*.

Example 4. An example will be given. Let P, Q and s be defined by

$$\begin{aligned} Q &= \big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\,||\,\big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big) \\ P &= \big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\,||\,\big(\mathbf{1}\,||\,\mathbf{1}\big) \\ s &= \langle 1 \rangle \end{aligned} \tag{12}$$

By choosing to execute the right parallel branch of Q, one can see that $P \in Q \otimes s$. To show $P \nleq Q$ it is sufficient to find a schedule for which Q is guaranteed to complete, but P is not. Let u be the schedule defined by

$$u = \langle 2, 4 \rangle$$

By computing $Q \otimes u$ we get that Q is guaranteed to complete on u:

$$\begin{gathered} \big(\big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\,||\,\big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\big) \otimes \langle 2 \rangle = \{\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}\} \\ (\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}) \otimes \langle 4 \rangle = \{\mathbf{0}\} \end{gathered}$$

By computing $P \otimes u$ we get that P is not guaranteed to complete on u. The first step yields

$$\big(\big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\,||\,\big(\mathbf{1}\,||\,\mathbf{1}\big)\big) \otimes \langle 2 \rangle = \big\{\big(\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}\big),\ \big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\big\}$$

Computing the second step for each of these results yields

$$\begin{gathered} (\mathbf{1}\,||\,\mathbf{1}\,||\,\mathbf{1}) \otimes \langle 4 \rangle = \{\mathbf{0}\} \\ \big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big) \otimes \langle 4 \rangle = \{\mathbf{1}\,||\,\mathbf{1}\} \end{gathered}$$

As Q will complete on u, but P may or may not, $P \nleq Q$.

Corollary 1. *There exist processes $Q \in \mathbb{P}$ and schedules $u, v \in \mathbb{S}$, where $\forall i \colon u_i \leq v_i$, and where*

$$(Q \otimes u = \{\mathbf{0}\}) \wedge (Q \otimes v \neq \{\mathbf{0}\}) \tag{13}$$

Proof. This can be shown setting $u = \langle 0, 2, 4 \rangle$ and $v = \langle 1, 2, 4 \rangle$ and using P and Q from Example 4. Then, $Q \otimes \langle 0 \rangle = \{Q\}$ and $Q \otimes \langle 1 \rangle = \{P\}$. The rest of the schedule is $\langle 2, 4 \rangle$, for which Q will always complete, but P may not. $\square$

Corollary 2. *There exist processes P and Q so that*

$$P \sqsupseteq Q \land P \not\leq Q \tag{14}$$

Proof. Executing a process will always result in a new process for which the original is an upper bound (see Eq. 7). The processes in Example 4 therefore satisfies $P \sqsupseteq Q$, and $P \not\leq Q$. □

For a process $P \sqsupseteq Q$ not to complete when Q is guaranteed to complete, it is necessary that the scheduler at some point makes a "wrong" decision. An argument may therefore be made that the counterintuitive behaviour of processes is in some part due to the scheduler; and moreover, that this behaviour can be eliminated by attempting to make a better scheduling algorithm. The following observation illustrates the futility of such an attempt.

Observation 2. There exists some process Q and schedule s so that the set $Q \otimes s$ has no least element in the schedulability order.

Example 5. An example will be given. Consider the schedule $s = \langle 1, 3\rangle$ and the process Q defined by

$$Q = \big(\mathbf{1}\,;(\mathbf{1} \,||\, \mathbf{1})\big) \,||\, \big(\mathbf{1}\,;\mathbf{1}\,;\mathbf{1}\big)$$

The first step of computation may result in one of the following two processes:

$$\begin{aligned} Q \otimes \langle 1\rangle = \big\{&\big(\mathbf{1} \,||\, \mathbf{1} \,||\, (\mathbf{1}\,;\mathbf{1}\,;\mathbf{1})\big), \\ &\big(\mathbf{1}\,;(\mathbf{1} \,||\, \mathbf{1})\big) \,||\, \big(\mathbf{1}\,;\mathbf{1}\big)\big\} \end{aligned}$$

Computing the second step for each of these processes yields

$$\begin{aligned} \big(\mathbf{1} \,||\, \mathbf{1} \,||(\mathbf{1}\,;\mathbf{1}\,;\mathbf{1})\big) \otimes \langle 3\rangle &= \{\mathbf{1}\,;\mathbf{1}\} \\ \big(\big(\mathbf{1}\,;(\mathbf{1} \,||\, \mathbf{1})\big) \,||\, \big(\mathbf{1}\,;\mathbf{1}\big)\big) \otimes \langle 3\rangle &= \{\mathbf{1} \,||\, \mathbf{1} \,||\, \mathbf{1}\} \end{aligned}$$

so

$$Q \otimes s = \big\{(\mathbf{1}\,;\mathbf{1}), (\mathbf{1} \,||\, \mathbf{1} \,||\, \mathbf{1})\big\} \tag{15}$$

The two resulting processes are incomparable with respect to schedulability, so $Q \otimes s$ has no least element.

Observation 2 implies that there is no such thing as a "correct choice" for a scheduler; whether or not a scheduling choice leads to the completion of a process may depend on the future schedule. The future schedule again depends on the behaviour of other tasks in the system, and is generally not predictable. For example, take the processes in Eq. 15. If the rest of the schedule is $\langle 1, 1\rangle$, then the first result will complete the process, but not the second. If a future schedule is $\langle 3\rangle$, then the second result completes the process, but not the first.

5. Well-behaved Processes

In the previous section it was demonstrated that there exist processes which are harder to schedule if computation is removed from them. Such a process cannot be used in schedulability analysis, because for real programs, only upper bounds on computation may be determined in advance. However, there exist processes that do not exhibit this kind of behaviour. These processes will be referred to as *well-behaved*, and will be the main topic of this section.

Definition 3 (Well-behaved)**.** A computation time process Q is *well-behaved* if and only if

$$\forall P \in \mathbb{P} \colon (P \sqsupseteq Q \implies P \leq Q) \tag{16}$$

Whether or not a process Q is well-behaved can be determined by exhaustive examination of all schedules in which Q is guaranteed to complete, and checking these schedules against all processes $P \sqsupseteq Q$ for which Q is an upper bound. However, such a test has at least exponential complexity with respect to $\mathcal{C}(Q)$, making it infeasible for most practical purposes. Instead, one may try to find general classes of processes where good behaviour is guaranteed by the process structure.

5.1. Classes of Well-Behaved Processes

Some simple processes are quickly found to be well-behaved:

Lemma 2. *The zero process* ($\mathbf{0}$) *and the unit process* ($\mathbf{1}$) *are well-behaved.*

Proof. The only process for which $\mathbf{0}$ is an upper bound is $\mathbf{0}$ itself, so it follows that $\mathbf{0}$ is well-behaved. The process $\mathbf{1}$ will complete on any schedule with at least one non-zero element. The only processes for which $\mathbf{1}$ is an upper bound is $\mathbf{0}$ and $\mathbf{1}$. Both will complete on any non-zero schedule, so it follows that $\mathbf{1}$ is well-behaved. □

Theorem 1. *Let P be a process with the following structure:*

$$P = P_1 \,;\, P_2 \,;\, P_3 \,;\, ... \,;\, P_n$$

If the processes $P_i|_{i=1...n}$ are well-behaved, then P is well-behaved.

Proof. Note that we only need to prove that $P_1 \,;\, P_2$ is well-behaved. If that is true, then

$$P = (P_1 \,;\, P_2) \,;\, P_3 \,;\, ... \,;\, P_n \tag{17}$$

will be a sequence of well-behaved processes for which the first element, $(P_1 \,;\, P_2)$, is well-behaved. The same proof can then be used to show that $((P_1 \,;\, P_2) \,;\, P_3)$ is well-behaved, and so on.

Let $s \in \mathbb{S}$ be any schedule for which $(P_1 \,;\, P_2) \otimes s = \{\mathbf{0}\}$. Let s_1 be the shortest prefix of s for which $P_1 \otimes s_1 = \{\mathbf{0}\}$ and let s_2 be the suffix of s so that $s = s_1 \frown s_2$. It follows that $P_2 \otimes s_2 = \{\mathbf{0}\}$, otherwise it would be possible to execute $(P_1 \,;\, P_2)$ on s without it completing. Let P_1' and P_2' be processes so that $P_1' \sqsupseteq P_1$ and $P_2' \sqsupseteq P_2$. Because P_1 is well-behaved, $P_1' \otimes s_1 = \{\mathbf{0}\}$.

P_1' must complete for s_1, but it may also complete earlier, leaving some rest of the schedule s_{rest}. It remains to show that all processes P_2' must complete within the schedule $s_{\text{rest}} \frown s_2$. As was noted in Eq. 7, execution of a process must lead to a process for which the original is an upper bound. Therefore, for all s_{rest},

$$\forall P_2'' \in P_2' \otimes s_{\text{rest}} \colon P_2'' \sqsupseteq P_2' \sqsupseteq P_2$$

The remaining schedule is s_2. It was already shown that $P_2 \otimes s_2 = \{\mathbf{0}\}$. As P_2 is well behaved, and $P_2'' \sqsupseteq P_2$, then P_2'' will also always complete on s_2. This shows that $P_1' \,;\, P_2' \leq P_1 \,;\, P_2$, so $P_1 \,;\, P_2$ is well-behaved. □

In general, $P \,||\, Q$ is not well-behaved, even if P and Q are well-behaved. For example, Q given in Eq. 12 is not well-behaved even though both of the parallel branches are well-behaved. A special case where a parallel process is indeed well-behaved is given below:

Theorem 2. *If $Q \in \mathbb{P}$ has the following structure*

$$Q = (\mathbf{1} \,;\, \mathbf{1} \,;\, \mathbf{1}...) \,||(\mathbf{1} \,;\, \mathbf{1} \,;\, \mathbf{1}...) \,||(\mathbf{1} \,;\, \mathbf{1} \,;\, \mathbf{1}...)... \tag{18}$$

where the branches need not be of the same length, then Q is well-behaved.

Proof. First note that all processes for which Q is an upper bound has the same structure as Q.

Let $Q = Q_1 \,||\, Q_2 \,||\, ...Q_N$ so that the Q_i processes are sequences of $\mathbf{1}$s. If some P satisfies $P \sqsupseteq Q$, then P must be equal to Q with some $\mathbf{1}$s removed, so if $P = P_1 \,||\, P_2 \,||\, ...P_N$, then we have

$$\forall i \in [1, N] \colon \mathcal{L}(P_i) \leq \mathcal{L}(Q_i) \tag{19}$$

We also have $\mathcal{C}(P) \leq \mathcal{C}(Q)$ and $\mathcal{H}(P) \leq \mathcal{H}(Q)$. Let m be the first element in a schedule. If

$$m \leq \mathcal{H}(P) \;\vee\; m \geq \mathcal{H}(Q)$$

then every choice that the scheduler can make for P it can also make for Q. For all choices of P, a corresponding choice can be made for Q so that Eq. 19 is still satisfied. If

$$\mathcal{H}(P) \leq m \leq \mathcal{H}(Q)$$

then the scheduler may choose to execute additional branches for Q. However, these branches must already be of zero length for P, so for all results for P, a corresponding result for Q can be found that satisfies Eq. 19. This can be repeated for each element in the schedule.

This shows that for all schedules s, and all processes $P' \in P \otimes s$ there exists some process $Q' \in Q \otimes s$ so that $\mathcal{C}(P') \leq \mathcal{C}(Q')$. A consequence is that we cannot have $Q \otimes s = \{\mathbf{0}\}$ when $P \otimes s \neq \{\mathbf{0}\}$. Therefore,

$$Q \otimes s = \{\mathbf{0}\} \implies P \otimes s = \{\mathbf{0}\}$$

P is easier to schedule than Q, which implies that Q is well-behaved. □

5.2. *Safe Upper Bound*

If schedulability analysis is to be performed on some ill-behaved process P, then the analysis will not be sustainable. To make the analysis sustainable, it would be better to replace P with some well-behaved process Q that is harder to schedule than all processes $P' \sqsupseteq P$, and then analyse Q instead. Such a Q will be referred to as a *safe upper bound*.

Definition 4 (Safe Upper Bound)**.** A process Q is a safe upper bound for a process P if

$$\forall P' \sqsupseteq P \colon P' \leq Q \tag{20}$$

The existence of a safe upper bound is guaranteed by the following lemma:

Lemma 3. *For all $P \in \mathbb{P}$, if Q is a sequence of $\mathbf{1}$s with length $\mathcal{C}(P)$, then Q is a safe upper bound for P.*

Proof. Q will complete for all schedules that have at least $\mathcal{C}(P)$ non-zero elements. P will also complete for all these schedules, so $P \leq Q$. Furthermore, all processes P that satisfy $P' \sqsupseteq P$ will also complete for this schedule, so Q is a safe upper bound. □

A process P can be replaced by a safe upper bound to make temporal analysis of P sustainable. However, the analysis will no longer be exact, as the safe upper bound may be harder to schedule than P. For example, the choice of safe upper bound used in Lemma 3 suppresses all parallelism of a process, which could make the analysis unnecessarily pessimistic. Sometimes, other safe upper bounds exist that maintain some of this parallelism and thus lead to less pessimistic results:

Example 6. Take the ill-behaved process Q given below:

$$Q = \big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)\,||\,\big(\mathbf{1}\,;(\mathbf{1}\,||\,\mathbf{1})\big)$$

Let Q_S be defined by

$$Q_S = \mathbf{1}\,;\mathbf{1}\,;\mathbf{1}\,;\mathbf{1}\,;\mathbf{1}\,;\mathbf{1}$$

According to Lemma 3, Q_S is a safe upper bound for Q. Let Q_P be defined by

$$Q_P = (\mathbf{1}\,||\,\mathbf{1})\,;(\mathbf{1}\,||\,\mathbf{1})\,;(\mathbf{1}\,||\,\mathbf{1}) \tag{21}$$

Q_P will complete on some schedule s if and only if $s = b_1 \frown b_2 \frown b_3$ where each $b_i|_{i=1..3}$ is either $\langle 1, 1\rangle$ or $\langle 2\rangle$, or similar schedules with strictly larger elements. For example, Q_P will complete on $\langle 2, 1, 1, 2\rangle$, but not on $\langle 1, 2, 1, 2\rangle$. It can be found by systematic examination that Q will complete on all these schedules, and that all $Q' \sqsupseteq Q$ also will complete on these schedules. We know from Theorem 1 that Q is well-behaved. It follows that Q_P is also a safe upper bound of Q. Moreover, as $Q_P \leq Q_S$ there are no schedules for which Q_S will complete and Q_P will not. Q_P is therefore a better choice of safe upper bound.

For a process P, the best choice of safe upper bound would be the least element in the set of safe upper bounds with respect to schedulability:

$$Q^\star = \min_{\leq}\,\{Q \in \mathbb{P} \colon \forall P' \sqsupseteq P \colon\, P' \leq Q\} \tag{22}$$

At this point we do not know of a method to find the best safe upper bound, or if a unique best safe upper bound generally exists.

6. Conclusions and Future Work

In this paper, we introduced the computation time process model as a tool for temporal analysis of non-communicating programs with an arbitrary parallel structure. The simplicity of dealing only with time, rather than with time and computation, allows the temporal properties of simple processes to be examined more easily than would be possible with for example Timed CSP. Moreover, the CTP model explicitly models timing of programs on multiprocessor systems, not timing of abstract processes in general.

Somewhat counterintuitively it was shown that an upper bound on the estimate of computation does not in general represent a worst-case scenario with respect to schedulability; there exist processes that are unable to complete on schedule only if requiring less than their upper bound. The existence of these processes has important implications for temporal analysis of all programs with an arbitrary SEQ/PAR structure, because it implies that exact schedulability analysis will in general not be sustainable.

The processes where worst-case execution did represent a worst-case scenario were labelled well-behaved. If a process has an upper bound that is well-behaved, then this is called a safe upper bound. By replacing an ill-behaved process with a safe upper bound, one can enable sustainable schedulability analysis, but the analysis will no longer be exact. Some safe upper bounds were found to be easier to schedule than other safe upper bounds, and would thus result in less pessimistic analysis. We do not know of a method to construct the best safe upper bound, nor if a best safe upper bound generally exists. This should be investigated as part of future work.

It was also shown that there cannot exist a perfect scheduler for SEQ/PAR programs, because there are examples of schedules where the decision that leads to the completion of a process depends on the future schedule. This was shown using an ill-behaved process.

Whether or not there exists a perfect scheduling choice when processes are well-behaved has not yet been determined.

The CTP model should be extended to handle communication. Committed synchronous communication may be modelled with sequences: a parallel first part (where the processes are not communicating); then the communication; then a parallel second part where the processes go their own ways. However, alternation would require additions to the existing model. Additions are also needed to model conditional computation; the existing model can only model conditional computation for the special case where the branches have the same parallel structures.

Another topic for future work is to study the interaction between CTPs executing on the same platform. For a schedulability analysis of a complete system, bounds on the amount of computation from one CTP that may interfere with another is required, as well as a function from interference to delay so that it can be determined whether or not a task based on a CTP misses a deadline. The results in this paper should prove useful in developing such an analysis.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and constructive criticism, which has helped to improve the presentation of this paper.

References

[1] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 553–566, April 2009.

[2] C.-C. Han and K.-J. Lin, "Scheduling parallelizable jobs on multiprocessors.," in *IEEE Real-Time Systems Symposium'89*, pp. 59–67, 1989.

[3] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, pp. 180–187, May 2008.

[4] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," in *18th International Conference on Real-Time and Network Systems*, 2010.

[5] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," (Los Alamitos, CA, USA), pp. 459–468, IEEE Computer Society, 2009.

[6] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," *Real-Time Systems Symposium, IEEE International*, pp. 259–268, 2010.

[7] M. Korsgaard and S. Hendseth, "Combining EDF scheduling with occam using the Toc programming language," in *Communicating Process Architectures 2008* (A. A. McEwan, W. Ifill, and P. H. Welch, eds.), pp. 339–348, IOS Press, sept 2008.

[8] M. Korsgaard and S. Hendseth, "Design patterns for communicating systems with deadline propagation," in *Communicating Process Architectures 2009* (P. H. W. et. al, ed.), pp. 349–361, IOS Press, november 2009.

[9] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, 1978.

[10] A. W. Roscoe, *The Theory and Practice of Concurrency*. Hertfordshire, England: Prentice Hall Europe, 1998.

[11] S. Schneider, *Concurrent and Real Time Systems: The CSP Approach*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[12] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[13] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, (Washington, DC, USA), pp. 159–168, IEEE Computer Society, 2006.