A Uniform Model for Tolerance-Based Real-Time Computing *

amended version, 3.9.2014

Raimund Kirner
University of Hertfordshire
United Kingdom
r.kirner@herts.ac.uk

Abstract

Standard real-time models do not consider the fact that a chosen technical deadline is different from the critical latency where the service utility becomes zero. This is in mismatch with engineering practice.

In this paper we propose a tolerance-based refinement of the real-time model. By doing so we make the process of deriving the estimation of the critical latency explicit. The difference between the technical deadline and the critical latency is a measure for the safety margin of the system. This safety margin is important for both, soft real-time and hard real-time systems, though with different quantities and qualities. Furthermore, we explain why the critical latency can hardly be quantified by a concrete value. However, we demonstrate how to derive reasonable estimates for it. We use a concrete application to show how the distinctive knowledge of the critical latency and the technical deadline are useful for real-time scheduling.

1 Introduction

The central timing metrics to specify real-time requirements is the deadline. The deadline is the time interval from an input stimuli until when the output has to be provided [8, 12]. The deadline limits the latency of a real-time system. Instead of a simple deadline, one might specify a function of the output utility depending on the latency [7, 1]. Besides the latency, also the throughput and the jitter of a real-time output can be of relevance.

In standard real-time models the deadline serves as the specification of the critical latency imposed by the environment as well as for the maximum latency by which a real-time computer system has to provide its output. This concept is actually in mismatch with engineering practice, where a technical deadline is chosen conservatively and explicitly different to the critical latency. The critical latency itself often cannot be bounded precisely, as it may highly depend on the environmental context of the real-time computer system. However, using a conservative estimate of the critical latency is sufficient for the purpose to show the safety margin respectively quality of control one has allowed in the system design.

Section 2 reviews different existing real-time computing models.

The contribution of this paper is to introduce a tolerance-based refinement of the real-time computing model, which is described in Section 3. This model makes an explicit distinction between the chosen technical performance metrics and the critical performance metrics where the output utility becomes zero. Besides the primary performance metrics a tolerance performance metrics can be specified as well. Section 4 describes how a concrete scheduling problem can be derived from the tolerance-based real-time model. Section 5 shows how the system utility can be used to optimise the system schedule. Section 6 shows on a realistic example how tolerance-based scheduling can be used to optimise hardware costs. This is just one possible use case, others, for example, include smart fault-tolerance for systems with requirements of mixed criticality [11, 5, 2]. Fault-tolerant scheduling by using a tolerance range is a different concept than soft real-time computing, as the tolerance ranges allow for a systematic choice of service degradation. Section 7 concludes the paper.

2 Real-time Computing Models

In this section we review the very basic concept of how real-time systems are specified. We further motivate a refinement towards tolerance-based specification of real-time
systems.

2.1 Deadline-based RT Model

For the correctness of real-time systems, the timeliness of computed actions plays an important role:

* A real-time computer system is a computer system where the correctness of the system depends not only on the logical results of the computations, but also on the physical time when these results are produced.* (Kopetz [8], page 2)

The most fundamental description of timing requirements of a real-time systems is the specification of latency bounds, the so-called deadline:

* A real-time computer system must react to stimuli from its environment (the controlled cluster or the operator cluster) within time intervals dictated by its environment. The instant when a result must be produced is called a deadline. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If severe consequences could result if a firm deadline is missed, the deadline is called hard.* (Kopetz [8], page 3)

Above definition of a deadline also takes into account the criticality of the real-time service in order to judge the utility of a result at different times.

Independently of the concrete chosen deadline-based specification of real-time systems, we see an important refinement potential to make the specification closer to engineering practice. The basic limitation of the pure deadline-based specification of real-time systems is that (implicitly) the mentioned deadline is used for two purposes:

1. the deadline determines the utility of the output (we call this critical latency), and
2. the deadline is used as latency constraint for the design of the real-time system (we call this technical deadline).

Conceptionally, in the pure deadline-based real-time specification the principally independent latencies called critical latency and technical deadline are assumed to be equal. The critical latency defines the utility of an output imposed by the real-time environment. The technical deadline denotes the maximum latency the engineers have chosen to allow for the response time of the real-time controller. To quantify the critical latency can be non-trivial as it can depend very much on the concrete context. However, in practice engineers have to estimate the critical latency in order to define a reasonable technical deadline. The chosen technical deadline has to be sufficiently smaller than the estimated critical latency in order to allow for a sufficiently large safety margin.

Therefore, the pure deadline-based real-time specification is mere a simplification to teach the real-time concepts, but at the same time hides an important aspect of engineering practice.

2.2 Time/Utility-based RT Model

There have been approaches to extend the pure deadline-based real-time specification by a generic function. For example, the so-called time/utility function evaluates the utility of the output for any latency [7]. As shown in Figure 1, it is possible with such a generic time/utility function to express negative utility, i.e., potential damage in case of deadline violations.

![Figure 1. Time/Utility-based Real-Time Model](image)

*Figure 1. Time/Utility-based Real-Time Model*

While the time/utility function adds expressiveness to the specification of the characteristics of a timing requirement, it does by itself not change one inherent limitation of the pure deadline-based real-time specification: the critical latency and the chosen technical deadline are not explicitly specified.

3 Tolerance-based Real-Time Model

In the following we introduce a tolerance-based refinement of the real-time model. This tolerance-based real-time model closes the gap between real-time modelling and engineering practice. Furthermore, it provides the foundation for new methods of real-time scheduling.

A real-time computer system typically has to provide different functionalities, which each of them having its own real-time requirements. In order to distinguish these different functionalities, we call them services.

To clarify the meaning of a service, Figure 2 shows the relation between the terms service, interface, and system. The interface describes the system access by input and output. An interface might be specified at different levels of detail, for example, by an API, or by some abstract behaviour rules. A system is said to realise an interface. A service is a slice of the total system behaviour, with a subset of the system’s interface used to access that service. Services stretch
from the input to the output of a system, which is in con-
trast to a system component, which is a subsystem on its
own, also realising its own interfaces and services. It is im-
portant to note that none of the concepts of Figure 2 are a
synonym for the scheduling objects to be executed at run-
time, as services may result in more than one scheduling
object and services may also share some of their scheduling
objects.

\[\text{Figure 2. UML Relation between System and Services}\]

3.1 Performance Characteristics

System services are normally connected with require-
ments of their timing properties. Important timing prop-
erties of a service can be throughput, latency, and jitter.
The throughput describes how many input elements a
system is able to process per time. The latency describes
the delay between arrival of data at the system input and the
release of the processed results at the system output. The
jitter describes the variation of the latency.

These three performance metrics are the basic perfor-
ance characteristics of a real-time system. For real-time
services it is important to consider the boundaries of these
performance metrics. Foremost the minimum throughput,
the maximum latency, and the maximum jitter are impor-
tant to ensure the timeliness of a real-time service. It is
worth noting that by binding the maximum latency and jitter, we implicitly also bind the minimum latency.

3.2 Temporal Service Utility

As described in Section 2.2, real-time models had been
developed that specify the utility of a real-time system as a
function over latency. We generalise that concept by apply-
ing such utility functions not only for latency, but also for
the other performance metrics, namely throughput and jitter. However, we exemplify the concepts of utility functions primarily for latency, as they are analogous for throughput and jitter.

As mentioned in Section 2, the classical utility-based
real-time model as exemplified in Figure 1 does not really
reflect reality. Because in reality no one should chose a
deadline such that after the deadline the utility of the re-
result is immediately zero. Doing so would leave no safety
margin, and furthermore, the relative distance from the cho-
sen deadline and the precise point where the utility of the
result could become zero is generally not known.

We propose a refined real-time model that acknowledges
the difference of the latency where the resulting utility could
become zero and the latency chosen as technical deadline.
We describe the timing requirements of a real-time service
by the utility of the computed result over the performance
metrics. We call the utility functions specifically to their
performance metrics as:

- service latency utility function (SLUF),
- service throughput utility function (STUF), and
- service jitter utility function (SJUF)

\[\text{Figure 3. Service Latency Utility Function (SLUF)
of a Real-Time Service}\]

Figure 3 shows our real-time specification exemplary for latency. Compared to the classical time/utility function as shown in Figure 1, we make a distinction between what is considered the critical latency with zero utility and the latency we choose as a technical deadline (latency limit) for the realisation of a concrete real-time computer system. In the given example a service provided later than a certain time interval, denoted as critical latency, can provide system damage. This range of service latency is marked as damaging failure in Figure 3. The range of service latency marked as normal safe operation denotes the service latency considered as optimal operation, providing the maximum and constant service utility. There is also a latency range denoted as impaired safety margin, which does not yet cause any system damage, but exhibits a declining utility. The de-
clining utility is a reflection of the reduced safety margin of
the corresponding latency interval. When designing a sys-
tem we usually set the deadline of a service to the end of
the uncompromised service utility range, which is labelled as *technical deadline* in the figure.

In the example the utility drops abruptly to full potential damage after the *critical latency*. However, in general a latency transition from *impaired safety margin* to *damaging failure* could also be more smooth, depending on the application. For example, with fuel injection in a motor there is a certain range of injection time where the motor becomes less efficient and is going to be increasingly worn out with later/earlier injection timing. In general, the transition from *normal safe operation* to *damaging failure* can be quite manifold, depending on the concrete real-time service. Research on real-time computing is most often oblivious of this diversity of transition characteristics of different real-time services.

In Section 3.2 we made the case that it is important to distinguish between the chosen technical deadline and the critical latency. The same holds for throughput and jitter. It might be that there is a relatively large gap between the chosen technical performance metrics and the critical performance metrics. For such cases we can refine this gap towards more flexibility to allow to handle extraordinary situations of unexpected resource shortness.

We introduce an additional tolerance performance metrics besides the primary performance metrics. For example, in case of latency, the chosen tolerance latency $l_{tol}$ (i.e., tolerance technical deadline) is positioned between the chosen primary latency $l_{prim}$ (i.e., primary technical deadline) and the critical latency $l_{crit}$, as expressed by the chosen primary latency $l_{prim}$ (i.e., primary technical deadline) and the critical latency $l_{crit}$, as expressed by Equation 1.

$$
(l_{prim} < l_{crit}) \Rightarrow (l_{prim} < l_{tol} < l_{crit})
$$

Equation 1 considers both, upper latency limits as well as lower latency limits. In either case the utility of $l_{tol}$ will be lower than that of $l_{prim}$:

$$
SLUF(l_{prim}) > SLUF(l_{tol}) > SLUF(l_{crit})
$$

The formulas for a tolerance throughput $tp_{tol}$ or a tolerance jitter $j_{tol}$ are analogous to Equation 1 and 2 using the service utility functions $STUF(t)$ and $SJUF(j)$.

It is assumed that any tolerance performance metrics ($SLUF(l_{tol})$, $STUF(t_{tol})$, or $SJUF(j_{tol})$) provides a utility level that is still acceptable in case of the considered emergency conditions. The considered emergency conditions are meant to be those for which a tolerance mechanism is required.

3.3 Primary Limits and Tolerance Ranges

The gap between the primary performance metrics and the tolerance performance metrics is called *tolerance range*. Figure 5 shows an example of a tolerance range for the latency of a real-time service. In this example the latency of the real-time service has only an upper critical latency $l_{crit}$, but no lower critical latency, i.e., any response time between 0 and $l_{crit}$ has a positive utility.
### 3.4 Running Example: Video Processing System

We use video processing as an example of how to specify service utility functions of a system. We base this example on a simplified version of the X-ray treatment with image processing [10] published by Schrijver and Creemers from Philips Healthcare. In contrast to the Philips use case, we don’t use a background task. But as in the Philips use case, we assume that the images of the video stream are to be processed by two different operations, which can run in parallel for the same video frame. Furthermore, the Philips use case has different criticality levels for these operations. To keep it simple, we assume equal criticality levels, even though our tolerance-based scheduling optimisation allows to consider multiple criticality levels.

Figure 6 shows the basic dataflow of our video processing use case. The video frames are read from the input and are forwarded into two video processing operations, denoted ImageFilter and FeatureExtract. These two operations are independent, i.e., they can be executed concurrently. Once both operations have produced their results, they are merged together as a single video frame. The execution cost of the split and the join operation in the dataflow diagram are negligible compared to the execution cost of the two operations. Thus, we assume that they are incorporated into the execution cost of the operations. This assumption will keep the discussion of the scheduling problem shorter.

![Figure 6. Use Case: Video Processing Application](image)

The two operations are called task A and task B, assuming a non-concurrent implementation of each of these operations. Thus they can be used directly as scheduling objects. The instantiation of a task at runtime is called a job. This distinction is important as depending on the concrete scheduling problem, there can exist at the same time multiple instances as instance of the same task.

We focus on throughput and latency in our use case. Jitter is of less importance and thus ignored. For the throughput we assume $tp_{prim}$ is 25 frames/s, which is sufficient to achieve a very smooth visualisation of movements in the image. On the other hand $tp_{crit}$ is assumed to be 11 frames/s, as such a slow update can be already too slow for following precise movements on the video.

The latency of the video processing system $l_{prim}$ is assumed to be 80 ms or less for an optimal visual feedback. At the same time $l_{crit}$ is assumed to be 200 ms, as with longer delays the video processing system cannot be used for certain feedback where precise movement is necessary. Given that assumption, we can approximate the STUF and SLUF of the video processing system, as given in Figure 7 and Figure 8. The dotted line indicates the detailed utility function, while the bold line we use as a sufficient approximation.

![Figure 7. Specification of the STUF of the Video Processing System](image)

![Figure 8. Specification of the SLUF of the Video Processing System](image)

The tolerance latency $l_{tol}$ and $tp_{tol}$ have been chosen such that their SLUF and STUF are 50% of the maximum. This threshold of 50% is an arbitrary chosen value, and would need the consideration of the concrete application domain in order to justify it.

### 4 Derivation of Scheduling Problems

As mentioned in Section 3.1, the performance characteristics of a real-time as well as non-real-time service are throughput, latency, and jitter. These metrics are used to describe the timing requirements from an application point of view.

However, the objects used for scheduling might be described by different parameters. Actually, what these parameters are, depends completely on the concrete scheduling method. Mapping the three performance metrics to these scheduling parameters is - besides the scheduling method - also specific to the partitioning of the application code into scheduling objects, so-called tasks. As mentioned
before, tasks are the static scheduling objects, which are instantiated at runtime as jobs. More than one job instantiated from the same task can exist as the same time.

There exist a vast number of different real-time scheduling methods, for example, Davis and Burns wrote a survey on different hard real-time scheduling methods for multi-processor systems [4]. To keep it simple for the purpose of demonstration, we focus in the following on a static scheduling with a fixed instantiation pattern of jobs, as it is the case in time-triggered scheduling [8]. We can specify tasks \( t_i \) in our scheduling model by their worst-case execution time (WCET) \( e_{t_i} \), their period \( p_i \), their relative deadline \( d_i \), and the relative starting offset \( o_{f_{i,j}} \) in case more that one job chain \( j \) is to be instantiated at runtime. In our scheduling model it is also relevant whether we use multi-core processors. This is because the scheduling order has an impact on the resulting latency, etc.

To give an example of how local scheduling patterns are significant, we refer again to the two tasks \( t_A \) and \( t_B \) from our running example described in Section 3.4. We consider two different patterns of periodic scheduling for the tasks \( t_A \) and \( t_B \):

- \( sched_1 \): \( t_A, t_B \) are executed in sequence
- \( sched_2 \): \( t_A, t_B \) are executed in parallel

Which one of these pattern to chose, depends on the concrete context. For example, if \( e_{t_A} = e_{t_B} \) then scheduling pattern \( sched_2 \) is generally preferred in multi-processor systems, as it results in a lower latency. However, if \( e_{t_A} <> e_{t_B} \) then scheduling pattern \( sched_1 \) can be better on multi-processor systems, as it results in a maximum throughput.

We assume a dual-core system, which requires us for scheduling pattern \( sched_1 \) to instantiate two periodic chains for each tasks \( t_A \) and \( t_B \).

In case of scheduling pattern \( sched_1 \) we get the following specific relations between the performance metrics and the scheduling parameters:

\[
\begin{align*}
\text{latency } l &= e_{t_A} + e_{t_B} \quad \text{(3)} \\
\text{throughput } tp &= \frac{1}{l} \quad \text{(4)} \\
off_{A,1} &= o_{f_{B,1}} = 0 \quad \text{(5)} \\
off_{A,2} &= o_{f_{B,2}} = \frac{e_{t_A} + e_{t_B}}{2} \quad \text{(6)}
\end{align*}
\]

Equation 3 follows as in \( sched_1 \) we execute \( t_A \) and \( t_B \) in sequence. In this pattern we execute two video frames partially overlapping, thus the throughput is twice the reciprocal value of the latency, as denoted in Equation 4. Equation 5 and 6 express the overlapping instantiation of the two periodic chains. Scheduling pattern \( sched_1 \) is visualised in Figure 9. For \( sched_1 \) we also get the following relations between task periods, task deadlines, and latency:

\[
p_A = p_B = \frac{d_A}{2} = \frac{d_B}{2} = \frac{l}{2}
\]

\[
\begin{array}{c}
d_A, d_B, l \\
p_A, p_B \\
Core_1 \quad \begin{array}{c} A_1 \\
\end{array} \begin{array}{c} B_1 \\
\end{array} \\
\hline \\
Core_2 \quad \begin{array}{c} A_2 \\
\end{array} \begin{array}{c} B_2 \\
\end{array}
\end{array}
\]

**Figure 9. Scheduling Pattern \( sched_1 \)**

In case of scheduling pattern \( sched_2 \) we get the following specific relations between the performance metrics and the scheduling parameters:

\[
\begin{align*}
\text{latency } l &= \max(e_{t_A}, e_{t_B}) \quad \text{(8)} \\
\text{throughput } tp &= \frac{1}{l} \quad \text{(9)} \\
off_{A} &= o_{f_B} = 0 \quad \text{(10)}
\end{align*}
\]

Equation 8 follows from the fact that \( t_A \) and \( t_B \) are executed in parallel in \( sched_2 \). Since there is now only one video frame processed at a time, Equation 9 follows. Equation 10 expresses that we instantiate only one periodic chain per task. Scheduling pattern \( sched_1 \) is visualised in Figure 10. For \( sched_2 \) we also get the following relations between task periods, task deadlines, and latency:

\[
p_A = p_B = d_A = d_B = l
\]

\[
\begin{array}{c}
l, d_A, d_B \\
p_A, p_B \\
Core_1 \quad \begin{array}{c} A \\
\end{array} \begin{array}{c} A \\
\end{array} \\
\hline \\
Core_2 \quad \begin{array}{c} B \\
\end{array} \begin{array}{c} B \\
\end{array}
\end{array}
\]

**Figure 10. Scheduling Pattern \( sched_2 \)**

From Figure 9 and Figure 10 we also see that \( sched_1 \) has better throughput, while \( sched_2 \) has better latency.

5 Tolerance-based Scheduling

In the following we describe how the tolerance-interval can be used to optimise the scheduling. According to our running example described in Section 3.4 we focus only on throughput and latency.
An optimal schedule according to the overall utility \(SU_F_{opt, rts}\) of all system services can be found by maximising the following goal function:

\[
SU_F_{opt, rts} = \max \sum_{i \in Services} SLUF(i, l_i) \cdot STUF(i, tp_i)
\]  \hspace{1cm} (12)

In the goal function given in Equation 12 the total utility for each service \(i \in Services\) is calculated by multiplying the individual utility functions for the performance metrics of interest. The maximisation of above goal function is subject to the following constraints:

\[
\forall i \in Services, l_i \leq l_{tol, i}
\]  \hspace{1cm} (13)

\[
\forall i \in Services, tp_i \geq tp_{tol, i}
\]  \hspace{1cm} (14)

\[
\forall i \in Services \land \forall sp \in SchedPattern, \text{scheduling}\_\text{constraints}(i, sp)
\]  \hspace{1cm} (15)

Equation 13 and Equation 14 simply limit the latency and throughput of each service to be within the tolerance range.

Equation 15 is used to denote a predicate that subsumes all the constraints used to link the performance metrics with the scheduling parameters as described in Section 4. For our running example we have the following set of scheduling choices: \(SchedPattern = \{sched_1, sched_2\}\).

It is worth noting that the goal function of Equation 12 does treat all services with the same weight, i.e., they are services of the same criticality. However, in application domains with safety-critical functions it is necessary to differentiate between the different criticality levels of services. For example, the DO-178B standard of the civil avionics domain defines five Design Assurance Levels (DAL) [9], the ISO 26262 standard of the automotive domain defines four Automotive Safety Integrity Levels (ASIL) [6], and the IEC 61508 standard of the automation domain defines four Safety Integrity Levels (SIL) [3].

Our tolerance-based utility optimisation can be extended to mixed-criticality systems by replacing the goal function of Equation 12 by the following goal function, which specifies an additional criticality parameter \(CRIT(i)\) for each service \(i\):

\[
SU_F_{opt, mcrt} = \max \sum_{i \in Services} CRIT(i) \cdot SLUF(i, l_i) \cdot STUF(i, tp_i)
\]  \hspace{1cm} (16)

In this section we have shown how to optimise system utility by using tolerance ranges. In the next section we will illustrate exemplary for what this can be used.

6 Evaluation

To demonstrate the use of the tolerance-based real-time model, we assume the following scenario: A company has developed a system to be runnable on a platform \(HW_1\). Close to the mass production they realised that platform \(HW_1\) is too expensive to get a competitive product. So they want to investigate whether two less powerful but cheaper platforms \(HW_2\) and \(HW_3\) are able to provide a still sufficient good enough quality of service, even potentially below the originally envisioned quality level. The properties of all available platforms are summarised in Table 1. \(HW_3\) is the only platform with a single core, thus only scheduling pattern \(sched_1\) is applicable to \(HW_3\). With respect to peak speed, \(HW_3\) is faster than \(HW_2\), but slower than \(HW_1\).

<table>
<thead>
<tr>
<th>Platform</th>
<th>#cores</th>
<th>relative frequency</th>
<th>peak speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>(HW_1)</td>
<td>2</td>
<td>100%</td>
<td>1.00</td>
</tr>
<tr>
<td>(HW_2)</td>
<td>2</td>
<td>66%</td>
<td>0.66</td>
</tr>
<tr>
<td>(HW_3)</td>
<td>1</td>
<td>150%</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 1. Set of Available Platforms

We again use our running example of the video processing system described in Section 3.4. Using \(HW_1\) as the reference system, we assume the execution times of task \(t_A\) and \(t_B\) as given in the first two columns of Table 2. Each row in the table is just a different numeric example with a different computational load on \(HW_1\). The remaining columns of Table 2 show the resulting system utility according the method described in Section 5 for the different platforms \(HW_1\), \(HW_2\), \(HW_3\) and the different scheduling patterns \(sched_1\), \(sched_2\). The columns \(r_{Thp}\) show for each scenario the achieved throughput compared to \(tp_{prim}\):

\[
r_{Thp} = \frac{tp}{tp_{prim}}
\]

However, given the STUF in Figure 7, a better throughput than \(tp_{prim}\) does not lead to an improvement of the STUF of that service.

Analogously, the columns \(r_{Lat}\) show for each scenario the achieved latency compared to \(l_{prim}\):

\[
r_{Lat} = \frac{l_{prim}}{l}
\]

Again, given the SLUF in Figure 8, a better latency than \(l_{prim}\) does not lead to an improvement of the SLUF of that service. \(^1\)

Table 2 finally provides several interesting results. For

\(^1\)In case the paper is available as colour print, then each case where the latency or throughput is better than the corresponding primary limit is highlighted with red colour.
example, with \( HW_1 \) the scheduling pattern \( sched_1 \) has always a higher utility than that of \( sched_2 \). But this is only because we considered both throughput and latency for our optimisation. For example, if we were only interested in latency, then \( sched_2 \) would always lead to a better latency for \( HW_1 \).

If we would be interested whether \( HW_2 \) is a useful replacement for \( HW_1 \), we can indeed identify scenarios where this is the case. For example, with the system load given in the fourth line, \( HW_2 \) indeed reaches already a utility of 0.88, and reaches even the same utility as \( HW_1 \) with the system load given in the last line.

Having the choice between \( HW_2 \) and \( HW_3 \), Table 2 actually shows that from the utility point of view \( HW_3 \) is preferable compared to \( HW_2 \) for all listed load scenarios.

To summarise, with the tolerance-based real-time model we can indeed study the trade-off of different hardware configurations.

As a final note, it is worthwhile to say that the evaluation of the tolerance-based real-time model also could be combined with utility functions of other properties such as energy consumption. For example, using the utility function of energy consumption as an additional selection criterion would typically favour lower clock frequencies for the same peak speed. Back to our example, \( HW_2 \) typically could have a better energy consumption than \( HW_3 \) as long as the applied software scales well to two cores.

### 7 Summary and Conclusion

In this paper we introduced a tolerance-based real-time computing model. The novelty of this model is to express primary performance metrics and tolerance performance metrics separately from the critical performance metrics where the output utility becomes zero. Besides latency, the model can be also used for other metrics like throughput and jitter. Central to this model is the observation that the output utility does not abruptly decrease at the critical performance metrics, but rather decreases gradually. We presented a video processing use case to demonstrate how to derive the tolerance-based real-time model and how to use it to optimise the scheduling.

This tolerance-based real-time computing model has a wide range of applications. Besides the example where it has been used for trading between service quality and hardware costs, it can be used, for example, for fault-tolerant real-time computing. Besides resource outages, another source of fault could be an execution time overrunning its estimated WCET.

### References


