Article

ATMP-CA: Optimising Mixed-Criticality Systems Considering Criticality Arithmetic

Sajid Fadlelseed [1], Raimund Kirner [1] and Catherine Menon [1]

[1] Department of Computer Science, University of Hertfordshire, Hatfield, United Kingdom; email: {s.q.fadlelseed, r.kirner, c.menon}@herts.ac.uk

Abstract: In a safety-critical system typically not all provided services have the same criticality, which we call mixed-criticality systems. Criticality arithmetic, also called SIL arithmetic, is an approach to lower the development effort of a service by providing redundancy with tasks that are developed for a lower criticality level. In this paper we present ATMP-CA, which is a derivation of the multi-core scheduler ATMP. ATMP-CA is able to take into account the knowledge about the use of criticality arithmetic. ATMP-CA has a modified core allocation and procedure for utility optimisation, considering the context of the replicated tasks. We conducted experiments that show that ATMP-CA is able to provide the services using criticality arithmetic, while the reference schedulers were not.

Keywords: real-time systems; safety integrity level; scheduling; mixed-criticality

1. Introduction

Mixed-criticality systems are a special kind of safety-critical systems, where not all provided services have the same criticality. For example, in an aeroplane, the correct operation of the engines is of higher criticality than the onboard intercom system. With the seminal work by Vestal in 2007 [2], scheduling of mixed-criticality systems has become a quite active research field [3].

The development of services with higher criticality requires higher effort than services with lower criticality [4]. Criticality arithmetic – also referred to as *SIL arithmetic* – is a way of reducing that effort [5]. *Criticality arithmetic* is the process of realising a single function of importance to safety by combining multiple redundant independent components each of which implement this function. Should any one of these components fail, the others - being independent - will continue to provide this function. One consequence of this is that the correct and continued functioning of any single one of these components need not be assured to the same rigour as would be necessary if it alone were to be relied upon to provide this function.

Criticality arithmetic has a number of benefits as identified in Section 2.2.1. These largely refer to the reduced development and assurance cost, which is a result of each individual component being of lesser importance to safety than it might otherwise be. However, there are also a number of drawbacks, as discussed in Section 2.2.2. Using criticality arithmetic can make it more difficult to adequately determine the impact of individual component failures. *Criticality arithmetic* is the process of realising a single function of importance to safety by combining multiple redundant independent components each of which implement this function. Should any one of these components fail, the others - being independent - will continue to provide this function. One consequence of this is that the correct and continued functioning of any single one of these components need not be assured to the same rigour as would be necessary if it alone were to be relied upon to provide this function. Criticality arithmetic is sometimes referred to as *SIL arithmetic*.

So far, the use of criticality arithmetic is an informal and qualitative process with no formal universally-accepted definition. Individual standards prescribe different methods for, and constraints on, the use of criticality arithmetic within relevant domains as described in Section 2. As such, we do not attempt in this paper to define a quantitative method for estimating the increase in dependability afforded by the use of criticality arithmetic. Rather, we examine the consequences of mixed-criticality task scheduling in an environment where criticality arithmetic has been used.

Specifically, in this paper we show an example of modifying a scheduler to take advantage of the knowledge that criticality arithmetic is used in the system. To start with, we took the ATMP scheduler from Iacovelli et al. [6] and modified its core allocation and ILP constraint generation, so that the resulting scheduler provides a better handling of tasks that are replicated for criticality arithmetic. This ATMP scheduler takes as input utility functions for each task, so that the overall system utility can be gracefully distributed among tasks in case of resource shortages, e.g., caused by faults [7,8]

The remainder of the paper is structured as follows: Section 2 describes criticality arithmetic in further detail with the link to safety standards. Section 3 describes the system model that is used for the criticality-arithmetic-aware ATMP-CA, described in Section 4. An experimental evaluation is given in Section 5. Finally, Section 6 concludes this paper.

2. Criticality Arithmetic

While there does not exist a formal definition of *criticality arithmetic*, in this section we describe some practical aspects of its applicability.

2.1. Safety Integrity Implementations

Safety integrity of components or modules is an important concept across multiple domains, including the nuclear domain [4], the automotive domain [9] and civil aviation [10]. The standards associated with each of these domains identify terminology for denoting the safety integrity of a component, including *Safety Integrity Levels (SILs)*, in IEC 61508, *Automotive Safety Integrity Levels (ASILs)* in ISO 26262 and *Development Assurance Levels (DALs)* in ARP 4654. In this section we provide a brief introduction to safety integrity, using the SIL terminology of IEC 61508 [4] as an exemplar. Further details can be found in [5].

IEC 61508 [4], defines the *safety integrity* of a component as the probability of that component satisfactorily performing its specified safety function. It defines four *Safety Integrity Levels* (SILs), with a higher safety integrity level being ascribed to those components which are more important to safety. In this way, the SIL of a component is an indication of the extent to which that component is important with regards to the safety of the overall system. As an indication, Table Table 1 provides the association between the target failure rate of a component (the probability of failure on demand (PFD) or, for continuous operations, the probability of failure per hour (PFH)) and the consequent associated SIL of that component, as described in [4].

In more detail, the SIL assigned to a component provides information about the level of rigour expected when developing that component. Generally, development and validation of a component to a higher SIL requires more rigour than development of that component to a lower SIL. Specific details of the required development activities at each SIL are provided in [4], and cover areas including as test coverage, coding practices and acceptability of documentation. It is important to note that using the development techniques recommended for a particular SIL does not guarantee the achievement of a given failure rate. That is, Table 1 should be used only for determining the SIL based on the required failure rate and *not* to claim satisfaction of a target failure rate based on the use of specified development techniques.

SIL	*PFD*	*PFH*
4	10^{-4} to 10^{-5}	10^{-8} to 10^{-9}
3	10^{-3} to 10^{-4}	10^{-7} to 10^{-8}
2	10^{-2} to 10^{-3}	10^{-6} to 10^{-7}
1	10^{-1} to 10^{-2}	10^{-5} to 10^{-6}

Table 1: Resulting safety integrity level (SIL) from different failure probabilities PFD and PFH

As identified earlier, the concept of assigning integrity requirements is common to multiple domains. Broadly similar processes to the above are discussed in detail in a number of standards, including [10] (DALs) and [9] (ASILs).

2.2. Criticality Arithmetic and Safety Integrity

Criticality arithmetic (or SIL arithmetic as termed in [5]) refers to the practice of using multiple redundant independent implementations of a lower integrity level component providing a function F, in order to realise F at a higher integrity level than that of any of the individual components [11]. Criticality arithmetic therefore relies on the use of *functional redundancy*, or the duplication of certain critical system components which all provide a defined function. This means that if any one of these components fails, the remaining components will still be able to provide that function.

Different domains make use of this concept with their own specific integrity terminology. In IEC 61508 [4], *SIL arithmetic* is used when discussing ways in which hardware systems of different SILs can be combined, and in determining the SIL of the resultant combined system. Where a safety function is implemented via multiple channels with a given hardware fault tolerance, IEC 61508 defines that the overall SIL is calculated by identifying the channel with the highest SIL, and adding a number of integrity levels dependent on the hardware fault tolerance of the combined channels.

Similarly, ISO 26262 [9] identifies *ASIL decomposition* as a process permitting a safety function assigned a nominated ASIL to be decomposed into redundant safety requirements, satisfied by independent architectural elements. This is most often used to decompose a safety requirement into a functional requirement and a safety mechanism acting against failure of that functionality.

ARP 4754 [10] also uses criticality arithmetic within the civil aviation domain. This standard permits the establishment of functional failure sets with multiple members, where the Development Assurance Level (DAL) assigned to these members is permitted to be lower than the DAL assigned to the top-level failure condition.

In all these examples, we note that there is a limit to the extent of the criticality arithmetic which can be performed, i.e. to the consequent increase in criticality of the top-level component as a result of leveraging redundancy at lower levels. We further note that an effective system of redundancy management [12] is required in order to detect primary component failure and to reconfigure the system to use the redundant component in place of the primary. Effective redundancy also requires independence of the redundant components such that multiple components will not be affected, for example, by a common mode failure. Systems with redundancy built in can continue to operate - in some cases up to several days [13] - in the event of partial failure.

2.2.1. Benefits of Criticality Arithmetic

It is generally regarded as less resource-intensive to develop components at lower integrity levels, as the development and validation processes are correspondingly less rigorous. This is discussed in more detail in Section 2.1. As a result, employing criticality arithmetic in system development can provide benefits in terms of both reduced development time and reduced development cost. We do note, however, that demonstrating sufficient independence between these relevant components may still be a non-trivial task [14]. Criticality arithmetic also allows for the commercial pressures of developing and procuring systems. In some industries logistical factors mean that components have to be procured before their integrity levels can be assured. Should these components be later shown to have achieved a lower integrity level than that expected, criticality arithmetic may be used to address the gap. Similarly criticality arithmetic can in some cases permit the use of legacy components (i.e. where the development effort is already completed) at lower integrity levels [15].

Criticality arithmetic also permits the use of less-complex components, where these have achieved a lower SIL than that needed by the overall system. Components of lower complexity are easier to develop, as well as typically being easier and cheaper to maintain. Moreover, their use can reduce the risk of an undetected failure mode.

2.2.2. Drawbacks of Criticality Arithmetic

Although the use of criticality arithmetic may confer benefits as described in Section 2.2.1, it can also lead to conflicts in determining the impact of system or component failures.

Given a system where two or more redundant components are linked via criticality arithmetic to provide a service, a failure of one of these components eliminates the "protective" element of redundancy. Consequently, the entire service can no longer be adequately assured at the higher integrity, being now provided only by a single component of lower integrity.

This is of particular relevance where the integrity of the multiple redundant independent components (i.e. the SIL, DAL, ASIL) is used as input to a mixed-criticality task scheduler. Mixed-criticality scheduling allows for tasks of higher criticality to be given preference when resources are scarce. An example of such a scheduler we introduce in Section 4.

Informally, this means that where necessary those tasks most important to safety will be given preference over tasks of lesser importance to safety. However, if one of the components in the set providing a function F at higher integrity than that of the individual components (via criticality arithmetic) were to fail, the other components in this set would have to be given greater preference by the task scheduler if F is still to be maintained at the required integrity level.

A second issue relevant to scheduling is the case of multiple dependent (but not redundant) components working together to implement a service. In this situation a failure of one of these components could result in a failure of the overall service since an important part of the sub-goal would be no longer achievable. A mixed-criticality scheduler may therefore choose to abandon all the related tasks implementing the entire service.

3. System Model

In the following we describe our system model and assumptions. We assume a mixed-criticality system, which consists of multiple services that could have different levels of criticality. A service can be implemented by one task or multiple tasks using criticality arithmetic. The system provides a number of services:

$$s = \langle id, l, T \rangle$$

id is the service's name.

l is the service's criticality level, with $l > 0$. A higher value of l_i means a higher level of criticality. The vector $\vec{l}$ is used to represent all possible criticality levels in a system: $\vec{l} = (l_1, \ldots, l_k)$, with l_1 being the minimum and l_k being the maximum possible criticality level.

T is the set of tasks $\tau \in T$ that implement the service s. If only one task implements the service ($|T| = 1$), then no criticality arithmetic is used, and the task in this case has the same criticality as the service. If multiple services implement the service ($|T| > 1$), then criticality arithmetic is used: the criticality of each task $\tau \in T$ has a criticality less than the service's criticality, but their redundant execution

Each task τ of a task set T is defined as follows:

$$\tau = \langle uf, s, d, c, l, p, u \rangle$$

uf is the utility function of task τ. The input parameter of the utility function is the chosen period, i.e. throughput. The utility function is characterised by the following properties: $uf = \langle p_{prim}, p_{tol}, u_{tol} \rangle$. p_{prim} is the primary period, with the relative utility being 1.0 up to this period. At the tolerance period p_{tol} the resulting utility is u_{tol}, with the utility linearly interpolated between p_{prim} and p_{tol}.

s is the service that is implemented by task τ.

d is the relative deadline of task τ. We assume implicit deadlines, e.g. $d = p_{prim}$. (Note that this assumption is only chosen for the concrete scheduling test in our implementation, but it is not a requirement of our optimisation method.)

c is the WCET estimate of task τ. Depending on the underlying short-term scheduling protocol, the WCET estimate could be different for each criticality level. However, the mid-term scheduler described in this paper does not require this.

l is the criticality level of task τ with $l > 0$. A higher value of l means a higher level of criticality. The vector $\vec{l}$ is used to represent all possible criticality levels in a system: $\vec{l} = (l_1, \ldots, l_k)$, with l_1 being the minimum and l_k being the maximum possible criticality level.

p, u represent the task's chosen period and the resulting utility u. The period p can be chosen within the tolerance interval: $p_{prim} \leq p \leq p_{tol}$. The resulting utility is defined by the task's utility function: $u = uf(p)$. Section 3.1 We also use an *absolute utility* U, which is calculated as $U = u \cdot l$.

The aim of the method described in this paper is to find for each task τ_i a period p_i so that the overall system utility is maximised.

The individual instances of a task at runtime are called jobs. A job j is described by the following tuple:

$$j = \langle a, et, \tau \rangle$$

where a is the arrival time and et is the actual execution time. The entry τ refers to the task this job is instantiated from.

3.1. TRTCM

The fundamental concept of our scheduler is the *tolerance-based real-time computing model* (TRTCM) [1,8]. Instead of using a single performance limit like a deadline or throughput limit, in TRTCM a tolerance range is added, which allows in case of resource shortage a guided search for the best overall system utility.

In this paper we focus on optimising the throughput of a system based on TRTCM. For any period $\leq p_{prim}$ the relative utility is 1.0, i.e., the maximum. For any period higher than p_{prim}, the relative utiliy of a service degrades. This degradation is approximated by the utility function of a service, which defines another period p_{tol} up to which the service is still considered acceptable but with lower utility u_{tol}. For any period $p_{prim} \leq p \leq p_{tol}$ the relative utility is expressed as a linear function, as shown in Figure 1.

The utility u of a task is calculated from its period p. The periods of all tasks are adjusted such that the overall system utility if maximised under the given resource

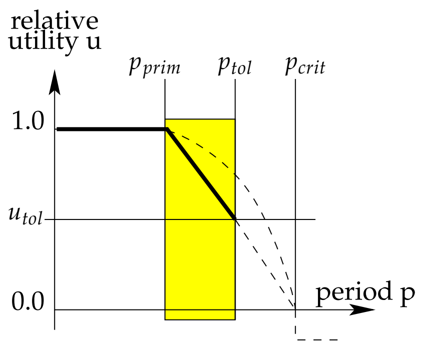


Figure 1. Utility function: calculate relative utility based on chosen period

constraints. For example, if some of the computing cores fail, then the system will have a significantly lower utility. But with the help of the utility functions the degradation can be done in a graceful way.

3.2. Computing Elements

The platform model consists of multiple computing elements, we call cores $cr \in Cores$. For our purpose is does not matter whether these cores are part of a multi-processor CPU, or whether they are on separate CPUs. The central objective is to optimise the system utility over all cores with potential shortage of resource.

We also model a notion of computing capacity $Cap(cr)$ for each core $cr \in Cores$. The computing capacity is linked to the worst-case execution time (WCET) $c(\tau)$ of a task τ, as the WCET is based on a reference computing capacity we denote as $Cap(cr) = 1.0$. If another core cr' has a computing capacity of $Cap(cr) = 2.0$, i.e., running twice as fast as the reference core, then the effective WCET of a task τ running on that cores would be $\frac{c(\tau)}{2}$. This way we can model, for example, the case that different cores are running with different clock frequencies.

The total computing capacity of the whole system can be calculated as

$$\sum_{cr \in Cores} Cap(cr)$$

This platform model allows the precise modelling of platforms with homogeneous cores. In case of non-homogeneous cores it would be better to instead use a different WCET of a task for each core $cr \in Cores$.

4. ATMP with Criticality Arithmetic

The Adaptive Tolerance-based Mixed-criticality Protocol (ATMP) [6], is an application of the TRTCM model [1,8] that maximises the system utility on each core by adjusting the periods of tasks within their tolerance range. The basic implementation of ATMP, categorises system tasks according to their adaptation capability. In other words, the ability of a task to relax its interarrival rates and its usefulness to the overall system decides if it will be allocated or not in case of computing resource shortage. In such a case, ATMP sort tasks according to decreasing criticality. Then, a critilcality-utility aware allocation for system tasks is performed on available cores. On each core, if the partitioned tasks on that core is schedulable, then it is processed by the underlying scheduler. Otherwise, a binary search heuristics with Integer Linear Programming (ILP) is performed to optimise the task set periods. The optimisation process maximises the overall system utility through maximising the utility of individual tasks by exploiting their safety margins. However, if no feasible solution is found for the whole task set, ATMP drops tasks with least criticality each step in the binary search according to their adaptation capability.

Our modification of ATMP consists of a criticality-arithmetic-aware allocation of tasks to computing cores, and an adaptation for the ILP objective function to consider different contexts of replicated tasks implementing a service based on criticality arithmetic.

4.1. Criticality-Arithmetic-Aware Allocation to Cores

The task allocation to cores in ATMP-CA differs from the original one in ATMP by avoiding that more than one of the replicated tasks from a service with criticality arithmetic gets allocated to the same core. The reason for this is simply to ensure fault tolerance for the replicated tasks, such that each core failure can disrupt at maximum one of the replicated tasks. In addition, the new core allocation also drops task replicas in case that there are more task replicas for one service than there are cores available. By dropping these replicas we avoid that they block computing resources on a core for no benefit.

Algorithm 1 shows the implementation of the core allocation in ATMP-CA. The algorithm has two input parameters: Γ, the list of all tasks, pre-sorted with decreasing criticality, and CS, the list of all cores available for allocation. The outer while-loop from line 2-12 runs as long as there are tasks in Γ. In line 3 the function $getTaskWithMaxCrit(\Gamma)$ removes from Γ the task with maximum criticality. In line 4 the list CS with all core ids is copied as CS'. This copy is needed in case of replicated tasks to make sure that no two replicated tasks of the same service end up on the same core. In line 5 the function $getCoreWithMinLoad(CS')$ removes from the core list CS' the core with currently the minimum task load assigned. Line 6-8 checks whether the task t_{id} already has a replica on the core c_{id}. If this is the case, then inside the loop a new core is extracted from CS' until either a core is found that has no replica of t_{id} allocated or all cores have been tried without success. Line 9-11 does register the allocation of the task t_{id} to core c_{id} only if the previous search for the core without a replica already allocated was successful. If this search was not successful, then task t_{id} is simply dropped and not allocated to a core. This search can only fail if there are less operational cores available than the number of tasks replicas a services using criticality arithmetic is implemented with.

Algorithm 1: Criticality-Arithmetic-Aware Allocation of Tasks to Cores

```
Input  : $\Gamma$: task list sorted by criticality;
         $CS$: list of cores;
1  begin
2      while $\Gamma \neq \emptyset$ do
3          $t_{id} \leftarrow getTaskWithMaxCrit(\Gamma)$;
4          $CS' = CS$;
5          $c_{id} \leftarrow getCoreWithMinLoad(CS')$;
6          while $hasReplica(t_{id}, c_{id}) \wedge CS' \neq \emptyset$ do
7              $c_{id} \leftarrow getCoreWithMinLoad(CS')$;
8          end
9          if $\neg hasReplica(t_{id}, c_{id})$ then
10             $addTaskToCore(t_{id}, c_{id})$;
11         end
12     end
13 end
```

When Algorithm 1 terminates, then each of the tasks in the taskset has been either allocated to a core or has been dropped. The purpose of this allocation is to just assign the tasks to a core. Later, within each core, as part of the utility optimisation, which is the same as in ATMP [6], some tasks might be removed again from a core in order to pass the schedulability test.

4.2. Formulation of ILP Problem for ATMP-CA

In this section we describe the ILP formulation to find the optimal task periods. We describe the constants and variables of that ILP problem, the goal function to optimise the system utility and the different constraints that have to be considered.

Optimisation parameters (constants): In ATMP the units of scheduling are tasks. As described in Section 3, each task τ of a task set T consists of the following components:

$$\tau = \langle uf, s, d, c, l, p, u \rangle$$

where the utility function uf is characterised by the following properties: $uf = \langle p_{prim}, p_{tol}, u_{tol} \rangle$. We model the utility function and the criticality of each task τ_i in the ILP problem with the following constants:

c_i	...	the WCET of τ_i
$p_{prim,i}$	...	the primary period (with utility $u_{prim,i} = 1.0$),
$p_{tol,i}$	...	the tolerance period,
$u_{tol,i}$	...	the utility at the tolerance period $p_{tol,i}$
WT_i	...	the criticality weight of τ_i
$Cap(cr)$	...	the computing capacity of $cr \in Cores$

The parameters $p_{prim,i}$, $p_{tol,i}$, $u_{tol,i}$ characterise a task's utility function by two linear lines, as shown in Figure 1. The horizontal line is a constant utility of 1.0, which can be directly expressed as an ILP constraint. The sloped line of each task's utility function can be also derived from $p_{prim,i}$, $p_{tol,i}$, $u_{tol,i}$, for which we have to calculate its slope k_i and y-intercept q_i to express it as a line equation:

$$\text{line equation} \ldots \quad u_i = p_i \cdot k_i + q_i \tag{1}$$

$$\text{slope} \ldots \quad k_i = \frac{u_{tol,i} - 1}{p_{tol,i} - p_{prim,i}} \tag{2}$$

$$\text{y-intercept} \ldots \quad q_i = \frac{p_{tol,i} - u_{tol,i} \cdot p_{prim,i}}{p_{tol,i} - p_{prim,i}} \tag{3}$$

Optimisation variables: We use the following optimisation variables to find the optimised task configurations:

p_i	...	the chosen period of task τ_i,
u_i	...	the relative utility of task τ_i,

Objective function The optimisation ILP goal function maximises the system utility through maximising the utility variable u_i of each task τ_i multiplied by its criticality weight WT_i:

$$SU_{tol} = \sum_{\tau_i \in TS} WT_i \cdot u_i \tag{4}$$

The criticality weight WT_i is explained below at the optimisation constraints.

Optimisation constraints We express the piecewise affine approximations

$$u_i = \begin{cases} 1 & \text{if } p_{prim,i} \geq p_i \\ p_i \cdot k_i + q_i & \text{if } p_{prim,i} < p_i \leq p_{tol,i} \end{cases}$$

of the utility functions to the following constraints:

$$u_i \leq 1 \tag{5}$$

$$u_i \leq p_i \cdot k_i + q_i \tag{6}$$

The *resource constraints* are used to limit the workload at each of the available cores $cr_i \in Cores$. The maximum workload a core cr can take is its computing capacity $Cap(cr)$:

$$\sum_{\tau_i \in TS} \frac{c_i}{p_i} \quad \leq \quad \sum_{cr \in Cores} Cap(cr) \tag{7}$$

The *tolerance constraints* determine the maximal acceptable period of p_i

$$p_i \quad \leq \quad p_{tol,i} \tag{8}$$

In ATMP, the weight WT_i is always set to the criticality $\tau_i.l$ of a task τ_i. In contrast, in ATMP-CA, the calculation of the weight WT_i of a task $\tau_i.l$ depends on the context with the replicas on other cores. In ATMP-CA, the WT_i is only set to the criticality $\tau_i.l$ of the task τ_i in case that another replica of the task has already been allocated with its maximum utility or in the cores still to be processed there is another replica of the task allocated. Otherwise the weight WT_i is set to the criticality $\tau_i.s.l$ of the service $\tau_i.s$ it implements, which is higher than $\tau_i.l$.

Algorithm 2: Calc-CA-Aware-ILP-Weight

```
Input  : $ø$: task for which to calculate its ILP weight $WT$;
1 begin
2     if $CoresHaveReplicaWithMaxUtil(\tau) \vee CoresHaveReplica(\tau)$ then
3         $l = ø.l$
4     else
5         $l = ø.s.l$
6     end
7     $WT = l$
8 end
```

The implementation of ILP formulation in ATMP-CA to calculate the weight WT_i is shown in Algorithm 2. The algorithm has single input, the task τ, for which we want to calculate the weight WT_i as used in Equation 4. In line 2, function $CoresHaveReplicaWithMaxUtil(\tau)$ whether a replica of task τ has been optimised with maximum utility in already processed cores, and function $CoresHaveReplica(\tau)$ checks whether the not-yet processed cores included an allocation of a replica of task τ. If one of these two functions returns *True*, then we chose in line 3 the task's criticality $ø.l$. Otherwise, we chose in line 4 the criticality $ø.s.l$ of the task's service $ø.s$. In line 7, the weight WT for the ILP objective function is set to the determined criticality l.

5. Experiments

In the following we describe the setup and results of our experiments.

5.1. Setup of Experiments

We have implemented an ATMP-CA scheduling simulator as described in Section 4. We configured the simulator to simulate a multi-core system with 10 cores, where we simulate fault scenarios by making 10, 4, or 2 cores out of the 10 cores available. This way we can simulate the resulting overall system utility for different cases of resource shortage. Besides the ATMP-CA protocol, this simulator has also implemented the ATMP and SAMP protocol for reference, as described in [6]. In essence, ATMP is similar to ATMP-CA in the sense that it also does a utility optimisation, but its core allocation and ILP constraints for utility maximisation does not take into account that any of the services use criticality arithmetic. So, the comparison of ATMP-CA and ATMP should show what is the potential benefit of supporting in the scheduler any knowledge about criticality arithmetic. The other scheduler SAMP is generally less capable than ATMP and is just included for further reference. We then also implemented a protocol SAMP-CA, which is basically the simple SAMP protocol, but using for core allocation the new Algorithm 1, which is also using knowledge about criticality arithmetic. As such, SAMP-CA might perform better for systems with criticality arithmetic than SAMP, but it is not supposed to be able to compete with the utility optimisation performed by ATMP-CA.

Service:	*name*	S1	S2	S3	S4	S5	S6	S7	S8
	crit.	HI	HI	**HI**	**HI**	LO	LO	LO	LO
Task:	*name*	T1	T2	$T3_a$, $T3_b$	$T4_a$, $T4_b$	T5	T6	T7	T8
	crit.	HI	HI	**LO**	**LO**	LO	LO	LO	LO

Table 2: Set of Services/Tasks (S3 and S4 use Criticality Arithmetic)

We have generated a taskset with random parameters for worst-case execution time c and utility function uf. The implicit deadlines d are chosen to be equal to the primary period p_{prim}. The criticality of a task or service is either HI or LO, which corresponds to a numeric value of either 2.0 or 1.0 respectively. We have constrained the task generation such that it includes two normal HI services (S1, S2), two HI services that use criticality arithmetic (S3,S4), and a few other LO services (S5, S6, S7, S8). The whole structure of this taskset is shown in Table 2. As shown in the table the tasks T1 and T2, which implement the HI services S1, S2 have the same criticality as the service itself. However, the HI services S3 and S4, which use criticality arithmetic, are both implemented by two redundant tasks $T3_a$, $T3_b$ respectively $T4_a$, $T4_b$, which have all criticality LO.

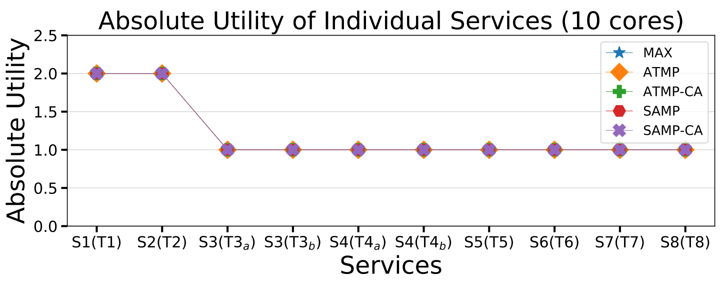
5.2. Results of Experiments

Figure 2 shows the results for our experiments with either 10, 4, or 2 cores out of 10 cores available. The *MAX* line denotes the maximum possible absolute utility for each task, which is either 2.0 or 1.0.

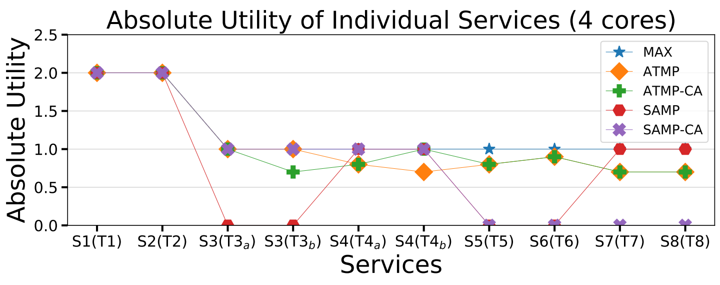
Figure 2.a shows the case with no resource shortage, i.e. all 10 cores out of 10 cores are available. This case represents the optimal allocation, which means for each task it was possible to assign them their primary period p_{prim}, resulting in the maximum relative utility of 1.0, and no service is dropped at all.

Figure 2.b shows the case with 4 cores out of 10 cores available. Here, SAMP allocated the tasks of all HI-criticality services S1, S2 and S4 to cores, except the two LO-criticality task replicas $T4_a$,$T4_b$ that implement service S3, whereas SAMP-CA allocated all HI-criticality tasks to cores at the cost of dropping all LO-criticality tasks. Though both SAMP and SAMP-CA show an equivalent number of allocated and dropped tasks, they differ in the behaviour of dropping tasks belong to services with criticality arithmetic. For ATMP and ATMP-CA, both protocols have successfully allocated all services to cores, with a slight degradation of their absolute utility without dropping any tasks at all. The effect of the criticality-arithmetic-aware generation of the ILP objective function as described in Section 4.2 in this case for service S3. With ATMP-CA the first task $T3_a$ has been set to full utility, resulting in the degraded utility in $T3_b$, which allows to give resources to other tasks. Since both tasks implement the same service S3, there is no need to allocate both of them at maximum utility when the system experience an overload as seen by the classical ATMP. The similar effect can be seen with service S4, where ATMP-CA allows a degraded utility for task $T4_a$ and then gives full utility to $T4_b$, while with ATMP both tasks of S4 are degraded.

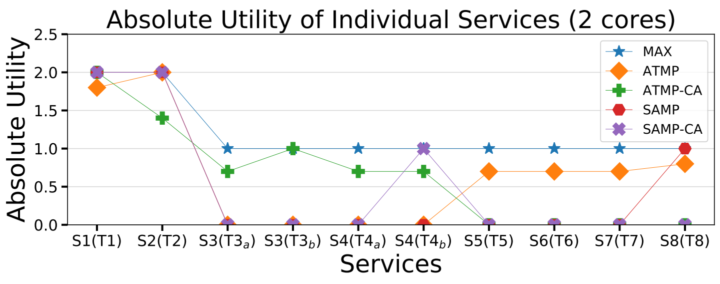
Figure 2.c shows the case with 2 cores out of 10 cores available. Here, SAMP retained the tasks of HI-criticality services S1 and S2 and just one LO-criticality task T8, but dropped all the tasks of all other HI-criticality services, including S3 and S4. SAMP-CA performed already a bit better by retaining the tasks of HI-criticality services



a) 10 cores available, 0 cores failed



b) 4 cores available, 6 cores failed



c) 2 cores available, 8 cores failed

Figure 2. Experiment: Absolute Utility of Individual Services (Replications: service S3 with tasks $T3_a$,$T3_b$, and service S4 with tasks $T4_a$,$T4_b$)

S1 and S2 and also retaining one task T4$_b$ of HI-criticality service S4, while dropping all other services. This shows that the criticality-arithmetic-aware core allocation in SAMP-CA shows some benefit, but generally both SAMP and SAMP-CA have limited performance, as they don't support the flexibility with the tolerance range. On the other hand, ATMP and ATMP-CA have shown the significant difference of dropping Hi-criticality services. ATMP retained two of HI-criticality services and four LO-criticality tasks but dropped both S3 and S4 replicas, where ATMP-CA have successfully allocated all HI-criticality services including the replicated tasks and dropped all LO-criticality ones. In addition, ATMP-CA have allocated S3 replica task D with maximum utility as result to the degradation of task C, and both S4 replicas been degraded which shows that the modified optimisation process couldn't find a solution that allocate task F at maximum utility.

5.3. Discussion

The aim of the experiments was to show the benefit of taking into account knowledge about criticality arithmetic in the design of a mixed-criticality scheduler. The experiments have shown that the criticality-aware core allocation in ATMP-CA and also SAMP-CA had a benefit over ATMP respectively SAMP, by ensuring that tasks from services using criticality arithmetic are allocated to separate cores. Also, the modified ILP objective function of ATMP-CA has shown to make better use of resources in case of resource limitations. Overall, ATMP-CA allows an even more smooth degradation compared to ATMP in case of services using criticality arithmetic.

The limitation of the current experiments is that we only looked into systems with two criticality levels. Future work would be to extend the method to multiple levels of criticality.

6. Summary and Conclusion

In this paper we described the concept of criticality arithmetic, also known as SIL arithmetic, which is a technique to reduce the required development effort of a service by using task replication. The contribution of the paper is the development of ATMP-CA, a mid-term scheduler that takes into account information about criticality arithmetic to provide a graceful degradation of system utility in case of resource shortages, for example, caused by faults. This way, ATMP-CA can optimise the overall system utility in case of resource shortages, with special consideration of services implemented via criticality arithmetic. While it is common to limit the source of resource shortages in mixed-criticality systems to overruns of WCET estimates, we are able to consider any source of resource shortage, including failure of computing elements.

We have conducted experiments, comparing ATMP-CA with ATMP and also the simpler scheduler SAMP. The results show that ATMP-CA is capable to serve systems with criticality arithmetic better than the others. For example, ATMP-CA was the only scheduler that was able to retain service S3 with criticality arithmetic in case that 8 out of 10 cores failed. In the case of 6 out of 10 cores failed, both ATMP-CA and SAMP-CA (an extension of SAMP with the core allocation the same as ATMP-CA) were able to retain service S3 with criticality arithmetic. The latter case shows that considering criticality arithmetic during the core allocation is important on its own for criticality arithmetic.

Author Contributions: Motivation of criticality arithmetics: C.Menon; design of the ATMP-CA method: S.Fadlelseed and R.Kirner; implementation: S.Fadlelseed; supervision: R.Kirner.

Funding: This research received no external funding.

Acknowledgments: The authors would like to thank Saverio Iacovelli for help on the implementation of the scheduling framework.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kirner, R.; Iacovelli, S.; Zolda, M. Optimised Adaptation of Mixed-criticality Systems with Periodic Tasks on Uniform Multiprocessors in Case of Faults. Proc. 11th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'15); , 2015.
2. Vestal, S. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. Proc. 28th IEEE International Real-Time Systems Symposium (RTSS'07), 2007, pp. 239–243. doi:10.1109/RTSS.2007.47.
3. Burns, A.; Davis, R.I.; Baruah, S.; Bate, I. Robust Mixed-Criticality Systems. *IEEE Transactions on Computers* **2018**, *67*, 1478–1491. doi:10.1109/TC.2018.2831227.
4. International Electrotechnical Commission. Functional Safety of Electrical / Electronic / Programmable Electronic safety-related systems. IEC standard 61508, 1998.
5. Catherine, M.; Saverio, I.; Raimund, K. Analysis for Systems Modelled in Matlab/SimulinkUsing SIL Arithmetic to Design Safe and Secure Systems. Proc. 23rd IEEE International Symposium on Real-time Distributed Computing. IEEE (2020), 2020, pp. 213–218.
6. Iacovelli, S.; Kirner, R.; Menon, C. ATMP: An Adaptive Tolerance-based Mixed-criticality Protocol for Multi-core Systems. Proc. 13th International Symposium on Industrial Embedded Systems (SIES'18); , 2018.
7. Anderson, J.S.; Ravindran, B.; Jensen, E.D. Consensus-driven distributable thread scheduling in networked embedded systems. Proc. International Conference on Embedded and Ubiquitous Computing; Springer-Verlag: Berlin, Heidelberg, 2007; pp. 247–260.
8. Kirner, R. A Uniform Model for Tolerance-Based Real-Time Computing. Proc. 17th IEEE Int'l Symposium on Object/Component/Service-oriented Real-Time Distributed Computing; , 2014; pp. 9–16. doi:10.1109/ISORC.2014.8.
9. International standards Organisation. ISO26262: Road vehicles – Functional safety. ISO/DIS standard 26262, 2011.
10. SAE International. ARP4754A:Guidelines for Development of Civil Aircraft and Systems. SAE standard ARP4754A, 2010.
11. Wu, P. Preventing interference between subsystem blocks at design time, 2015. US8938710B2.

12. Frigerio, A.; Vermeulen, B.; Goossens, K. Component-level ASIL Decomposition for automotive architectures. *Proceedings of the 2019 International Conference on Dependable Systems and Networks Workshops* **2019**, pp. 62–69.
13. John Rushby. A Comparison of Bus Architectures for Safety-Critical Embedded Systems **2003**.
14. Piovesan, A.; Favaro, J. Experience with ISO 26262 ASIL Decomposition. *Proceedings of the Automotive SPIN Italia Workshop* **2011**.
15. Ward, D.; Crozier, S. The uses and abuses of ASIL decomposition in ISO 26262. *Proceedings of the Society of Automotive Engineers World Congress* **2012**.