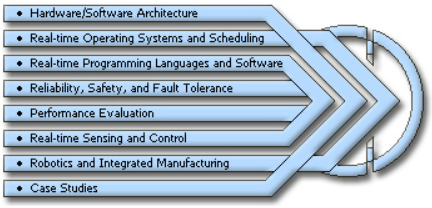
IEEE Real-Time Systems Symposium
RTSS 2014
December 2-5, 2014
Rome, Italy



IEEE
TCRTS Technical Committee on Real-Time Systems

WMC

Proceedings of the 2nd International Workshop on Mixed Criticality Systems

Edited by Liliana Cucu-Grosjean and Rob Davis

Program Chairs

Liliana Cucu-Grosjean
Rob Davis

Steering Committee

Sanjoy Baruah
Liliana Cucu-Grosjean
Rob Davis
Claire Maiza

Program Committee

Sebastian Altmeyer
Sanjoy Baruah
Iain Bate
Gerhard Fohler
Laurent George
Haohan Li
Nan Guan
Claire Maiza
Vincent Nelis
Sophie Quinton
Marcus Völp
Gabriel Parmer
Suzanne van der Ster
Wang Yi

Message from the Program Chairs

It is our pleasure to welcome you to the 2nd International Workshop on Mixed Criticality Systems (WMC) at the Real-Time Systems Symposium (RTSS) in Rome, Italy on 2nd December 2014.

The purpose of WMC is to share new ideas, experiences and information about research and development of mixed criticality real-time systems.

The workshop aims to bring together researchers working in fields relating to real-time systems with a focus on the challenges brought about by the integration of mixed criticality applications onto singlecore, multicore and manycore architectures. These challenges are cross-cutting. To advance rapidly, closer interaction is needed between the sub-communities involved in real-time scheduling, real-time operating systems / runtime environments, and timing analysis.

For this second edition of the workshop a total of 8 submissions were received. The review process involved 13 Program Committee members, with each submission receiving at least 3 reviews. In total, 7 papers were selected for presentation. Our thanks go to the WMC Program Committee for the time and effort they put into carefully reviewing the submissions, and for meeting the tight timescales set for reviews.

In addition to the regular papers, the workshop program also includes an invited talk from Risat Pathan (Chalmers University of Technology, Sweden). His talk entitled "Real-Time Scheduling of Mixed-Criticality Systems: What are the "X" Factors?" will introduce another factor relevant to high criticality tasks.

WMC 2014 would not be possible without the hard work of a number of people involved in the organisation of RTSS. In particular, we would like to thank the RTSS 2014 Workshops Chair, Rodolfo Pellizzoni (University of Waterloo, Canada) for his excellent organisation of the overall workshop program. We also thank the WMC Steering Committee for their guidance, and the MCC (UK EPSRC EP/K011626/1), Proxima (EU FP7 IP 611085) and Departs (French BGLE O16526-405635) projects for their support.

Finally, we would like to thank *all* of the authors who submitted their work to WMC 2014, whether it was accepted or not; without them, this workshop would not be possible.

We wish you an interesting and exciting workshop and an enjoyable stay in Rome. We look forward to seeing you again at WMC 2015.

Liliana Cucu-Grosjean (INRIA, Paris-Rocquencourt, France)
Rob Davis (University of York, UK)
WMC 2014 Program Chairs

Table of Contents

INVITED TALK

Risat Pathan (Chalmers University of Technology)

Title: Real-Time Scheduling of Mixed-Criticality Systems: What are the "X" Factors?

Abstract: Mixed-criticality (MC) systems consist of tasks with different degrees of importance or criticality. Correctly executing relatively higher critical tasks (e.g., meeting their deadlines) is more important than that of any lower critical task. Therefore, scheduling algorithm and its analysis have to consider runtime situations where the *correct* execution of higher critical tasks can be threatened by some events that I call "X" factors of MC systems. Example of such an X factor is "execution overrun" which is pointed out by Steve Vestal in RTSS 2007. The purpose of my talk is to highlight another X factor: the frequency of error detection and recovery.

The design and analysis of real-time scheduling algorithms for safety-critical systems is a challenging problem due to the *temporal* dependencies among different design constraints. This work is based on scheduling sporadic tasks with *three* interrelated design constraints: (i) meeting the hard deadlines of application tasks, (ii) providing fault tolerance by executing backups, and (iii) respecting the criticality of each task to facilitate system's certification. First, a new approach to model mixed-criticality systems from the perspective of fault tolerance is proposed. Second, a uniprocessor fixed-priority scheduling algorithm, called fault-tolerant mixed-criticality (FTMC) scheduling, is designed for the proposed model. The FTMC algorithm executes backups to recover from task errors caused by hardware or software faults. Third, a sufficient schedulability test is derived, when satisfied for a (mixed-criticality) task set, guarantees that all deadlines are met even if backups are executed to recover from errors. Finally, evaluations illustrate the effectiveness of the proposed test.

System Mode Changes - General and Criticality-Based

A. Burns
Department of Computer Science,
University of York, UK.
Email: alan.burns@york.ac.uk

***Abstract*—In this paper we summarise, and attempt to unify, the many descriptions that have been published on general mode changes. We then use this summary to position the criticality mode change. We conclude that a criticality mode change (from low to high) is closest in nature to a (graceful) degradation mode change following (partial) system failure. However, a criticality mode change (from high to low) has more in common with a (exceptional) functional mode change. The paper also addresses systems that may have both criticality and general mode changes.**

I. Introduction

Many real-world applications involve systems that operate in a number of clearly defined *modes*. Aircraft flights progress through phases (e.g. taxiing, take-off, climbing, level flight, etc) and automotive systems have modes to cover start-up, cruise control, driver control, 'limp home' etc.

If a system has more than one mode then there must be a *mode change protocol* to control how the system moves between modes. Such (general) protocols have been the subject of considerable study over a number of years [26], [10], [30], [20], [29], [13], [23], [21], [1], [11].

The more recent literature on supporting mixed criticality systems has identified situations in which the system must move from one criticality level to another [31], [6], [5], [12], [24], [15], for example, in a dual criticality system (low and high) a move from the low criticality mode to the high criticality mode. As a consequence of this move some low criticality work is abandoned (either temporally or permanently).

This type of criticality mode change has a number of similarities with the more general mode changes, but there are some important differences. The abandonment of work (even of a lower criticality) is clearly unacceptable in a fully functioning system, but it may be acceptable as part of a response to a (partial) system failure. In this paper we argue that a criticality mode change is equivalent to a particular form of general mode change (providing graceful degradation).

The paper is organised as follows; next we review the models and forms of analysis for general mode changes. In Section III, we use this context, to position the definition of a *criticality mode change*. Systems with both general and criticality mode changes are considered in Section IV. Conclusions are provided in Section V.

II. General Mode Changes

To formalise what it means for a system's behaviour to be described in terms of modes, a number of aspects need to be covered:

- Type – what are the different classes of modes.
- Trigger – what causes a mode change.
- Protocol – how is the mode change managed.
- Attributes – properties of modes.
- Definition – the software, and its operational parameters, that constitute a mode.
- Analysis – in particular the scheduling analysis used to verify the timing properties of the system during a mode change.

We look at each of these in turn. We assume a standard system model in which periodic and sporadic tasks, characterised by their minimum inter-arrival time, deadline and one or more measures of their worst-case execution time, give rise to a potentially unbounded sequence of jobs.

A. Type

In the literature on mode change protocols (cited above) three distinct types of mode can be identified. These provide a natural partitioning of the functionality of the system:

1) Normal Functional Modes – the application moves through a number of different phases. These phases are planned and are entered regularly. An example would be moving from driver control to cruise control in a standard family car.
2) Exceptional Functional Modes (sometimes called *Operational mode*) – rare events that will cause code to be executed that is not otherwise required. The response is planned, but the resulting mode change may never occur. An example would be 'prepare for crash' mode contained within a car – when the on-car monitoring system detects that an impact may be about to happen, it winds up the windows, tightens the seat belts, applies the brakes so the pads are just in contact with the disks (making them more responsive if applied by the driver) and prepares to deploy the airbag.
3) Degraded Functional Modes (sometimes called simply *Graceful Degradation*) – errors require load to be shed and priority given to issues of safety and minimum functionality. General responses to mode change events

are planned but the full set of error conditions may not be known in advance. An example would be a 'limp home' mode following engine sensor failure.

A system might have ten or more normal modes that are progressed through in a statically defined sequence. It might also have a small number of exceptional modes, and perhaps one or two degraded modes. With more complex systems, modes might be organised hierarchically.

B. Trigger

The mode change *event* (sometimes called *request* or *trigger*) is typically related to the state of the system or the system's environment as indicated via an input reading or an internal state change. For example, a driver touching the brake peddle will generate an event that will move the engine control system from 'cruise control' to 'driver control'.

An event could be *timed-trigger* if the mode change is coordinated to the 'time' of the environment. So a power generation system may switch modes at midnight. Also air traffic control systems have day and night modes (which switch over at a particular time – or at least should do, failure to return to 'day' mode being a cause of system failure in the past (see http://www.bbc.co.uk/news/uk-25278163). An example of a relative time trigger is a data collection mode that executes for just 10 minutes before returning to some previous mode.

For graceful degradation the trigger may come from the hardware platform or some health monitoring subsystem.

C. Protocol

The mode change event requires a protocol to manage the actual mode change. Such protocols can also be characterised in three ways:

1) Immediate – the mode change event causes an immediate mode change with the old mode jobs being *suspended* or *aborted*, and new mode jobs starting immediately.
2) Bounded (sometimes called *synchronous* [21]) – within a bounded time from the mode change event a point in time is reached in which there are no active jobs from the old mode and hence a clean switch of modes is then possible.
3) Phased (sometimes called *asynchronous*) – following the mode change event old mode jobs are allowed to complete, and new mode jobs are started within a bounded time.

With Immediate and Bounded, the system is only ever in one mode; with Phased there is a (limited) interval of time in which the modes are overlapping. Some jobs from the new mode have started while other old-mode jobs are yet to complete.

Overlapping can also happen in a distributed or multiprocessor system in which a phased change is necessary as it is not possible to simultaneously inform the entire system of the need to change mode. The propagation of the mode change event will inevitable take time.

Phased changes are the most difficult to analyse as the load on the system is typically higher during the change than it is in either mode [30]. Analysis can however be used to start new tasks as soon as possible commensurate with all deadlines being met during transition [20]. Here a worst-case scenario is assumed, with all old-mode tasks releasing a job just before the mode change occurs. Each of these jobs is allowed to terminate. Each new-mode task has a temporal offset (from the time of the change event) that the scheduling analysis has furnished. So this offset is the minimum possible that will not undermine schedulability during the mode change.

The completion of the old mode is usually defined to be when all current jobs, released in the old mode, have completed. However in some situations a number of old-mode jobs may need to be executed to complete the work of that mode. For example, a buffer of sensor input values from the old mode may need to be cleared before the mode change can occur [21]. And in distributed systems a series of old-mode messages may be in transit and need to be delivered and processed before the mode can be considered complete.

A single processor system implemented via a cyclic executive can easily support a Bounded mode change by waiting until the end of the current major cycle and then switching the (pre-computed) scheduling tables. A system using fixed priority scheduling or EDF can also support a Bounded change by waiting for the next idle tick and then changing the set of eligible tasks [29]. An idle tick is an instant in time when there is no work to undertake apart from new jobs released at that instant. Clearly there can be no causal effect from before to after an idle tick. For multiprocessor systems the coordination of the mode change across the entire platform is more of a challenge [25].

The definition of the mode change protocol must be closely tied to the form of analysis used to verify the system's behaviour (as indicated below).

D. Attributes

In this subsection we define a number of attributes that have been used to define properties of modes and mode change protocols.

A mode is *re-entrant* if it can be returned to at some time after it was left. Other modes can be termed *one-shot* (or *single-shot*) if they can only be entered once, *sink* if the system never leaves this mode once it has entered it, or *initial* if the system always starts in that mode. The full set of modes is *cyclic* if the system systematically and repeated moves through the modes. Alternatively the set of modes is *connected* (sometime called *strongly connected*) if the system can move from any mode to any other mode. Obviously a connected set does not have any one-shot or sink modes.

A mode that has aborted jobs is not usually re-entrant. A mode which can give rise to suspended jobs can, however, be re-visited with the suspended jobs continuing from the state they were in.

These definitions are useful but do not cover all interesting cases. For example, a system with four modes, A to D, might have the behaviour that it can start in A or B; it moves backwards and forwards between these modes unless some

event occurs that moves the system to mode C, A and B are never returned to but the system then moves between C and D. The descriptive terms can be assigned to the pairs of modes but not to the individual modes.

Although the above classifications are independent there is some common coupling: Functional mode changes tend to be Bounded and often re-entrant, Operational changes can be Immediate or Phased and may lead to the use of one-shot modes, and Graceful Degradation may require Phased or Immediate changes, and the degraded mode may well be a sink mode.

A particular case of Graceful Degradation concerns execution time overruns. If a task, due to a software error, enters an infinite loop then the only recovery strategy is to abort the task (its current job must be abandoned). This will require an Immediate change. Later the task could be restarted (*cold restart*) or an alternative task introduced in the new 'mode'. This task could start *cold* (no relevant internal state) or *warm* (if it has access to state updated by the aborted task). A *hot* standby would most probably be present in the old mode, but be deemed more important once it had taken over from the aborted primary task [22].

E. Definition

In terms of the code contained within a mode, a mode change may involve:

- Tasks that run unaffected in both modes.
- Tasks that run only in the old mode.
- Tasks that run only in new mode.
- Tasks that run in both modes but have their defining parameters changed.

In the latter case, a task could have its period and/or deadline altered, and in a fixed priority scheme its priority. A suspended job may actually be allowed to execute at a background priority; hence there is some overlap in these definitions between tasks that only execute in the old mode, and those that run in both modes but with diminished urgency in the second mode.

A task that has the same release characteristics, but which undertakes altered functionality in the new mode may have a different worst-case execution time in the new mode.

Finally, once 'criticality' becomes a task parameter then it is possible for a task to remain unchanged during a mode change, but for its designated criticality to alter. The hot standby introduced above is an example of such a change.

F. Analysis

From a schedulability point of view, different modes have different code requirements. So schedulability in one mode does not imply schedulability in another mode or in any Phased mode change. All modes must be checked, and all Phased changes.

For Immediate mode changes there is no specific scheduling problem, but there is an obligation on the RTOS and/or real-time programming language to facilitate immediate task suspension and/or aborting (which may be quicker). Suspension is needed for re-entrant modes, abort for non re-entrant. If a suspended or aborted job could be holding a resource that is used in the new mode then action must be taken to recover the resource or to allow a 'suspended' task to continue to execute until it has released the resource. From a scheduling point of view the mode change may therefore not be truly immediate.

For Phased changes there has been scheduling analysis produced [20] that computes a set of minimum release offsets for each new mode task. Whenever the mode change event occurs these offsets will ensure that all old mode jobs complete by their deadlines, but new mode tasks start as soon as possible. The scheduling of Phased changes is complicated by tasks that change their periods. A seemingly simple change of a task that moves from requiring 6 ticks of computation every 20 to 3 every 10 (or visa versa) can cause deadline misses on other non-changing tasks.

A final complication with Phased changes comes from the possibility of overlapping phases; e.g. during the move from mode A to mode B, a move to mode C is required. Systems tend to avoid this difficult to analysis situation by not allowing a further change until the current change has been completed. However, it may again be necessary to wait until there is a system idle tick before a Bounded or Phased change can be guaranteed to be complete.

A complex system with a large number of modes and possible mode changes can be modelled using state and state transitions formalisms [21]. Formal analysis can be used to verify that a system always remains within safe modes [1].

III. Criticality Mode Changes

In this section we review the literature on mixed criticality systems (MCS) that has utilised the notion of criticality modes and mode changes.

Consider a system with N criticality levels, $L_0 \dots L_{N-1}$, executing on a uniprocessor and using priority based scheduling of constrained tasks. Perhaps up to five levels of criticality may be identified in a system (see, for example, the IEC 61508, DO-178B, DO-254 and ISO 26262 standards). Typical names for the levels are ASIL (Automotive Safety and Integrity Levels) and SIL (Safety Integrity Level). It should be noted that not all papers on MCSs assign to 'criticality' the same meaning, an issue explored by Graydon and Bate [14].

The standard MCS's model [31], [6], [5], [12], [24], [15] has the following properties:

- Each task in the system is characterised by the minimum inter-arrival time of its jobs (period denoted by T), deadline (relative to the release of each job, denoted by D) and worst-case execution time (one per criticality level), denoted by $C(L_0) \dots C(L_{N-1})$. A key aspect of the standard MCS model is that $L_x > L_y \rightarrow C(L_x) \geq C(L_y)$.
- The system starts in the L_0 mode, and remains in that mode as long as all jobs execute within their low criticality computation times ($C(L_0)$).
- If any job executes for its $C(L_0)$ execution time without completing then the system immediately moves to the

next criticality mode, L_1.
- As the system moves to the L_1 mode all L_0 criticality tasks are abandoned. No further L_0 criticality jobs are executed.
- The system remains in the L_1 mode unless a job executes for its $C(L_1)$ execution time without completing, the system then immediately moves to the next criticality mode; jobs with criticality level L_1 are dropped.
- This process continues (potentially) until the top criticality mode is reached (L_{N-1}) with only tasks of this criticality level executing.
- Tasks are assumed to be independent of each other (they do not share any resource other than the processor).

This abstract behavioural model has been very useful in allowing key properties of mixed criticality systems to be derived, but it has been necessary to extend the model to allow for more realistic characteristics such as allowing some lower criticality work to execute in the higher criticality modes and for the lower criticality modes to be reinstated when conditions are appropriate. This is covered in the following papers [5], [28], [27], [18], [9], [4], [16], [17].

So the standard model (SM) defines a path from L_0 to L_{N-1}. The adaptive model (AM) allows movements in the opposite direction.

Note that work has also been focused on criticality-aware resource control protocols that will allow resource sharing between tasks [7], [32], [19], [33]. This work does not however directly impact mode changing unless resources can be used by tasks of different criticality.

A. Characteristics of a criticality mode change

Using the terms introduced in the Section II we can define the above SM criticality mode change protocol as follows

- L_0 is the initial mode.
- L_{N-1} is a sink mode.
- All modes are one-shot.
- Mode transitions are Immediate (or Phased in some models where executing lower criticality jobs are allowed to complete – though usually their deadlines are not guaranteed).
- Following a mode change some tasks only execute in the old mode.
- Some tasks execute in both modes, but their execution times are increased[1].
- There are no 'new mode' tasks.

As discussed above the more expressive and adaptive mode (AM) allows systems to regain functionality and move back towards the initial (fully functional) mode [5], [28], [27], [18], [9]. AM is therefore characterised as follows:

- L_0 is the initial mode.
- There are no sink modes.
- Mode transitions are typically Bounded.
- All modes are re-entrant.
- Some tasks execute in both modes, but their execution times (and periods) are deemed to vary.
- There are new-mode tasks when moving mode in the direction of L_0.

But what type of mode change are these? First for the standard model (SM). Early papers on MCS [31], [6] were clear that the initial L_0 mode is the only expected state for the system to be in. Other criticalities were only introduced so that scheduling analysis can be used to reduce the resource needs of the system. This is done by leveraging the pessimistic execution times assumed for high criticality tasks in the higher criticality modes.

In a two criticality system (LO and HI), these pessimistic values (the $C(HI)$ values) are *not* expected to be experienced at run-time. Indeed the $C(LO)$ values are most likely to also be pessimistic (though less so of course).

Therefore, a task executing for longer than expected (beyond $C(LO)$) can be deemed to be at fault. And hence a criticality mode change should be described as a form of Graceful Degradation. If one accepts this view then of the N modes, only one reflects normal functionality, all the other $N-1$ are forms of degraded service – as increasing levels of functionality are being dropped.

For the adaptive model (AM) mode changes are better defined as exceptional (operational). They are planned but may not occur.

All protocols and forms of analysis that have been developed for general mode changes are directly applicable to criticality mode changes (albeit often in a simpler form as a criticality mode change does not have all the characteristics of the more general protocol). So, for example, in the standard model where L_0 is the initial mode and L_{N-1} is the sink mode, there are no new-mode tasks. But in the more adaptive scheme where lost work can be returned to (i.e. L_{N-1} is not a sink mode) then new-mode tasks will need to be supported.

In the general literature on fault tolerance, recovering from an error (or partial failure) can either be: degraded service followed by active recovery, or degrading service followed by 're-boot' (e.g. channel re-initialisation in an avionics system). With a 're-boot' the system, in effect, moves from the sink mode to the initial mode, but this is done outside the model of the software. With active recovery the system recovers by moving away from the degraded modes, there are no longer sink modes.

For mixed criticality, the standard model (SM) assumes that the software cannot return to L_0. Active recovery requires an adaptive protocol (AM).

IV. Systems with Both General and Criticality Mode Changes

Having established that the main SM criticality mode change is usefully defined as a form of graceful degradation, it seem perfectly reasonable for a large system or system of systems to have both general and criticality mode changes. Some points of interest are:

[1]Some models for MCS have period as well as execution time being criticality dependent [8], [2], [4], [3]; in these models a task's period may reduce (as well as computation time increase) during a criticality mode change.

- Assume the system consists of a set of applications, of potentially different criticality levels.
- A General Mode Change may impact on just one or a subset of applications and therefore criticality levels.
- Graceful Degradation, in general, is most likely to be influenced by criticality.
- A General Mode Change Protocol may involve some tasks changing their criticality designation.

In the latter case a set of tasks may be more critical, say, during take-off than during taxiing. So the same tasks are executing, but are deemed to have different worst-case execution times. Fortunately this is equivalent to the tasks having added functionality and therefore modified worst-case execution times.

If any system uses mode changes in response to component failure then they are bound to use 'criticality' to decide which code to abandon and which to retain. One of the common forms of error detection is to use a watch-dog timer. If some event has not occurred by a fixed time then switch mode and protect the key computations. A task executing for longer than assumed during system verification can be identified via timers; the fault that causes the error could be in hardware or software. Here a criticality mode change and a general mode change are essentially the same thing.

A. Example of a system with both forms of mode change

Consider as an example a simplistic cruise control system that has just three modes: two normal modes, standby (SB) and speed control (SC), and one exceptional, collision avoidance (CA). The following point appertain:

- The system starts in SB with the driver in control of the vehicle.
- Movements between the SB and SC modes are normal.
- The transition to CA is operational.
- Movements between SB and SC are Bounded or Phases.
- The trigger for transition to CA is, however, Immediate.
- In all three modes a task that undertakes proximity analysis executes, this task has a reduced period in the CA mode.

The system software is partitioned between two levels of criticality: SIL4 for the safety critical functions, and SIL2 for the rest. The standby (SB) mode contains mainly SIL2 code. The collision avoidance (CA) mode has predominantly SIL4 code and the speed control (SC) mode has both SIL4 and SIL2 code in approximately equal amounts. All SIL2 code has a WCET based on extensive measurement. All SIL4 code has WCET based on pessimistic static analysis. In addition all SIL4 code also has a SIL2 estimate based only on measurement.

If one focuses on the SIL4 code, as a certification authority might, then there is a three mode system with varying amounts of SIL4 code. Similarly, from the fully functional point of view there is the same three mode system but with both SIL4 and SIL2 code.

From a mixed criticality point of view the system must be schedulable when SIL2 values are used for all code, and the system moves between the three functional modes. Additionally, the system must be schedulable in the SC and CA modes when only SIL4 code is executing and SIL4 WCET values are used.

If only Bounded or Immediate modes changes are used then the system is, at any time, only in one of three normal functional modes. This leads to explicit tests to:

- Check SB in SIL2 mode and SIL4 mode.
- Check SC in SIL2 mode and SIL4 mode.
- Check SC during transition to SIL4 mode
- Check CA in SIL2 mode and SIL4 mode.
- Check CA during transition to SIL4 mode

If however Phased changes are part of the functional design then one would have to (in addition):

- Check Phased changes in SIL2
- Check Phased changes in SIL4
- Check Phased changes with transition to SIL4

This latter case might be difficult to formulate in terms of identifying the worst-case scenario.

What this simple example indicates is that a system has orthogonal functional and criticality modes. And a system can move between functional modes, criticality modes and both at the same time. So with this example, the system could move from SC in SIL2 to CA in SIL4. But it could not move in the opposite direction. All realistic possibilities must therefore be checked as part of the system's verification.

As indicated earlier, simultaneous general mode changes are often prohibited due to the complexity they introduce. Unfortunately the introduction of orthogonal criticality mode changes has re-introduced simultaneous changes.

V. Conclusions

We have surveyed existing mode change models to provide a framework in which:

- Mode change protocols are defined to move a system between Functional modes (normal, exceptional or degraded).
- Mode change events are Immediate, Bounded or Phased.
- Each mode is defined by its tasks, and attributes such as being re-entrant, the initial mode, a sink mode or a one-shot mode (or a combination thereof).
- Tasks can exist in more than one mode, though parameters may be mode specific.
- Some tasks are mode specific.
- During a mode change, tasks may be suspended or aborted.

In the standard model of a criticality change, the proposed protocols are closest in behaviour to:

1) Graceful degradation; i.e. reduced functionality after the change.
2) Immediate or Bounded triggers, with aborted or suspended tasks.
3) Some tasks exist in both modes, but some only in the earlier mode; there are no new-mode tasks.

4) Tasks that exist in both modes may have their (worst-case) computation times increased and/or their periods decreased, and/or their criticality changed.

For papers that have attempted to define a more adaptive criticality mode change protocol, the behaviours are different:

1) The initial mode is normal, others are considered exceptional.
2) Bounded triggers are used, with suspended tasks.
3) Some tasks exist in both modes, new-mode tasks are present when changing mode in a direction toward the initial mode.
4) Tasks that exist in both modes may have their (worst-case) computation times, periods or criticality levels changed.

This difference underpins discussion that have occurred at workshops and seminars on mixed criticality. Low criticality work is still 'critical' and so cannot be abandoned lightly. The standard model appears to happily abort mission critical work. This has lead researchers to focus on adaptive schemes that minimise the harm done to this work. But the standard model does not advocate abandonment; rather it gives structural support to a form of graceful degradation following a timing error. It ensures that following a timing error the higher critical work can still be guaranteed. The more adaptive models should be seem as providing fault tolerance and error recovery.

In general, a system will be in both a functional mode and a criticality mode. But there will be some functional modes that have only one criticality; and some modes will be the target of graceful degradation both because of functional failures and execution time overruns.

Acknowledgements

The research described in this paper is funded, in part, by ESPRC (UK) grant, MCC (EP/K011626/1). The contents of this paper have benefited from fruitful discussions with Sanjoy Baruah.

REFERENCES

[1] R. Alur, A. Trivedi, and D. Wojtczak. Optimal scheduling for constant-rate multi-mode systems. In *Proc. of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 75–84. ACM, 2012.

[2] S.K. Baruah. Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *Proc. 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 31–38, 2012.

[3] S.K. Baruah. Response-time analysis of mixed criticality systems with pessimistic frequency specification. Technical report, University of North Carolina at Chapel Hill, 2013.

[4] S.K. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In A. Romanovsky, editor, *Proc. of Reliable Software Technologies - Ada-Europe 2011*, pages 174–188. Springer, 2011.

[5] S.K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

[6] S.K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.

[7] A. Burns. The application of the original priority ceiling protocol to mixed criticality systems. In L. George and G. Lipari, editors, *Proc. ReTiMiCS, RTCSA*, pages 7–11, 2013.

[8] A. Burns and S. Baruah. Timing faults and mixed criticality systems. In Jones and Lloyd, editors, *Dependable and Historic Computing*, volume LNCS 6875, pages 147–166. Springer, 2011.

[9] A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. WMC, RTSS*, pages 1–6, 2013.

[10] A. Burns and T.J. Quiggle. Effective use of abort in programming mode changes. *Ada Letters*, 1990.

[11] P. Ekberg, M. Stigge, N. Guan, and W. Yi. State-based mode switching with applications to mixed criticality systems. In *Proc. WMC, RTSS*, pages 61–66, 2013.

[12] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic task systems. In *ECRTS*, pages 135–144, 2012.

[13] P. Emberson and I. Bate. Minimising task migrations and priority changes in mode transitions. In *Proc. of the 13th IEEE Real-Time And Embedded Technology And Applications Symposium (RTAS 07)*, pages 158–167, 2007.

[14] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Proc. WMC, RTSS*, pages 19–24, 2013.

[15] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *IEEE RTSS*, pages 13–23, 2011.

[16] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. Technical Report 350, ETH Zurich, Laboratory TIK, 2013.

[17] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore, Jan 2014.

[18] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality system. In *Proc. WMC, RTSS*, pages 43–48, 2013.

[19] K. Lakshmanan, D. de Niz, and R. Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *IEEE RTAS*, pages 47–56, 2011.

[20] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *10th Euromicro Workshop on Real-Time Systems*, pages 172–179. IEEE Computer Society, 1998.

[21] L.T.X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 91–100, 2011.

[22] D. Powell. Failure mode assumptions and assumption coverage. In *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, pages 386–95. IEEE Computer Society Press, 1992.

[23] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new protocol. *Journal of Real-Time Systems*, 26(2):161–197, 2004.

[24] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 155–165, 2012.

[25] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *Proc. RTNS*, pages 183–192. ACM, 2013.

[26] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven premptive scheduling. *Journal of Real-Time Systems*, 1(3):244–264, 1989.

[27] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, pages 147–152, 2013.

[28] H. Su, D. Zhu, and D. Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Proc. RTCSA*, 2013.

[29] K. Tindell and A Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

[30] K. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptive scheduled systems. In *Proc. Real Time Systems Symposium*, pages 100–109, Phoenix, Arizona, 1992.

[31] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

[32] Q. Zhao, Z. Gu, and H. Zeng. Integration of resource synchronization and preemption-thresholds into EDF-based mixed-criticality scheduling algorithm. In *Proc. RTCSA*, 2013.

[33] Q. Zhao, Z. Gu, and H. Zeng. HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling. *Embedded Systems Letters, IEEE*, 6(1), 2014.

Mixed-Criticality Support in a High-Assurance, General-Purpose Microkernel

Anna Lyons, Gernot Heiser
NICTA and UNSW Australia
{anna.lyons,gernot}@nicta.com.au

***Abstract*—We explore a model for mixed-criticality support in seL4, a high-assurance microkernel designed for real-world use. Specifically we investigate how the seL4 model can be extended without compromising its security properties and its general-purpose nature, including high average-case performance. The proposed model introduces reservations, with admission control performed at user level, similar to how seL4 handles spatial resources.**

I. Introduction

seL4 is a high-performance microkernel of unprecedented assurance, with a machine-checked proof of implementation correctness, as well as proofs of spatial isolation properties (integrity and confidentiality) [1]. This makes it an excellent base for security-critical uses, particularly systems where security-critical components share a processor with less critical code, such as a critical crypto service co-located with an untrusted Linux system running in a virtual machine.

Our aim is to evolve seL4 into a platform for supporting similar setups in the safety-critical domain, without compromising the kernel's security properties nor its excellent performance [2]. A first step was a complete and sound analysis of seL4's worst-case execution-time (WCET) latencies [3]. The obvious next step is support for mixed-criticality scheduling, i.e. the ability to guarantee the timely execution of highly critical tasks in the presence of less critical tasks with potentially tighter timeliness requirements.

The cost of formal verification (although less than that of traditional high assurance [1]) provides a strong disincentive to maintaining multiple variants of a verified system. Furthermore, security is increasingly becoming a safety issue, as demonstrated by the recent spate of car-hacking [4]. We are therefore interested in widening the application domain of seL4, without losing any of its existing benefits.

Specifically, we are looking for a design which satisfies the following requirements:

- It preserves seL4's strong spatial isolation properties, its support for transparently interposing security monitors between communicating components, as well as its best/average case performance.
- Criticality (i.e. ability to meet deadlines) must be orthogonal to urgency (proximity of deadline), in that an over-committed system must meet deadlines of all criticality levels that would be met if none of the lower-critical tasks had been admitted. Admission control (i.e. schedulability analysis) must be possible without making any assumptions on less critical tasks.
- Tasks of different urgency and criticality must be able to share resources.
- There must be no significant (algorithmic or overhead-related) capacity loss, and any slack time must be available for best-effort tasks.
- To support certification re-use, it must be possible to admit black-box components solely based on their criticality, processor utilisation and minimal period.
- Any policy, including admission control, must be implemented at user level, the kernel is only to provide general mechanisms.
- The model must not impose restrictions on the programming model beyond what is required to satisfy all other requirements. In particular, we do not require that all shared resources are multi-threaded.

Clearly, this means that we need to provide asymmetric temporal isolation (lower criticality tasks cannot interfere with the timeliness of higher criticality tasks) enforced by runtime monitoring, with the ability to switch to a higher criticality mode of execution when the system is unable to meet all deadlines. The system should degrade gracefully in such a case, meaning that task of a certain criticality should only miss deadlines if higher-criticality tasks leave insufficient slack (i.e. we should maximize the number of high levels of criticality that meet their deadlines).

We rule out user-level, hierarchical scheduling as it introduces concurrency between user-level and the kernel. The verification of seL4 relies on the kernel remaining single-threaded to avoid the state-space explosion inherent in proofs about concurrent programs. Additionally, the current C semantics of the proof framework do not support concurrent programs [5]. However, the black-box and interposition requirements imply a requirement for delegation of CPU allocation, which we provide by leveraging seL4's capability system.

The requirements for retaining seL4's security and performance properties imply that we retain the basics of the seL4 model, which we summarize in Section II. We explore a model that satisfies the above requirements in Section III and discuss the approach to resource sharing in Section IV. We presently restrict our thinking to uniprocessor systems.

II. seL4 Basics

seL4 is a capability-based microkernel system with strong security and spatial isolation guarantees. Like other L4 microkernels, seL4 adheres to the *minimality principle* which allows features in the kernel only if the required functionality could not be achieved by a user-level implementation [6].

Specifically, device drivers are not part of the kernel, but run as unprivileged processes, the only exceptions being a timer driver and a driver for the interrupt controller.

The most significant difference between seL4 and other microkernels is its (spatial) resource-management model: The kernel, after booting up, never allocates any memory. Instead, all memory not needed to boot the kernel is handed to a user-level manager. When performing an operation that requires allocation of kernel data structures, such as creating threads or address spaces, the invoking user-level process must provide the kernel with memory for storing those data structures. Hence, all memory is completely managed by user-level code, subject to policies implemented at user level.

The kernel only supports a small number of abstractions: *threads* as the execution abstraction, *address spaces* for memory management and spatial protection, and *endpoints* for communication. *Synchronous endpoints* are rendezvous points for message-passing communication (synchronous IPC). *Asynchronous endpoints* support non-blocking signalling, they are essentially binary semaphores. Threads are tied to address spaces and communicate via endpoints. In earlier versions of L4, IPC messages were addressed directly to threads rather than endpoints. This model was abandoned as it introduces covert channels [7].

All access rights in seL4 are represented by capabilities [8], unforgeable access tokens protected by the kernel. Capabilities can be transmitted via IPC (subject to appropriate access rights) and support privilege delegation. For example, the initial resource manager can hand control over a partition of memory to another process, which then can manage that memory autonomously.

The delegatable user-level control over memory is the key to the strong, provable spatial isolation properties of seL4 [9], [10]. It is also useful for temporal isolation, as it can be used to partition caches [11], which can reduce WCET bounds [12].

IPC is also authorised by capabilities: a thread needs an endpoint capability in order to send or receive messages. Besides the simple `send()` and `wait()` (i.e. receive) operations, the kernel offers two combined send-receive operations, `call()` and `reply_wait()`.

`call()` is an RPC-like operation typically used by clients to invoke a server; it consists of a send to a specified endpoint, followed by waiting on a reply. It is semantically different from `send()` immediately followed by `wait()` in two respects: (i) the transition between sending and waiting to receive is atomic, and thus non-preemptible,[1] and (ii) instead of specifying an endpoint from which to receive the reply message, the kernel during the send phase creates a temporary one-shot endpoint for the reply, and transfers the corresponding *reply cap* to the server. Similarly, `reply_wait()` combines the reply to the caller (through the reply cap) and waiting for the next request in one atomic system call.

Management of the resource *time* is less developed, and time is in fact considered the last concept for which no satisfactory abstraction has been found to date [2]. Consequently, scheduling is deliberately left underspecified in seL4 [14]; the present implementation uses a fixed-priority round-robin scheduler.[2]

III. Proposed Scheduling Model

In order to support temporal isolation we add *reservations* to seL4. This approach had been introduced by RT-Mach [15], and later deployed in resource kernels [16]. Traditional reservations contain task scheduling parameters enforced by the kernel, specifically a limit on CPU time consumed over some interval. Additionally, the kernel performs an admission test to make sure the set of reservations is schedulable.

Mixed-criticality systems leverage the slack left from conservative WCET estimates of higher criticality tasks to run lower criticality tasks, thus increasing the overall utilisation of a system. This is achieved by allocating the excess budget of high-criticality tasks (from now on called "high tasks" for simplicity) to tasks of lower criticality. Asymmetric protection ensures high tasks meet their deadlines, even if this violates the temporal constraints of low tasks, but not vice versa. Recent models for mixed criticality systems [17] implement this through a *mode change*: if the system is unable to meet its deadlines, it increases the system criticality level, and tasks below that level are no longer guaranteed to meet their deadlines.

Our proposed model differs from traditional reservations in that we only guarantee upper bounds on execution time, and by delegating all admission control to user level.

A. Reservation capabilities

An seL4 reservation is a kernel object, and thus is represented by a *reservation capability* ("resCap"). Like any capabilities, resCaps can be easily delegated to subsystems through existing capability transfer mechanisms. A thread can only run if it is associated with a resCap, and a resCap can only be associated with a single thread at a time. Threads can share resCaps by cooperatively scheduling through IPC, as will be explained in Section IV.

Reservations act as sporadic servers [18], characterized by a budget, period and relative deadline, which encapsulates the processor share and replenishment frequency the reservation entitles. The kernel enforces budgets through a timer interrupt.

B. Scheduling

For now we retain seL4's fixed-priority scheduler, although, in order to experiment with EDF scheduling, we treat the median priority (126) special: threads at this priority use the deadline parameter for EDF scheduling (but only if no threads of a higher fixed priority are runnable), similar to Ada [19]. Reservations of EDF threads are treated as hard CBS [20].

When the current reservation's budget is depleted, it is placed into a waiting queue ordered by replenishment time, unless the reservation is a *full* reservation (100%, i.e. budget

[1] The kernel executes with interrupts disabled and limits interrupt latencies through strategically placed preemption points; none are in performance-critical IPC code [13].

[2] For security-oriented temporal isolation the scheduler is configurable with multiple non-preemtible scheduling domains, which are scheduled for a fixed time slice. These domains are unsuitable for real-time use due to the large algorithmic capacity loss and the high interrupt latencies.

= period), in which case the thread is appended to the end of its priority's scheduling queue. Full reservations preserve L4's traditional round-robin scheduling.

Obviously a thread with a full reservation should have a low priority, unless it is *trusted* not to overrun its budget, in which case a full reservation with a long period can be used to avoid the overhead of run-time monitoring.

Our model of reservations enforcing upper bounds of CPU usage encourages overcommitting, round-robin threads being an example. Schedulability analysis is a user-level concern. In fact, the kernel lacks the information to determine schedulability, as this would require locating and examining all resCaps that are associated with some thread.

C. Admission testing

Admission testing implements a particular policy, eg. on-line vs off-line, dynamic vs static, the degree of overloading allowed, and whom to trust not to overrun their reservations. According to the minimality principle it should therefore be performed at user level. Admission tests can also be very complex and hard to formally verify.

The basic safety mechanism is control over creation of reservations. We restrict this to the holder of the special `sched_control` capability, who is in complete control over time allocation in the system. The holder is trusted to perform an admission test upon a request for a reservation. seL4's startup protocol provides the `sched_control` capability to the initial process, which may then transfer it to a dedicated time manager. It may also split the total available bandwidth and delegate partitions to individual managers, which achieves most of the benefits of hierarchical scheduling without its cost.

This approach is analogous to seL4's mechanism for controlling memory, where the initial process obtains rights to all free memory. It is also similar to how seL4 manages access to devices: the holder of a special `IRQ_control` capability grants device drivers the rights to specific interrupts. On seL4, all resource management is performed by trusted user-level servers, and time is no longer an exception.

Schedulability depends on priorities as well as reservations. The system provides a safety mechanism by associating each thread with a *maximum controlled priority* (MCP). While a holder of a thread capability can control that thread's priority, the kernel will not allow it to raise any thread's priority (including its own) to a value higher than its own MCP.[3]

D. Task Model

We adopt the sporadic task model, where tasks are an infinite series of jobs. A task is represented by an seL4 thread, and a job is the release of a thread by the kernel.

A thread has an optional asynchronous *trigger endpoint*; by waiting on that endpoint, the thread indicates *job completion*. A thread that does not complete is rate-limited by its reservation.

Job release happens by signalling that endpoint, thus resuming the thread's execution. The kernel signals the endpoint when the thread's budget is recharged, thus supporting time-triggered tasks. Alternatively the endpoint can be signalled by some event, e.g. an interrupt or another thread, resulting in an event-triggered task. Such a thread does not actually become runnable until its recharge time has passed (until that occurs, it has no budget to run).

The kernel has no concept of threads being real-time or not: whether a thread is able to meet its deadlines solely depends on whether the thread's budget is sufficient for its WCET, and whether the system is over-committed at the thread's priority.

E. Criticality

We add a criticality field to seL4 threads, and track a global kernel criticality level. The criticality level is changed at user-level by invoking the `sched_control` capability. Threads whose criticality is less than the global kernel criticality will not be scheduled: instead, they are post-poned by the period of their reservation, at which point the criticality level may have changed. This approach maintains the preemption level of the lower criticality workload, but allows threads to come back online automatically once the criticality level is restored.

F. Mode changes

To enable the mode change required for mixed-criticality support, we introduce a simple, policy-free mechanism: *temporal exceptions*. This extends the existing seL4 exception handling approach, which associates an exception endpoint with each thread. When a thread triggers an exception, the kernel sends a message to the exception endpoint. A handler thread waiting on that endpoint can then handle the exception. In a practical system, many threads share the same exception endpoint (and thus handler), typically the responsible operating-system personality.

For temporal exceptions we introduce a second, optional, temporal exception endpoint. The kernel sends a message to this endpoint if the thread exceeds its budget or overruns its deadline. If the thread has no temporal exception endpoint, it is silently rate-limited. The handler, assumed to be a highly-privileged thread, can then transition the system into high-criticality mode.

How the handler responds to the exception depends on the policy of the system. Some systems may have infrequent and short mode changes, where all lower criticality threads should be briefly suspended until the system returns to normal. In this case, using the kernels criticality mechanism is suitable: the overruning thread's budget can be increased to parameters for a higher criticality mode, and the kernel criticality level increased. Alternatively, if the system requires that lower criticality threads remain runnable but with weaker or no guarantees, the exception handler can reduce the priorities of lower criticality tasks [21], or give high tasks full reservations and boost their priorities. Under any mode switch policy, the exception handler needs its own (high-priority) reservation, which must be factored into the cost of the mode switch.

The opportunity to return to a lower criticality level can be detected by using a dedicated thread running at a priority below that of all threads at or above the current criticality level, but above the (down-graded) priority of all low threads (should

[3]Note that a thread's actual priority can exceed its MCP, provided it has been set by another thread with a sufficiently high MCP.

they be runnable). When the kernel schedules this thread, it is an indication that there is slack in the system, and the thread can move the system toward normality by restoring scheduling parameters or increasing the kernel criticality level.

IV. Resource Sharing

The frequently made assumption of no sharing across criticality levels is unrealistic [21]. For example, the low-level flight control of a unmanned aircraft (UAV) is highly critical, as it ensures the vehicle remains stable and on track, its failure would lead to loss of the UAV. The UAV's mission control determines, in communication with the ground station or based on analyzing senor input, where the vehicle is to go next. It is less critical, as ground control can re-transmit commands or the analysis can be repeated. But, in order to be effective, mission control must share resources with flight control, e.g. the way points updated by mission control and used by flight control.

By definition, sharing implies that a high task may be blocked while a low task is holding a resource. A shared resource must therefore be considered to have the same criticality as its highest client, including a WCET certified at the level required for that client. We furthermore require a mechanism that allows the high task to progress if the low task runs out of budget while holding the resource.

In seL4 we model shared resources as *resource servers* accessed via synchronous IPC [22]. We distinguish between *active* servers, which have their own reservation, and *passive* servers, which do not. A passive server can only execute by another thread *transferring* its reservation to the server. Such a transfer happens during synchronous IPC: when a client invokes a server (via a `call()` IPC operation), its reservation is transferred to the receiver, and the server returns it when completing (via the `reply_wait` operation), see Figure 1. Such a server is said to execute on a *borrowed* reservation.

This is similar to time-slice donation in earlier L4 versions [23], with one crucial difference: a reservation will only transfer if the receiver does not already have a reservation (a passive server or a thread which has transferred away its reservation). That way, all of a passive server's execution time is forced to be accounted against a client-provided reservation, while an active server will always execute on its own reservation. Both cases enforce temporal isolation between clients.

Reservation transfer avoids invoking the scheduler or updating accounting parameters, key properties for maintaining seL4's highly-efficient IPC. But we obviously need to consider budget expiry and mode changes.

A. Priority Inversion

Resource servers are critical sections, which means to maintain system schedulability we must provide a mechanism to avoid unbounded priority inversion. Priority inheritance (besides its other drawbacks such as implementation complexity and long worst-case blocking times) is infeasible to implement in a security-oriented model of IPC being mediated by endpoints: the kernel has no knowledge of who will be receiving messages sent to a specific endpoint, and thus cannot determine which thread should inherit the priority of the sender thread blocked on the endpoint. Similar comments apply to the original priority-ceiling protocol.

Instead we provide the means for user-level code to implement basic priority ceilings, following highest locker's protocol (HLP), where resources are assigned ceiling priorities and tasks that acquire a resource run at the ceiling priority immediately. HLP is used in POSIX for `PRIO_PROTECT` with one key difference, while POSIX runs the task at the highest priority of any resources held, our model assumes that nested resource access will be in ascending priority order. The kernel mechanism for this is simple: even a passive server has a defined priority, at which it executes irrespective of the priority of the thread whose reservation the server borrowed. A correct system configuration then requires that resource servers are given the correct ceiling priority. (Note that user-level can, in principle, do this assignment automatically: only clients who have a `send` capability on the server's request endpoint can invoke the server. The resource manager which distributes these capabilities can adjust the server priority to the maximum of the priorities of all clients to which it hands the server's request endpoint capability.)

B. Budget Expiry

If the budget of a server's borrowed reservation expires before the server completes the request, the server is left in a state where it cannot serve other client's requests until the borrowed reservation is replenished. This constitutes a potential criticality inversion, where a high thread must trust that any low thread invoking the server does it with sufficient budget, obviously not an acceptable situation.

The *helping* approach taken by Fiasco [23], where clients donate budget to the blocked thread to get it out of the server, does not work in the security-oriented IPC endpoint design: The kernel has no way of knowing on which endpoint the server will attempt to receive next, and thus cannot determine the helper.

Temporal exceptions are a suitable mechanism for recovering from this situation. When the reservation expires, the kernel sends an exception message to the *owner* of the reservation (i.e. the thread to which the reservation was allocated, ignoring any borrowing). The temporal exception handler is then responsible for the recovery action. Possible actions include giving the faulter an emergency budget or resetting the server back to a defined state (ready to receive further requests) and sending an error replying to the client on the server's behalf.

The exception handler has its own reservation, which must be sufficient to implement the policy required by that server. Note that the required budget can be quite large, if the number of a server's low clients is large, and it must be replenished at the highest rate of all clients. Clearly, cross-criticality resource-sharing must be done wisely. seL4's protection mechanisms help limit such sharing, by controlling the distribution of capabilities to server request endpoints.

C. Mode change

Mode changes can occur while a shared resource is being accessed, specifically while threads are enqueued on the resource endpoint or actively using the resource. We lazily detect

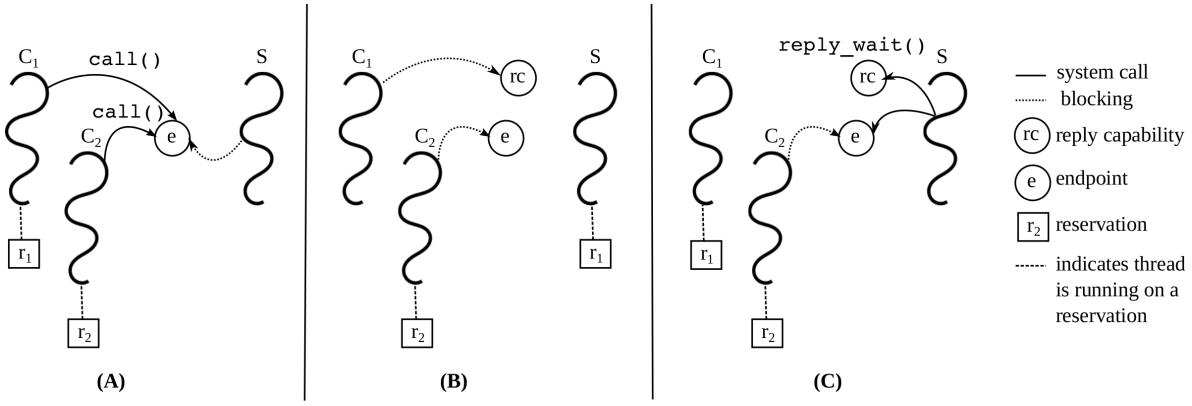


Fig. 1. Client threads invoke a passive server via IPC on endpoint e. In **(A)**, two clients, (C_1 and C_2 with reservations r_1, r_2) both send requests to the server S via `call()`. In **(B)**, C_1's message is processed first: the kernel generates a one-shot endpoint (rc) that C_1 blocks on, and the server borrows C_1's reservation r_1 while running on C_1's behalf, while C_2 remains blocked on the endpoint. **(C)** shows S completing the invocation using `reply_wait()` on e, transferring r_1 back to C_1 over rc. Note that the system is strictly speaking never in the state shown in **(C)**, as the `reply_wait()` operation is atomic, so S switches directly from the reply to C_1 (through rc) to receiving the message and reservation from C_2.

if threads queued on an endpoint have sufficient criticality: if a high-criticality server attempts to receive a message and the client has insufficient criticality, it will be removed from the endpoint queue and post-poned. The IPC operation will restart when the client is scheduled after the kernel criticality level has been raised. Threads actively using a resource during a criticality change are detected when they are next scheduled: the kernel detects that the server is running on a reservation belonging to a thread with an insufficient criticality level, and sends a temporal exception to the servers temporal exception handler, which can reset the server.

Of course, the approach described above works only for systems using the kernel criticality level to implement mode changes. Other mode change policies involve client priorities being lowered or raised, and/or reservation parameters changing. Endpoint queues are reordered on priority change, and tasks that are suspended have pending IPC messages cancelled, while changing reservation parameters has no effect on the endpoint queues, but will result in an exception triggering the budget expiry handler if a thread no longer has budget to complete a resource request.

A server's borrowed reservation may run out of budget after a mode change, resulting in a temporal exception. As the server runs at the ceiling priority, which should be unaffected by the mode change, a change of the client priority will not take effect until the server replies to the client. This increases the worst-case cost of the mode change.

We observe that handling of a temporal exception depends greatly on circumstances: An exception triggered by a low thread may simply be ignored, resulting in rate-limiting. If the low thread's budget expires while borrowed by a server, a reset action may be required. If, however, a high thread's budget expires, this may require a mode switch. This means that the handler needs sufficient information to determine the course of action. To solve this, we allow a data word to be set in each scheduling context which is delivered with the temporal fault message. Systems can set this data word to identify the client, or the clients criticality, within the temporal fault handler.

D. Summary

Our kernel changes in total account for a 2045 LoC patch[4] This includes the addition of a release queue of pending and rate-limited jobs, reservations, criticalities, improved timer driver and modifications to the IPC path.

V. Related Work

Traditional resource kernels [24] support slack reuse but do not guarantee deadlines of low-criticality tasks even if this does not prevent high tasks from timely execution. Burns and Davis [17] present a detailed survey of mixed-criticality systems research. The systems closest to ours in their aims are COMPOSITE and Fiasco.

COMPOSITE [25] completely frees the kernel from any scheduling policy by providing mechanisms for hierarchical user-level scheduling. It reduces overhead-related capacity loss by configuration buffers shared between user-level and the kernel. Some capacity loss remains as timer interrupts must be delivered down the scheduling hierarchy. This approach does not suit seL4, as the required reasoning about concurrent access (by kernel and user-level) to those buffers would drastically increase verification overhead [1]. Unlike all L4 microkernels, COMPOSITE implements a migrating thread model [26]. This implies that access to shared resources does not block, thus avoiding priority inversion, although at the cost of requiring all server code to be re-entrant, a requirement we do not want to impose.

A version of Fiasco [23] uses bandwidth inheritance [27] over IPC, which is analogous to priority inheritance. For security reasons, Fiasco has also moved to IPC mediated through endpoints, so this approach does not work in later versions of the kernel.

Brandenburg introduces an IPC protocol for clustered multicore mixed criticality systems using EDF and CBS, using multiple IPC queues to separate critical real-time and non-critical background tasks [22]. As it uses unmediated IPC,

[4]Counted by David A. Wheeler's "SLOCCount".

their approach does not directly apply to seL4. They avoid mode changes by servers prioritizing high clients irrespective of scheduling priority, and resetting a server on budget expiry.

Quest-V [28] is a separation kernel which can be used to sandbox tasks of different criticalities, allowing them to safely share hardware, however has no support for mode changes and thus offers no utilisation increase. Lackorzynski showed that to virtualise multiple mixed criticality RTOSes, information must be passed between the guest and host about mode changes to avoid violating the schedulability guarantees of either guest, and implemented this in Fiasco.OC [29]. An implementation of mixed criticality systems in Ada, demonstrates reordering of priorities on mode change [30].

Recent proposals adapt the original priority-ceiling protocol to mixed criticality [31], [32], but are unsuitable for us as explained in Section IV-A.

VI. Conclusions & Future Work

We have outlined a model for supporting mixed-criticality scheduling in seL4. The model supports cross-criticality resource sharing and mode switches, while retaining seL4's security properties and high average-case performance.

We have a mostly complete implementation and are presently working on evaluating it by building practical mixed-criticality systems on top, including a UAV and a space satellite. This will be the real test of the practicality of the proposed approach. In particular, we need this practical experience to determine the best approach to the (user-level) implementation of mode switches and temporal exception handling.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *Trans. Comp. Syst.*, vol. 32, pp. 2:1–2:70, Feb 2014.

[2] K. Elphinstone and G. Heiser, "From L3 to seL4 – what have we learnt in 20 years of L4 microkernels?," in *SOSP*, (Farmington, PA, USA), pp. 133–150, Nov 2013.

[3] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *32nd RTSS*, (Vienna, Austria), pp. 339–348, Nov 2011.

[4] C. Smith, *Car Hacker's Handbook*. 2014.

[5] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating system kernel," *CACM*, vol. 53, pp. 107–115, Jun 2010.

[6] J. Liedtke, "On μ-kernel construction," in *15th SOSP*, (Copper Mountain, CO, USA), pp. 237–250, Dec 1995.

[7] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *IEEE Symp. Security & Privacy*, (Oakland, CA, USA), May 2003.

[8] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *CACM*, vol. 9, pp. 143–155, 1966.

[9] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd ITP*, vol. 6898 of *LNCS*, (Nijmegen, The Netherlands), pp. 325–340, Aug 2011.

[10] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *IEEE Symp. Security & Privacy*, (San Francisco, CA), pp. 415–429, May 2013.

[11] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of some timing channels on seL4," in *ACM Conference on Computer and Communications Security (CCS)*, (Scottsdale, Arizona, USA), Nov 2014.

[12] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *19th RTAS*, (Philadelphia, PA, USA), pp. 45–54, Apr 2013.

[13] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *7th EuroSys*, (Bern, Switzerland), pp. 323–336, Apr 2012.

[14] S. M. Petters, K. Elphinstone, and G. Heiser, *Trustworthy Real-Time Systems*, pp. 191–206. Signals & Communication, Jan 2012.

[15] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: An abstraction for managing processor usage," in *Proceedings of the 4th Workshop on Workstation Operating Systems*, pp. 129–134, 1993.

[16] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: a resource-centric approach to real-time and multimedia systems," in *Readings in multimedia computing and networking*, pp. 476–490, 2001.

[17] A. Burns and R. Davis, "Mixed criticality systems – a review." http://www-users.cs.york.ac.uk/~burns/review.pdf, Jun 2014. Online; accessed 29-Sept-2014.

[18] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling sporadic and aperiodic tasks in a hard real-time system," technical report CMU/SEU-89-TR-011, Carnegie Mellon University, Software Engineering Institute, Apr 1989. URL resources.sei.cmu.edu/library/asset-view.cfm?assetid=10919.

[19] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. 2007.

[20] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *J. Real–Time Syst.*, vol. 27, no. 2, pp. 123–167, 2004.

[21] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Proceedings of the 1st Workshop on Mixed Criticality Systems*, pp. 1–6, 2013.

[22] B. B. Brandenburg, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *35th RTSS*, (Rome, Italy), Dec 2014. To appear.

[23] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT*, (Brussels, Belgium), Jul 2010.

[24] S. Oikawa and R. Rajkumar, "Linux/RK: A portable resource kernel in Linux," in *19th RTSS*, 1998.

[25] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *29th RTSS*, (Barcelona, Spain), Nov 2008.

[26] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *OSPERT*, (Brussels, Belgium), Jul 2010.

[27] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *Trans. Computers*, vol. 53, pp. 1591–1601, Dec 2004.

[28] Y. Li, R. West, and E. S. Missimer, "The Quest-V separation kernel for mixed criticality systems," in *1st WMC*, pp. 31–36, Dec 2013.

[29] A. Lackorzynski, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *EMSOFT*, (Tampere, Finland), pp. 93–102, Oct 2012.

[30] S. Baruah and A. Burns, "Implementing mixed criticality systems in Ada," in *Proceedings of Reliable Software Technologies – Ada-Europe*, pp. 174–188, 2011.

[31] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," pp. 7–11, 2013.

[32] Q. Zhao, Z. Gu, and H. Zeng, "HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling," *Embedded Systems Letters, IEEE*, vol. 6, pp. 8 – 11, Jul 2013.

On Spatial Isolation for Mixed Criticality, Embedded Systems

Eric Armbrust, Jiguo Song, Gedare Bloom, Gabriel Parmer
The George Washington University
Washington, DC
{earmbrust,jiguos,gedare,gparmer}@gwu.edu

***Abstract*—This paper addresses some of the challenges of creating a system that enables not only the temporal isolation required for mixed-criticality systems, but also the necessary spatial isolation that enables the decoupling of assurance levels required for different pieces of software. We discuss the application of fine-grained isolation, hierarchical resource management, and the paravirtualization of a legacy RTOSs API, all to enable the system designer to harness memory isolation to control the assurance required for system components.**

I. Introduction

Real-time / embedded system developers face increasing pressure to reduce the size, weight, and power (SWaP) requirements of devices. One solution to reduce SWaP is to package multiple applications onto a single chip and to partition access to the chip resources among the applications. When such applications have differing safety-critical importance, the integration of these applications on a shared platform creates a mixed criticality system (MCS). A problem with MCSs is in sharing resources between applications at different criticality levels, because blocking synchronization primitives can lead to low-criticality tasks interfering with high-criticality tasks. When the MCS is scheduled globally, *i.e.* the same scheduler handles all tasks, solutions based on priority-based synchronization primitives can be applied—for example, criticality-aware versions of the priority inheritance and priority ceiling protocols [1], [2]. However, if the MCS lacks a global scheduler then the job of ensuring that resource synchronization does not cause low-criticality tasks to interfere with high-criticality tasks becomes more difficult. In particular, MCSs that use hierarchical scheduling [3] do not have global scheduling knowledge.

When used with only two levels, a parent scheduler and its children schedulers, a hierarchically-scheduled MCS schedules applications at different criticality levels with distinct children schedulers. The parent scheduler is trusted to schedule the children according to criticality, and each child schedules the application tasks independently. A popular mechanism to support two-level hierarchical scheduling is to use virtualization, with the parent executing in the hypervisor and each child in a guest virtual machine. Applications can be used with minimal modifications and the hypervisor ensures safety of the high-criticality tasks. A problem with hierarchical scheduling with virtualization technology is that performance degradation is prohibitive when children are given small budgets, for example in RT-Xen budgets less than 1 ms lead to untenable scheduler overhead [4]. Another problem with using hierarchical scheduling for MCS is that the existing solutions for task synchronization cannot be adopted easily for resource sharing, because there is no uniform scheduling policy to arbitrate priority and criticality between different schedulers in the hierarchy.

In this paper, we introduce MC-HiRes, Mixed Criticality HIerarchical RESource management, which supports MCSs in the Composite operating system via hierarchical scheduling with the benefits of minimal modification of applications, small performance loss, and resource sharing between applications at different criticality levels. MC-HiRes is a logical extension of our HiRes [3] system with support for MCSs. The primary modification to HiRes is support for a range of mappings of event notification threads (ENTs) to nodes (components) in the hierarchy to provide for strong temporal isolation even in the presence of shared resource synchronization. MC-HiRes avoids the performance problems of virtualization-based hierarchical scheduling by using a guest-aware approach requiring minor modifications *i.e.* paravirtualization. The problem of task synchronization in a hierarchically-scheduled MCS is handled by locating synchronization primitive (locks) in servers that mediate access to shared resources among the clients, which may have different criticalities. Subject to the server having the highest criticality of its clients, MC-HiRes thus solves the two problems identified above for an MCS using hierarchical scheduling.

We demonstrate MC-HiRes by paravirtualizing a legacy RTOS, FreeRTOS, to split RTOS services and application threads into separate components. Isolating application threads enables system designers to assign different criticalities to threads, thus allowing an MCS design without modifying application software. (Slight modification is made to the FreeRTOS kernel.) We evaluate the overhead of using semaphore and message queue services within the FreeRTOS kernel and between application threads and the kernel. Performance loss occurs due to the introduction of spatial isolation along RTOS service calls, but the performance is reasonable with an overhead around 0.25 μ-seconds each time a call is made between criticalities.

Contributions. This paper's contributions include:

- An introduction to the component-based system structuring model, its uses, and the implications for both temporal and spatial isolation of mixed criticality embedded systems.
- A simple extension to our existing hierarchical resource management system (HiRes) to enable hierarchical scheduling for MCS to ensure the mutual temporal/spatial isolation of different criticalities.

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675 and ONR Award No. N00014-14-1-0386. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

- A system design that provides isolation of mixed-criticality applications from each other for a legacy embedded system that does not provide memory protection, using component-based memory isolation facilities of COMPOSITE.

II. MC-AWARE MEMORY ISOLATION IN COMPOSITE

The COMPOSITE component-based OS provides support for memory protection between separate user-level components based on hardware page-tables. Each component includes local memory for data, code, heap, and stacks that is isolated from other components. Each component exports zero or more interfaces consisting of a number of function call entry-points, and has a set of dependencies on the interfaces exported by other components. Components are passive: only if a thread is explicitly created within the component, or if a thread executing in another component invokes an exported function, will execution occur in a component.

Inter-component communication. Invocations of a function exported by a component result in IPC via *thread migration*. The same schedulable thread executing in the client, resumes execution in the server. After completing its execution in the server, it resumes execution in the client. Control flow integrity is maintained as the server controls its entry-points, thus ensuring that only intended functionality is conducted. Upon entry into the server, sanity checks akin to those done by a traditional kernel in system-call handlers are conducted on parameters. The scheduling context migrates between components (thus scheduling components control thread, not component, execution), it switches between component-local execution state, including execution stacks. These stacks are managed [5] to trade-off the amount of memory they require, and the timing properties of the system if multiple schedulable threads require more stacks than are available in a component (mediated by predictable sharing protocols). In this paper, we assume a simple static allocation of stacks to components commensurate to the number of threads in the system. The sum benefit of this inter-component communication mechanism is that the *end-to-end timing analysis of a thread are identical to those in traditional systems*. This explicitly avoids the dependent-task scheduling problems often encountered by IPC mechanisms that involve coordination between multiple threads. These existing analyses are often pessimistic, especially in a system with large numbers of components.

Resource sharing. Note that resource sharing must still be taken into account, but the prescribed mechanism for this is to mediate all sharing of a specific resource within a critical section within a component. For example, COMPOSITE has a mailbox component that enables multiple threads to communicate via asynchronous message passing. The buffers that hold the data being passed between threads are mediated within the component via a critical section protected by a lock supporting predictable resource sharing. By default we use a lock component that provides priority inheritance. For an MCS, the lock component is problematic because all threads of different criticalities are exposed to each other's interference within the server, and within the critical section. However, *all other execution in other components can be segregated between criticalities*.

Resource sharing mediated by memory-protected components simplifies the analysis of the system. The worst-case hold time of any shared resource (the worst-case critical section length) is provided by the implementation of the mediating component. That component's code is trusted given a combination of the control flow integrity of the IPC mechanism, and its memory isolation. Assuming the component has code appropriate for the criticality of all of its clients, the impact of the resource sharing on each of their timing is *invariant on the clients, and only a property of the mediating component itself*. This simplifies sharing between criticality levels by design, and enables traditional protocols such as Priority Ceiling Protocol to provide predictability, though MC-specific resource sharing protocols can be used (by simply using a different lock component).

Though implementing the sharing of resources in components implies the overhead of communication with that component, round-trip IPC (called "component invocation" here) in COMPOSITE is as efficient as the fastest IPC implementations, and is on the order of 600 cycles on our Intel i7-2760QM CPU running at 2.4Ghz (*i.e.* 0.25 μ-seconds).

A. Mixed-Criticality-Aware Memory Isolation in COMPOSITE

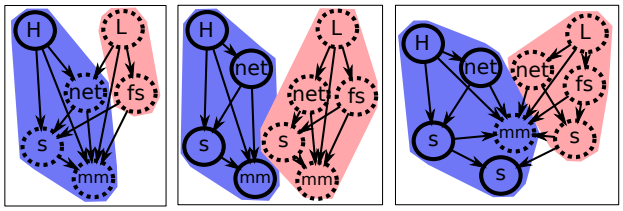


Fig. 1: Example system component structures (S component = scheduler, mm = memory manager). Arrows are component dependencies, and dashed components are harnessed by the low-criticality application. Thus, they require assurance under more complex workloads and must handle more features. The blue region requires a higher level of certification, than the red. Ideally, the blue region would contain only non-dashed, simple components. (a) Traditional structure where high (H) and low (L) criticality applications share all system services. Significant portions of the system requires both full featured support for L *and* high assurance of those components. (b) Separation kernel-style isolation. Only services specialized to H require high levels of assurance, but sharing between H and L is difficult. (c) Selective sharing of functionalities between applications controlling the assurance level required vs. complexity of each component, and hierarchical resource management.

The COMPOSITE thread migration-based inter-component communication mechanism enables the use of traditional scheduling analyses that operate on threads (rather than components, or component-specific threads) and that consider critical sections using predictable resource sharing protocols. The major implication is that thread migration enables the fine-grained decomposition of the system into components, thus strengthening memory isolation. In COMPOSITE, even the lowest-level system services are implemented as components, including: the schedulers, lock managers, time managers, physical memory management, file systems, networking, and drivers. Each component is independently redeployable (assuming its interface dependencies are satisfied), and the entire system can be viewed as a graph of components.

The combination of fine-grained components, the component definition of low-level services, and the use of traditional end-to-end thread timing analysis together provide the ability to explicitly design the structure of the system to mimic that of the criticalities of the different components in the system. The criticality of each component might be static—dependent on off-line tests and analysis of the component's code—or might be dependent on what other component's depend on it, i.e. the workloads for which it is tested. One thing is clear: the high-criticality applications must depend only on high-criticality components. More specifically, the transitive closure over the dependency relation seeded with the high-criticality applications must contain only components that are high-criticality. Importantly, this enables the system to minimize the amount of software that requires high-criticality certification. Whereas in traditional systems that include many system services in the same memory protection domain, all of which must be certified to the highest criticality level, COMPOSITE enables the decoupling of memory and temporal interference between different services.

What is the "correct" amount of sharing between the component graphs for different applications? Figure 1 depicts a simple embedded system with each policy implemented as a separate component. The three different configurations of the system represent (1) a traditional system structure in which applications of different criticalities share many components, requiring that they be certified to the confidence-level of the highest application, (2) a separation kernel-like system in which different criticalities share as few components as possible, and (3) a system with nuanced sharing of components between criticality levels dependent on the sharing relationships and resource availabilities of the system. The benefit of avoiding separation kernel-like share-nothing system organizations is exactly the fact that it is convenient for criticalities to share information and resources between each other (*e.g.* a hard real-time subsystem sharing data to be displayed by an interface). In resource-constrained embedded systems, the extra memory required for separate images and data structures for components replicated between criticalities can be undesirable.

Summary. COMPOSITE enables the straightforward end-to-end analysis of the timing of threads that execute across many components, thus enabling the fine-grained decomposition of the system software into memory-isolated protection domains. Sharing is mediated by service components, and the interference between threads (of different criticalities) is dependent on the mediating component's properties, and is not variant on the contending threads—the critical section interference stays the same, regardless. The criticality and structure of the system's components can be configured to minimize how much software requires high certification, mirror the sharing requirements of applications, and appropriately trade resource usage. This configurability is where the non-traditional, flexible means of constructing a system of components, combined with the fine-grained memory isolation, provides significant benefit for an MCS.

III. MC-AWARE HIERARCHICAL RESOURCE MANAGEMENT

Defining resource management components for a single resource, such as CPU scheduling, across different criticalities is difficult in a system structured as in Figure 1(b). Using CPU management as an example, each scheduler in the system contends for the processor, and the scheduler that should control the CPU at any point in time is not clear. The tension between decentralizing resource management—to customize it for different criticalities, and to increase isolation between them—and deciding at any point which manager to give access to the CPU, motivates our previous work on HIRES [3]. HIRES provides a set of protocols to enable resource manager coordination for hierarchical resource management for CPU, memory, and I/O. In this paper, we focus on extending such protocols to support CPU management and scheduling components for MCSs.

Hierarchical scheduling enables multiple concurrent scheduling components in the system. Though in the simplest case, schedulers form a tree of arbitrary depths, HIRES supports a directed, acyclic graph (DAG) of scheduler structures as well. Each (child) scheduler receives execution time from parent schedulers, and the root scheduler *delegates* all time to its children. HIRES enables each scheduler, regardless of how deep it is in the hierarchy, to dispatch between threads (and even schedule interrupts) with constant and comparable overhead. As each scheduler is implemented as a separate component, they benefit from memory isolation. Thus, possibly complex scheduling policies with dynamic workloads for low-criticality applications can be removed from the certification burden of the high-criticality domain by relying on a simple parent scheduler to mediate between criticalities (e.g., the bottom scheduler in Figure 1(c)).

HIRES Scheduler Coordination Protocols. Parent schedulers delegate to children using a simple mechanism. Parent schedulers are aware of, and uncommonly may dispatch directly, child threads. Normally a parent activates or deactivates a child thread by dispatching to a single *Event Notification Thread* (ENT). That thread executes in a loop delivering event notifications from parent to child. The parent sends a number of notifications: (1) child thread has blocked within an ancestor component, (2) child thread has been activated within an ancestor, (3) a given quantity of time has passed since the last notification—used as a timer for the child scheduler, and (4) the amount of time since the child was last executed (so that it can maintain proper accounting). The child scheduler, after processing all notifications, resumes its normal scheduling behavior and chooses a thread to dispatch.

Child schedulers use the same ENT to send notifications and requests to the parent: (1) thread creation and deletion requests, (2) timeout requests to block until the next notification or given timeout, (3) idle requests (block with an infinite timeout). (For details on the implementation of the protocol for how parent and children synchronize see the original HIRES [3].) The overhead of the ENT and parent-child coordination is on the order of two thread dispatch latencies (to the ENT in the parent, and away from it in the child), plus a single component invocation. The overhead is

around 1700 cycles in total.

HIRES and child thread scheduling. All parent schedulers are responsible for protecting their own data structures, thus they use critical sections. Though parent schedulers rarely dispatch to child threads, in the case of contended resources, parents do not rely on children to mediate the contention. Doing so would put the timing properties of the parent scheduler at the mercy of the child. Thus, in this case of contention, the parent scheduler will switch directly to the child thread that holds the critical section to grant the higher-priority contending thread access. This switch is essential to prevent low-criticality child schedulers from changing the timing properties of the parent scheduler, thus indirectly impacting the timing of high-criticality applications. The policy that the parent scheduler must control its own timing independent of the behavior of any child scheduler is consistent with the resource sharing between criticalities discussion in Section II—the parent makes timing guarantees for critical section length.

Blocking/waking threads in parent schedulers. Given the flexibility of component composition in COMPOSITE, a situation may arise in which a thread managed by a child scheduler will invoke a component that will attempt to block it (*e.g.* due to resource contention). Each component invokes a specific scheduler, and if the service is low-level it might invoke the parent scheduler to block the thread. This situation—and the one originating from the same component waking the thread at a later time—requires coordination between parent and child scheduler. In HIRES, this is where the parent notifications to the child that the thread has been blocked/woken are relevant. Note that this coordination between parent and child scheduler is lacking in user-level threading libraries as monolithic kernels are not aware of user-level threads. Any one thread will block all of them. Systems such as scheduler activations [6] attempt to solve this problem in a manner similar to HIRES. As both parent and child know of the existence of each thread, the HIRES protocols focus on enabling them to coordinate to schedule the threads appropriately.

A. MC-HIRES*: Mixed-Criticality-Aware, Hierarchical Resource Management*

Hierarchical scheduling and mixed-criticality workloads can often be ill-matched [7]. For example, a child scheduler controlling applications of a comparable critical level might need to execute threads at different priorities (there is no fixed relation between criticality levels and priorities). Thus, a single ENT that the parent dispatches to activate the child is insufficient. That single ENT is treated as a single thread by the parent, thus providing abstraction in hierarchical scheduling. This abstraction prevents multiple priorities and criticalities to be attributed to the child.

We generalize the HIRES model into MC-HIRES. MC-HIRES can describe the traditional setup of hierarchical scheduling with all execution in the child abstracted behind the parameters of the ENT, no hierarchical abstraction with one ENT for each child thread, and any configuration of ENTs to children threads in between these extremes. Each child thread is associated with an ENT when it is created, and all notifications for that thread are sent via that ENT. The parent scheduler schedules ENTs as normal threads, thus activating the child according to the parameters of each ENT.

IV. CASE STUDY: LEGACY RTOS MC-AWARE MEMORY ISOLATION

As an example of some of the techniques discussed in this paper, we have modified a popular, simple RTOS to execute in a paravirtualized environment in MC-HIRES, and have used component-based techniques to provide temporal and spatial isolation for MCSs.

A. FREERTOS *Background*

FREERTOS is a simple RTOS used in (deeply) embedded systems for its configurability and small footprint. It is simple and includes basic APIs for thread creation, message queues (synchronous or asynchronous), semaphores, memory allocation, and some facilities for sleeping threads to enable periodic behaviors. Scheduling is fixed priority, preemptive. All threads share the same protection domain, and FREERTOS is meant to execute on the bare metal (though ports exist that execute on POSIX).

FREERTOS is not a RTOS that is MC-friendly. Applications are not spatially isolated from each other, nor is the kernel code and data. Though the system does support temporal isolation between threads with predictable sharing of resources using fixed-priority, preemptive scheduling, it does not provide the necessary memory protection (spatial isolation) to enable the separate certification of code of different criticalities.

B. MC-FREERTOS*: When Virtual is Better than Real*

To enable legacy embedded tasks to execute within the context of an MCS, we provide means for both spatial and temporal isolation for FREERTOS. To accomplish this, we provide a series of modifications to FREERTOS to increase its capabilities within an MC environment, yielding MC-FREERTOS.

First, we paravirtualize FREERTOS to execute within a component in COMPOSITE. This extension is straightforward and involves adding a FREERTOS port that uses COMPOSITE scheduler library functions for switching between threads and for disabling interrupts. The FREERTOS component is a scheduler, thus has permission to dispatch between threads that can migrate between components via invocation. Thread dispatching involves making a COMPOSITE system call. *MC benefits:* FREERTOS and all of its tasks are now spatially-isolated from other components in the system, thus effectively enabling software of different criticalities.

Second, timer interrupts within FREERTOS are implemented using an MC-HIRES ENT. This enables FREERTOS to be integrated into the scheduling hierarchy. Now existing high-criticality tasks can be executed in MC-FREERTOS, while component-based applications with a lower level of assurance can be executed in other subsystems under different schedulers. A root scheduler that implements a simple policy (therefore capable of being high-criticality) schedules the various criticalities. In our prototype, the root scheduler is a simple fixed priority preemptive scheduler with MC-FREERTOS executing at the highest priority. *MC benefits:*

Multiple criticalities can exist in different scheduler subsystems. FREERTOS legacy applications are spatially-isolated (similar to a separation kernel) from other applications, although not from each other.

Third, we paravirtualize the API of FREERTOS to enable multiple criticalities even for legacy FREERTOS applications. Some subset of FREERTOS applications execute in a memory-isolated component, yet still harness the functionality of the FREERTOS kernel. *MC benefits:* Within the MC-FREERTOS environment, multiple criticalities can exist. More importantly, applications can be written that utilize both the FREERTOS API, and can access non-FREERTOS components. The following text describes this technique.

System call API via namespace virtualization. FREERTOS does not have a well-defined system-call layer like OSes that use dual-mode protection. The lack of memory isolation removes the motivation to define such a layer. However, FREERTOS does have a well-defined API, which applications are intended to use, that features functions of the main functionalities (thread manipulation, queue usage, semaphores, timed blocking). The second stage of converting this legacy system into one that is MC-capable is that we paravirtualize this API so that a FREERTOS application can be executed in a separate component (thus separate protection domain). Application code in the FREERTOS application component is identical to that linked into the FREERTOS kernel, except that it is linked with a small FREERTOS-lib that exports the FREERTOS API. That library interfaces with the IPC facilities of COMPOSITE, and invokes functions exported by a FREERTOS-klib (kernel library) linked into the FREERTOS kernel component. The FREERTOS-klib invokes the actual methods within FREERTOS to handle the requests.

The main functions of the two libraries are to (1) marshal arguments between components using the COMPOSITE IPC facilities, and (2) do namespace virtualization. Namespace virtualization does translation between two different namespaces, one in the FREERTOS application, and the other in the FREERTOS kernel. When an application is compiled into the FREERTOS kernel, it shares the namespace with the rest of the system, and most kernel objects (including threads, semaphores) are accessed directly by pointer. However, in MC-FREERTOS, pointers passed from the FREERTOS application *cannot be trusted* to contain a correct pointer. Thus, a set of translation tables exist in the FREERTOS-lib to map from the pointer—expected by the FREERTOS application as part of the FREERTOS API—to an integer descriptor that is passed via component invocation to the FREERTOS kernel. The FREERTOS-klib receives these descriptors, and translates them to the pointers to the corresponding objects within the FREERTOS kernel after validating that the objects are of the correct type for the function being invoked (*i.e.* that the object is used in a well-typed manner). Some details on the main APIs in FREERTOS, and how they are virtualized, follow:

- *Thread management.* The thread creation function must take a callback function to be executed in the new thread. This function pointer is saved in the FREERTOS-lib, and the FREERTOS-klib passes a function that upcalls at a known location into the FREERTOS application, where the thread retrieves the callback, and executes it.
- *Queues.* In addition to the namespace virtualization above, queues must pass data between the FREERTOS application and the FREERTOS kernel. We set up two uni-directional ring buffers of shared memory between the two libraries to pass the data from the FREERTOS application to the kernel, and vice-versa. Queues can behave synchronously, or asynchronously, as determined by the FREERTOS kernel. As the FREERTOS kernel component schedules its threads, including those in the FREERTOS application, with its own consistent notion of priority, we make no changes to the timing properties of the system aside from the overhead for the libraries and component invocation.
- *Semaphores.* These functions are simple and only conduct the virtualization already discussed.
- *Timed blocking.* The function that enables a thread to block for a span of time only requires marshalling the timeout argument to the FREERTOS kernel.

Summary. We present the design of MC-FREERTOS, which is a paravirtualized extension of FREERTOS. FREERTOS is incapable of mixed-criticality execution due to the inability to provide spatial isolation. This is a familiar story for many low-level RTOSes. We paravirtualize FREERTOS to provide a flexible MCS execution environment by porting FREERTOS to a component in COMPOSITE, implementing it in a hierarchical scheduler, and isolating in a separate component the low-criticality threads from those that are high-criticality, thus providing unchanged timing (minus constant overhead factors) and memory isolation.

C. MC-FREERTOS *Overhead and Performance*

Operation	Average	Stddev
FREERTOS Kernel Threads		
Semaphore w/ activation	0.368	0.014
Semaphore, no-contention	0.11	0.000
Enqueue	0.102	0.003
Dequeue	0.103	0.001
Queue round-trip	0.774	0.025
FREERTOS Application Threads		
Semaphore w/ activation	0.708	0.011
Semaphore, no-contention	0.669	0.002
Enqueue	0.418	0.008
Dequeue	0.567	0.009
Queue round-trip	1.699	0.066

TABLE I: Performance of the main MC-FREERTOS functions, both for FREERTOS kernel threads, and for FREERTOS application (low-criticality) threads. All measurements are in μ-seconds.

We measure the overheads of the FREERTOS kernel-resident threads (*i.e.* high criticality threads executing within the FREERTOS kernel component), and the overheads of FREERTOS application-resident threads that must make invocations to the FREERTOS kernel component for service. We execute a number of operations on a Intel(R) Core(TM) i7-2760QM CPU clocked at 2.4GHz. Table I displays these results. They include semaphore operations that activate a waiting thread, thus include the cost of a context switch; the overhead of a pair of uncontended semaphore operations (take + release); the separate cost of asynchronous enqueue and dequeue operations; and the cost of a "ping pong" through

queues which is synchronous round-trip communication between threads. There is no native x86 port of FREERTOS, so we could not compare against that.

Discussion: Adding memory isolation and the incumbent communication overheads between the FREERTOS application and FREERTOS kernel components does have an impact on performance. However, the performance of all relevant FREERTOS functions remains within reasonable bounds. Most overhead is directly attributable to component invocation costs of around 0.25 μ-seconds.

V. RELATED WORK

The most closely related work to this paper is HIRES by Parmer and West [3]. HIRES uses a similar resource hierarchy approach as this paper, which permits delegating memory and I/O in addition to CPU (scheduling), but HIRES does not provide the same strong temporal isolation guarantees in the presence of resource sharing as our work. Thus, HIRES is not directly usable to support an MCS, since the isolation of different criticality levels must be guaranteed.

The prevailing approach to resource sharing for hierarchical schedulers allocates budgets to each child, and then avoids budget exhaustion during critical section execution. SIRAP by Behnam et al., [8] checks for sufficient budget before entering a critical section. HSRP by Davis and Burns [9] permits bounded budget overruns so that critical sections may terminate despite budget exhaustion. Although HSRP can bound the overrun, preempting a critical section in case an overrun occurs is still problematic [10]. A quantitative evaluation of resource sharing approaches is given by Åsberg et al. [11]. Inam et al. modified FreeRTOS [12] to support two-level hierarchical scheduling for an MCS using HSRP to share resources. The authors evaluated the overhead of mode changes but do not examine resource sharing.

As mentioned in section I, hierarchical scheduling with virtualization can support MCSs. Unfortunately, virtualization suffers performance degradation due in part to a fundamental mismatch between mechanisms: virtualization technology was not designed with embedded systems/real-time scheduling in mind [13]. (Bruns et al. [14] argue contrarily that virtualization is effective for MCS on deeply-embedded devices that lack cache and memory management unit hardware. We do not consider such devices.) Prior work in MCSs with hierarchical scheduling attempts to remove the performance degradation while still isolating children [7], [15], [16]. In general, an MCS exacerbates the difficulties of resource sharing with hierarchical scheduling because of the need for strict isolation between different criticalities. The prevailing solution in the literature is to disallow resource sharing. Our work exhibits the strong isolation of virtualization-like approaches for MCSs with low performance loss despite allowing resource sharing.

VI. CONCLUSIONS

The flexibility and configurability of COMPOSITE makes it an ideal platform for MCSs not only because of its temporal guarantees, but also because of the capability to tailor system memory isolation to application criticalities. This paper has examined the use of the component-based model and the associated memory isolation within MC systems to enable configurable spatial isolation between different criticalities. We also approach the problem of extending HIRES into MC-HIRES as a generalization to enable a more descriptive and configurable interface between parent and child schedulers. Finally, we have introduced MC-FREERTOS, which utilizes the memory isolation in COMPOSITE and the hierarchical scheduling from MC-HIRES to enable both temporal and spatial isolation between different criticalities in a legacy RTOS that lacks memory isolation.

REFERENCES

[1] K. Lakshmanan, D. d. Niz, and R. R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11, 2011, pp. 47–56.

[2] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," in *1st workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, 2013, pp. 7–11.

[3] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[4] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11, 2011, pp. 39–48.

[5] Q. Wang, J. Song, and G. Parmer, "Stack management for hard real-time computation in a component-based OS," in *RTSS*, 2011.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1991, pp. 95–109.

[7] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12, 2012, pp. 93–102.

[8] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*. New York, NY, USA: ACM, 2007, pp. 279–288.

[9] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Washington, DC, USA, 2006, pp. 257–270.

[10] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 1, pp. 93–104, Feb 2010.

[11] M. Åsberg, M. Behnam, and T. Nolte, "An experimental evaluation of synchronization protocol mechanisms in the domain of hierarchical fixed-priority scheduling," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13, 2013, pp. 77–85.

[12] R. Inam, M. Sjodin, and R. Bril, "Mode-change mechanisms support for hierarchical freertos implementation," in *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, Sept 2013, pp. 1–10.

[13] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08, 2008, pp. 11–16.

[14] F. Bruns, D. Kuschnerus, and A. Bilgic, "Virtualization for safety-critical, deeply-embedded devices," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13, 2013, pp. 1485–1492.

[15] M. Völp, A. Lackorzynski, and H. Härtig, "On the expressiveness of fixed-priority scheduling contexts for mixed-criticality scheduling," in *1st International Workshop on Mixed Criticality Systems (WMC)*, 2013, pp. 13–18.

[16] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14, 2014, pp. 201–212.

Achieving temporal isolation in multiprocessor mixed-criticality systems

Sanjoy Baruah
The University of North Carolina
baruah@cs.unc.edu

Alan Burns
The University of York
alan.burns@york.ac.uk

***Abstract*—Upon mixed-criticality environments, the execution of high-criticality functionalities must be protected from interference from the execution of less critical functionalities. A means of achieving this objective upon multiprocessor environments is by forbidding less critical functionalities from executing anywhere upon the platform while more critical functionalities are executing upon any processor. This approach towards ensuring temporal isolation between the different criticalities that are co-implemented upon a common platform is explored in this paper, under both the global and partitioned paradigms of multiprocessor scheduling.**

I. Introduction

Thus far, mixed-criticality scheduling (MCS) theory has primarily concerned itself with the sharing of CPU computing capacity in order to satisfy the computational demand, as characterized by the worst-case execution times (WCET), of pieces of code. However, there are typically many additional resources that are also accessed in a shared manner upon a computing platform, and it is imperative that these resources also be considered in order that the results of MCS research be applicable to the development of actual systems. An interesting approach towards such consideration was advocated by Giannopoulou et al. [4] in the context of multiprocessor platforms: during any given instant in time, all the processors are only allowed to execute code of the same criticality level. This approach has the advantage of ensuring that access to all shared resources (memory buses, cache, etc.) during any time-instant are only from code of the same criticality level; since code of lower criticality are not allowed to execute simultaneously with code of higher criticality, the possibility of less critical code interfering with the execution of more critical code in accessing shared resources is ruled out.

In this paper, we seek to better understand the multiprocessor scheduling of mixed-criticality systems under this constraint of only executing jobs of a particular criticality at any given instant in time. To our knowledge, not much is known such scheduling problems; accordingly, we start out with a very simple model in which there are just two criticality levels (designated, as is standard, as HI and LO), and the workload is represented as a specified collection of independent jobs that all have a common release date and a common deadline. We consider the scheduling of such mixed-criticality systems both when jobs are permitted to migrate between processors (global scheduling), and when inter-processor migration is forbidden (partitioned scheduling). Global scheduling necessarily assumes a preemptive model of execution; for partitioned scheduling, preemption is not needed when all jobs have the same release date and deadline.

Organization. The remainder of this paper is organized as follows. In Section II, we elaborate upon the workload model that will be assumed in the reminder of this paper. We discuss the global scheduling of mixed-criticality task systems represented using this model in Section III, and partitioned scheduling in Section IV.

II. Model

A mixed-criticality *job* $j_i = (\chi_i, a_i, C_i(\text{LO}), C_i(\text{HI}), d_i)$ is characterized by a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$, a release time a_i, a deadline d_i, and two WCET estimates $c_i(\text{LO})$ and $c_i(\text{HI})$. We assume that $a_i, d_i, C_i(\text{LO})$, and $C_i(\text{HI})$ are all $\in \mathcal{R}^+$, the non-negative real numbers.

We seek to schedule a mixed-criticality instance I, consisting of a given collection of mixed-criticality jobs that all have the same release time (without loss of generality, assumed to be equal to zero) and the same deadline (denoted D) upon a platform of m unit-speed processors. Let I_{LO} denote the LO-criticality jobs in I, and I_{HI} the HI-criticality jobs in I. As is standard in mixed-criticality scheduling, we assume that the exact amount of execution required by a job is not known beforehand. If each job j_i completes upon executing for no more than $C_i(\text{LO})$ units, we say that the system exhibits LO-criticality behavior; if some j_i executes more than $C_i(\text{LO})$, but no more than $C_i(\text{HI})$, units, we say that the system exhibits HI-criticality behavior. The correctness requirement is that all jobs should complete by their deadlines in all LO-criticality behaviors, while only the HI-criticality jobs need to complete by their deadlines in any HI-criticality behavior.

Approach. The over-all approach that we advocate here is to first schedule the HI-criticality jobs during run-time — this can be thought of as a generalization of the *criticality monotonic (CM)* priority-assignment approach, which was previously shown [1] to be optimal for scheduling instances in which all jobs have equal deadlines (such as the instances considered here) upon uniprocessor platforms. If each HI-criticality job signals completion upon having executed for no more than its LO-criticality WCET, we recognize that we are in a LO-criticality behavior and begin execution of the LO-criticality jobs.

Notice that under the advocated approach, LO-criticality jobs only begin to execute after no HI-criticality jobs remain

	χ_i	a_i	$C_i(\text{LO})$	$C_i(\text{HI})$	d_i
j_1	LO	0	6	6	10
j_2	LO	0	6	6	10
j_3	LO	0	6	6	10
j_4	HI	0	2	10	10
j_5	HI	0	2	10	10
j_6	HI	0	4	4	10
j_7	HI	0	4	4	10

TABLE I
AN EXAMPLE MIXED-CRITICALITY TASK INSTANCE.

that need to be executed. The problem of scheduling these LO-criticality jobs therefore becomes a "regular" (i.e., not mixed-criticality) scheduling problem. Hence we can first determine, using techniques from conventional scheduling theory, the minimum duration (the *makespan*) of a schedule for the LO-criticality jobs. Once this makespan Δ is determined, the difference between D and this makespan (i.e., $(D-\Delta)$) is the duration for which the HI-criticality jobs are allowed to execute to completion in any LO-criticality schedule. Determining schedulabilty for the mixed-criticality instance is thus reduced to determining whether I_{HI} can be scheduled in such a manner that

- If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{LO})$, then the schedule makespan is $\leq (D-\Delta)$; and
- If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{HI})$, then the schedule makespan is $\leq D$.

Note that we do not in general know, prior to actually executing the jobs, whether each job will complete within its LO-criticality WCET or not. Hence it is not sufficient to construct two entirely different schedules that separately satisfy these two requirements above; instead, the two schedules must be identical until at least that point in time at which some job executes for more than its LO-criticality WCET. (This observation is further illustrated in the context of global scheduling in Example 1 below.)

III. GLOBAL SCHEDULING

We start out considering the global scheduling of instances of the kind described in Section II above. Given a collection of n jobs with execution requirements $c_1, c_2, \ldots, c_n$, McNaughton [8, page 6] showed as far back as 1959 that the minimum makespan of a preemptive schedule for these jobs on m unit-speed processors is

$$\max\left(\frac{\sum_{i=1}^{n} c_i}{m}, \max_{i=1}^{n}\{c_i\}\right) \tag{1}$$

A direct application of McNaughton's result yields the conclusion that the minimum makespan Δ for a global preemptive schedule for the jobs in I_{LO} is given by

$$\Delta \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{LO}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{LO}}\left\{C_i(\text{LO})\right\}\right) \tag{2}$$

Hence in any LO-criticality behavior it is necessary that the HI-criticality jobs in I — i.e., the jobs in I_{HI} — must be scheduled to have a makespan no greater than $(D-\Delta)$:

$$\max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{HI}}\left\{C_i(\text{LO})\right\}\right) \leq D-\Delta\ . \tag{3}$$

(Here, the makespan bound on the LHS follows again from a direct application of McNaughton's result.) Additionally, in order to ensure correctness in any HI-criticality behavior it is necessary that the makespan of I_{HI} when each job executes for its HI-criticality WCET not exceed D:

$$\max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{HI})}{m}, \max_{\chi_i=\text{HI}}\left\{C_i(\text{HI})\right\}\right) \leq D \tag{4}$$

(where the LHS is again obtained using McNoughton's result.)

One might be tempted to conclude that the conjunction of Conditions 3 and 4 yields a sufficient schedulability test. However, this conclusion is erroneous, as is illustrated in Example 1 below.

Example 1: Consider the scheduling of the mixed-criticality instance of Table I upon 3 unit-speed processors. For this instance, it may be validated that

- By Equation 2, Δ evaluates to $\max(\frac{6+6+6}{3}, 6)$ or 6.
- The LHS of Condition 3 evaluates to $\max(\frac{2+2+4+4}{3}, 4)$, or 4. Condition 3 therefore evaluates to true.
- The LHS of Condition 4 evaluates to $\max(\frac{10+10+4+4}{3}, 10)$, or 10. Condition 4 therefore also evaluates to true.

However for this example, the schedule that causes Condition 3 to evaluate to true *must* have jobs j_6 and j_7 execute throughout the interval $[0, 4)$, while the one that causes Condition 4 to evaluate to true must have j_4 and j_5 execute throughout the interval $[0, 10)$ — see Figure 1. Since we only have three processors available, during any given execution of the instance at least one of the four jobs j_4–j_7 could not have been executing throughout the interval $[0, 2)$.

- If one of $\{j_4, j_5\}$ did not execute throughout $[0, 2)$ and the behavior of the system turns out to be a HI-criticality one, then the job not executing throughout $[0, 2)$ will miss its deadline.
- If one of $\{j_6, j_7\}$ did not execute throughout $[0, 2)$ and the behavior of the system turns out to be a LO-criticality one, then the job $\in \{j_6, j_7\}$ not executing throughout $[0, 2)$ will not complete by time-instant 4, thereby delaying the start of the execution of the LO-criticality jobs to beyond time-instant 4. These jobs will then not be able to complete by their common deadline of 10.

■

The example above illustrates that the conjunction of Conditions 3 and 4, with the value of Δ defined according to Equation 2, is a necessary but *not* a sufficient global schedulability test. Below, we will derive a sufficient global schedulability test with run-time that is polynomial in the representation of the input instance; we will then illustrate, in Example 2, how this test does not claim schedulability of the instance from Example 1. This schedulability test is based upon a network flow argument, as follows. We will describe a

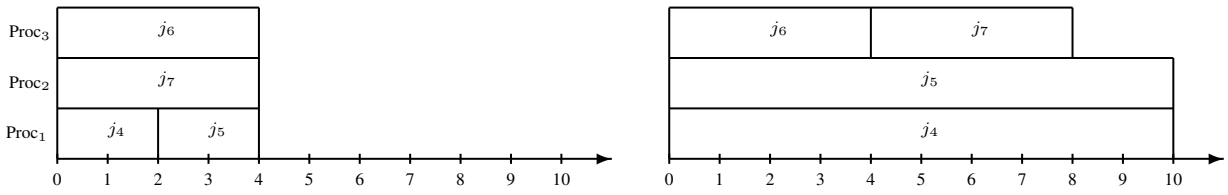


Fig. 1. Schedules for I_{HI} for the task system of Example 1. The left schedule is for a LO-criticality behavior, and has a makespan of four; it thus bears witness to the fact that this mixed-criticality instance satisfies Condition 3. The right schedule is for a HI-criticality behavior – it has a makespan of ten, thereby bearing witness that the instance satisfies Conditions4 as well. Observe that the schedules are *different* at the start: job j_5 does not execute over $[0, 2)$ in the left schedule but it does in the right schedule, while job j_7 does not execute over $[0, 4)$ in the right schedule but it does in the left schedule.

polynomial-time reduction from any dual-criticality instance I to a weighted digraph G with a designated source vertex and a designated sink vertex, such that flows of a certain size or greater from the source to the sink in G correspond exactly (in a manner that will be made precise) to a valid global schedule for the instance I. Thus, the problem of determining global schedulability is reduced to determining the size of a maximum flow in a graph, which is known to be solvable in polynomial time using, for instance, the Floyd-Fulkerson algorithm [3].

We now describe below the construction of a weighted digraph G from an input instance I. First, we compute the value of Δ for this input instance according to Equation 2. The graph we build is a "layered" one: the vertex set V of G is the union of 6 disjoint sets of vertices $V_0, \ldots, V_5$, and the edge set E of G is the union of 5 disjoint sets of edges $E_0, \ldots, E_4$, where E_i is a subset of $(V_i \times V_{i+1} \times \mathcal{R}^+)$, $0 \leq i \leq 4$. The digraph G is thus a 6-layered graph — see Figure 2 — in which all edges connect vertices in adjacent layers. The sets of vertices in G are as follows:

$$
\begin{aligned}
V_0 &= \{\mathsf{source}\},\\
V_1 &= \{\langle 1, j_i\rangle \mid j_i \in I_{\text{HI}})\},\\
V_2 &= \{\langle 2, j_i, \text{LO}\rangle, \langle 2, j_i, \text{HI}\rangle \mid j_i \in I_{\text{HI}}\},\\
V_3 &= \{\langle 3, j_i, \alpha\rangle, \langle 3, j_i, \beta\rangle \mid j_i \in I_{\text{HI}}\},\\
V_4 &= \{\langle 4, \alpha\rangle, \langle 4, \beta\rangle\}, \text{ and}\\
V_5 &= \{\mathsf{sink}\}.
\end{aligned}
$$

Intuitively speaking, each vertex in V_1 represents a HI-criticality job; for each such job, there are two vertices in V_2 representing respectively its LO-criticality execution and the excess execution (beyond its LO-criticaliy WCET) in case of HI-criticality behavior. The vertex $\langle 3, j_i, \alpha\rangle$ will correspond to the amount of execution job j_i actually receives over the interval $[0, D - \Delta)$ – i.e., during the interval within which it must complete execution within any LO-criticality behavior; the vertex $\langle 3, j_i, \beta\rangle$ will correspond to the amount of execution job j_i receives over the interval $[D - \Delta, D)$. The vertices $\langle 4, \alpha\rangle$ and $\langle 4, \beta\rangle$ represent the total amount of execution performed upon the platform during the intervals $[0, D - \Delta)$ and $[D - \Delta, D)$ respectively.

Next, we list the edges in G. An edge is represented by a 3-tuple: for $u, v \in V$ and $w \in \mathcal{R}^+$, the 3-tuple $(u, v, w) \in E$ represents an edge from u to v that has a capacity w. The sets of edges in G are as follows:

$$
\begin{aligned}
E_0 &= \{(\mathsf{source}, \langle 1, j_i\rangle, C_i(\text{HI})) \mid j_i \in I_{\text{HI}}\},\\
E_1 &= \{(\langle 1, j_i\rangle, \langle 2, j_i, \text{LO}\rangle, C_i(\text{LO})),\\
&\qquad (\langle 1, j_i\rangle, \langle 2, j_i, \text{HI}\rangle, C_i(\text{HI}) - C_i(\text{LO})) \mid j_i \in I_{\text{HI}}\},\\
E_2 &= \{(\langle 2, j_i, \text{LO}\rangle, \langle 3, j_i, \alpha\rangle, C_i(\text{LO})),\\
&\qquad (\langle 2, j_i, \text{HI}\rangle, \langle 3, j_i, \alpha\rangle, C_i(\text{HI}) - C_i(\text{LO})),\\
&\qquad (\langle 2, j_i, \text{HI}\rangle, \langle 3, j_i, \beta\rangle, C_i(\text{HI}) - C_i(\text{LO})), \mid j_i \in I_{\text{HI}}\},\\
E_3 &= \{(\langle 3, j_i, \alpha\rangle, \langle 4, \alpha\rangle, D - \Delta),\\
&\qquad (\langle 3, j_i, \beta\rangle, \langle 4, \beta\rangle, \Delta) \mid j_i \in I_{\text{HI}}, \text{ and}\\
E_4 &= \{(\langle 4, \alpha\rangle, \mathsf{sink}, m(D - \Delta)), (\langle 4, \Delta\rangle, \mathsf{sink}, m\Delta)\}.
\end{aligned}
$$

We now try and explain the intuition behind the construction of G. The maximum flow that we will seek to push from the source to the sink is equal to $\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$. Achieving this flow will require that $C_i(\text{HI})$ units of flow go through $\langle 1, j_i\rangle$, which in turn requires that $C_i(\text{LO})$ units of flow go through $\langle 2, j_i, \text{LO}\rangle$, and $(C_i(\text{HI}) - C_i(\text{LO}))$ units of flow go through $\langle 2, j_i, \text{HI}\rangle$, for each $j_i \in I_{\text{HI}}$. This will require that all $C_i(\text{LO})$ units of flow from $\langle 2, j_i, \text{LO}\rangle$ go through $\langle 3, j_i, \alpha\rangle$; the flows from $\langle 2, j_i, \text{HI}\rangle$ through the vertices $\langle 3, j_i, \alpha\rangle$ and $\langle 3, j_i, \beta\rangle$ must sum to $(C_i(\text{HI}) - C_i(\text{LO}))$ units. The global schedule for I_{HI} is determined as follows: ***the amount of execution received by*** j_i ***during*** $[0, D - \Delta)$ ***is equal to the amount of flow through*** $\langle 3, j_i, \alpha\rangle$***; the amount of execution received by*** j_i ***during*** $[D - \Delta, D)$ ***is equal to the amount of flow through*** $\langle 3, j_i, \beta\rangle$**.** Since the outgoing edge from $\langle 3, j_i, \alpha\rangle$ has capacity $(D - \Delta)$, it is assured that j_i is not assigned more execution than can be accommodated upon a single processor; since the outgoing edge from $\langle 4, \alpha\rangle$ is of capacity $m(D - \Delta)$, it is assured that the total execution allocated during $[0, D - \Delta)$ does not exceed the capacity of the m-processor platform to accommodate it. Similarly for the interval $[d-\Delta, D)$: since the outgoing edge from $\langle 3, j_i, \beta\rangle$ has capacity Δ, it is assured that j_i is not assigned more execution than can be accommodated

upon a single processor; since the outgoing edge from $\langle 4, \beta \rangle$ is of capacity $m\Delta$, it is assured that the total execution allocated during $[D - \Delta, D)$ does not exceed the capacity of the m-processor platform to accommodate it. Now for both the intervals $[0, D - \Delta)$ and $[D - \Delta, D)$, since no individual job is assigned more execution than the interval duration and the total execution assigned is no more than m times the interval duration, McNaughton's result (Condition 1) can be used to conclude that these executions can be accommodated within the respective intervals.

This above informal argument can be formalized to establish the following lemma; we omit the details.

Theorem 1: If there is a flow of size

$$\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$$

in G then there exists a global schedule for the instance I. ■

Example 2: Let us revisit the task system described in Example 1 — for this example instance, we had seen by instantiation of Equation 2 that $\Delta = 6$. The digraph constructed for this task system would require each of j_4–j_7 to transmit at least their corresponding $C_i(\text{LO})$'s, i.e., $2, 2, 4$, and 4, respectively, units of flow through the vertex $\langle 4, \alpha \rangle$, which is do-able since the platform capacity over this interval is $3 \times 4 = 12$. But such a flow completely consumes the platform capacity during $[0, 4)$, which requires that *all* of j_4 and j_5's $(C_i(\text{HI}) - C_i(\text{LO}))$ flows, of $(10 - 2) = 8$ units each, flow through the edges $(\langle 3, j_4, \beta \rangle, \langle 4, \beta \rangle, 6)$ and $(\langle 3, j_5, \beta \rangle, \langle 4, \beta \rangle, 6)$. But such a flow would exceed the capacity of the edge (which is six units), and is therefore not feasible. The digraph constructed for the example instance of Example 1 thus does not permit a flow of size $\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$, and Theorem 1, does not declare the instance to be globally schedulable.

■

As previously stated, determining the maximum flow through a graph is a well-studied problem. The Floyd-Fulkerson algorithm [3], first published in 1956, provides an efficient polynomial-time algorithm for solving it. In fact, the Floyd-Fulkerson algorithm is constructive in the sense that it actually constructs the flow – it tells us how much flow goes through each edge in the graph. We can therefore use a flow of the required size, if it exists, to determine how much of each job must be scheduled prior to $(D - \Delta)$ in the LO-criticality schedule, and use this pre-computed schedule as a look-up table to drive the run-time scheduler.

IV. Partitioned scheduling

We now turn to partitioned scheduling, which was the context within which Giannopoulou et al. [4] had initially proposed the paradigm of only executing jobs of one criticality at any instant in time. In partitioned scheduling, we will partition the entire collection of jobs – both the HI-criticality and the LO-criticality ones – amongst the processors prior to run-time. We will also determine some time-instant S, $0 \leq S \leq D$, such that only HI-criticality jobs are executed upon all the processors during $[0, S)$, and only LO-criticality jobs are executed during $[S, D)$. A run-time protocol needs to be defined and supported that will manage the change from HI-criticality to LO-criticality execution. As HI-criticality and LO-criticality jobs cannot execute concurrently, any processor that is still executing a HI-criticality job at some time t must prevent *all* other processors from switching to LO-criticality jobs. Such a protocol would need to be ether affirmative ("its OK to change") or negative ("do not change"):

- *affirmative*: each processor broadcasts a message (or writes to shared memory) to say it has completed all its HI-criticality jobs; when each processor has completed its own HI-criticality work and has received $(m - 1)$ such messages it switches to LO-criticality work.
- *negative*: if any processor is still executing its HI-criticality work at time S it broadcasts a message to inform all other processors; any processor that is not in receipt of such a message at time S+δ will move to its LO-criticality work (where δ is a deadline for receipt of the 'no-change' message, determined based upon system parameters such as maximum propagation delay).

In terms of message-count efficiency, the negative message is more effective since normal behavior would result in no messages being sent; whereas the affirmative protocol would generate m broadcast messages. The affirmative protocol is, however, more resilient and less dependent on the temporal behavior of the communication media.

If shared memory is used then a set of flags could indicate the status of each processor. However, spinning on the value of such flags could cause bus contention issues for those processors attempting to complete their HI-criticality work.

We now turn to the problem of partitioning the jobs amongst the processors prior to run-time. Let us first consider the scheduling of just the LO-criticality jobs — i.e., the jobs in I_{LO}. Determining a schedule of minimum makespan for these jobs is equivalent to the bin-packing [6] problem, and is hence highly intractable: NP-hard in the strong sense. Hochbaum and Shmoys [5] have designed a *polynomial-time approximation scheme* (PTAS) for the partitioned scheduling of a collection of jobs to minimize the makespan that behaves as follows. Given any positive constant ϕ, if an optimal algorithm can partition a given task system τ upon m processors each of speed s, then the algorithm in [5] will, in time polynomial in the representation of τ, partition τ upon m processors each of speed $(1 + \phi)s$. This can be thought of as a *resource augmentation* result [7]: the algorithm of [5] can partition, in polynomial time, any task system that can be partitioned upon a given platform by an optimal algorithm, provided it (the algorithm of [5]) is given augmented resources (in terms of faster processors) as compared to the resources available to the optimal algorithm.

We can use the PTAS of [5] to determine in polynomial time, to any desired degree of accuracy, the minimum makespan of any partitioned schedule of the LO-criticality jobs in the instance I. Let Δ_P denote this makespan. Hence

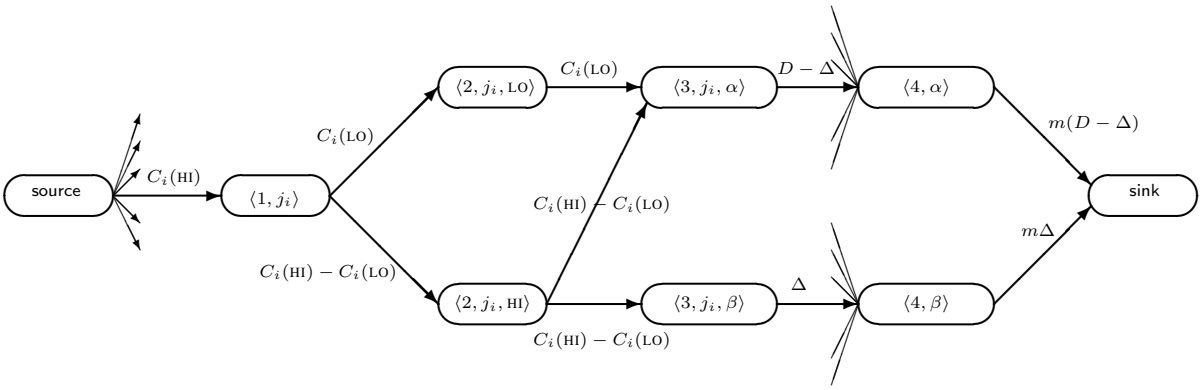


Fig. 2. Digraph construction illustrated. All vertices and edges pertinent to the job j_i are depicted. Additional edges emanate from vertex sink to a vertex $\langle 1, j_\ell \rangle$, for each $j_\ell \in I_{\text{HI}}$; additional edges enter the vertices $\langle 4, \alpha \rangle$ and and $\langle 4, \beta \rangle$ from vertices $\langle 3, j_\ell, \alpha \rangle$ and $\langle 3, j_\ell, \beta \rangle$ respectively, for each $j_\ell \in I_{\text{HI}}$.

to ensure a correct schedule we need to complete scheduling all the HI-criticality jobs in I within the time interval $[0, D - \Delta_P)$; i.e., with a makespan $(D - \Delta_P)$. (The time-instant S mentioned above in the context of the run-time management of the system is therefore equal to $D - \Delta_P$.) Now, determining whether I can be successfully scheduled using partitioned scheduling reduces to determining whether there is a partitioning of just the HI-criticality jobs — i.e., the jobs in I_{HI} — satisfying the properties that

P1. If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{LO})$, then the schedule makespan is $\leq (D - \Delta_P)$; and

P2. If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{HI})$, then the schedule makespan is $\leq D$.

(Note that both these properties must be satisfied by a *single* partitioning of the jobs in I_{HI} – it is not sufficient to identify one partitioning that satisfies P1 and another that satisfies P2.)

This partitioning problem turns out to be closely related to the *vector scheduling problem*. Vector scheduling is the natural multi-dimensional generalization of the partitioning problem to minimize makespan, to situations where jobs may use multiple different kinds of resources and the load of a job cannot be described by a single aggregate measure. For example, if jobs have both CPU and memory requirements, their processing requirements are appropriately modeled as two dimensional vectors, where the value along each dimension corresponds to one of the requirements. Clearly, an assignment of vectors to the processors is valid if and only if no processor is overloaded along any dimension (i.e., for any resource). Chekuri and Khanna [2] give a PTAS for solving the vector scheduling problem when the number of dimensions is a constant.

It is not difficult to map our problem of partitioning the HI-criticality jobs (discussed above) to the vector scheduling problem. Each HI-criticality job $j_i \in I_{\text{HI}}$ is modeled as a two-dimensional load vector $\langle C_i(\text{LO}), C_i(\text{HI}) \rangle$, and the capacity constraint for each processor is represented by the vector $\langle (D - \Delta_P), D \rangle$. We can therefore use the PTAS of [2] to determine whether I_{HI} can be partitioned in a manner that satisfies the properties P1 and P2 above, to any desired degree of accuracy in time polynomial in the representation of the instance.

V. Context and Conclusions

Mixed-criticality scheduling (MCS) theory must extend consideration beyond simply CPU computational demand, as characterized by the worst-case execution times (WCET), if it is to be applicable to the development of actual mixed-criticality systems. One interesting approach towards achieving this goal was advocated by Giannopoulou et al. [4] — enforce temporal isolation amongst different criticality levels by only permitting functionalities of a single criticality level to execute at any instant in time. Such inter-criticality temporal isolation ensures that access to all shared resources are only from equally critical functionalities, and thereby rules out the possibility of less critical functionalities compromising the execution of more critical functionalities while accessing shared resources.

We have considered here the design of scheduling algorithms that implement this approach. For a very simple workload model — a dual-criticality system that is represented as a collection of independent jobs that share a common release time and deadline — we have designed asymptotically optimal algorithms for both global and partitioned scheduling:

- For global scheduling, we have designed a polynomial-time sufficient schedulability test that determines whether a given mixed-criticality system is schedulable, and an algorithm that actually constructs a schedule if it is.
- For partitioned scheduling, we have shown that the problem is NP-hard in the strong sense, thereby ruling out the possibility of obtaining optimal polynomial-time algorithms (unless $\text{P} = \text{NP}$). We have however obtained what is, from a theoretical perspective, the next best thing – a polynomial-time approximation scheme (PTAS) that determines, in polynomial time, a partitioning of the task system that is as close to being an optimal partitioning algorithm as desired.

The work reported here should be considered to be merely a starting point for research into the particular approach towards mixed-criticality scheduling advocated in [4]. While the PTAS

for partitioned scheduling is likely to be the best we can hope for (in asymptotic terms), we do not have a estimate as to how far removed from optimality our global schedulability test is. We also plan to extend the workload model to enable consideration of jobs with different release dates and deadlines, and later to the consideration of recurrent task systems. An orthogonal line of follow-up research concerns the implementation of the global and partitioned approaches presented here – experimentation is needed to determine how they behave upon actual systems.

ACKNOWLEDGEMENTS

This research is partially supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1409175; ARO grant W911NF-09-1-0535; and ESPRC grant MCC (EP/K011626/1).

REFERENCES

[1] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

[2] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999, pp. 185–194.

[3] L. Ford and D. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, 1956.

[4] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *International Conference on Embedded Software (EMSOFT)*, Montreal, Oct 2013, pp. 17:1–17:15.

[5] D. Hochbaum and D. Shmoys, "Using dual approximation algorithms for scheduling problems: Theoretical and practical results," *Journal of the ACM*, vol. 34, no. 1, pp. 144–162, Jan. 1987.

[6] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[7] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM*, vol. 37, no. 4, pp. 617–643, 2000.

[8] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, pp. 1–12, 1959.

A Memory Arbitration Scheme for Mixed-Criticality Multicore Platforms

Bekim Cilku*, Alfons Crespo†, Peter Puschner*, Javier Coronel‡ and Salvador Peiro†
*Vienna University of Technology, Vienna, Austria
{bekim,peter}@vmars.tuwien.ac.at
†Universidad Politecnica de Valencia, Valencia, Spain
{acrespo,speiro}@ai2.upv.es
‡fentISS, Valencia, Spain
jcoronel@fentiss.com

***Abstract*—In mixed-criticality systems, applications of different criticality levels share the same computing platform. To avoid spatial and temporal interference of the applications, the computing platform must implement measures for spatial and temporal isolation. In this paper we show how the enhancement of a static memory arbiter by a second, dynamic arbitration layer facilitates the interference-free integration of mixed-criticality applications with different performance requirements. This paper (a) compares the performance tradeoffs of the new dual-layer arbiter and a COTS arbiter and (b) evaluates the performance of an XtratuM hypervisor system running on a platform with this dual-layer arbiter.**

I. Introduction

The high processing capability that multi-core embedded systems have reached allows us to run multiple applications on a single shared hardware platform [1]. However, some of the integrated applications may have firm real-time constraints that require a formal proof that the deadlines are met, while the others may be less demanding. For such a mixed-criticality system, only the set of critical applications needs to be certified; the rest of the applications do not need certification or may be certified to a lower level [2]. Obtaining a certification only for critical applications can become a difficult task due to the hardware sharing dependency and the diversity of functionalities that the system performs concurrently. The key approach towards a complexity reduction is to prevent the interference between integrated applications both in the temporal and spatial domain [3]. Temporal isolation preserves the timing behavior of applications such that they do not affect one another while executing concurrently on the shared platform [4]. Spatial isolation protects memory elements of applications so they cannot be accessed by the other applications.

For a multi-core platform with shared main memory, spatial isolation between applications can be achieved simply by integrating a Memory Management Unit (MMU) [5] as part of the memory-address translation process. The MMU table is set up to assign a different memory space to each application. In this way, the MMU table protects the given memory space of applications against possible violations from the other ones. Establishing temporal isolation between time-critical applications is a more complex problem. This comes as a consequence of the resource sharing between applications on the same core, as well as from the sharing of resources (main memory and I/O components) between applications running on different cores.

In this paper we present the MultiPARTES[1] platform that provides full temporal isolation for time-critical applications at all levels of a computing system. This platform consists of multi-core hardware with a shared bus on top of which the XtratuM [6] hypervisor executes (Figure 1).

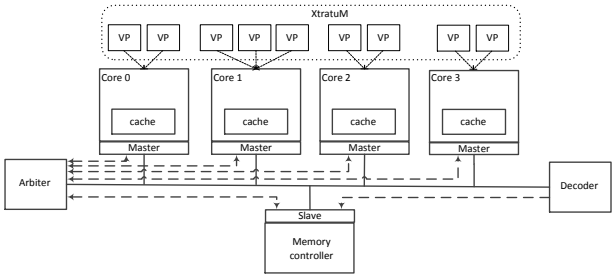


Fig. 1: MultiPARTES Platform

The hypervisor eliminates the possibility of interference between virtual partitions running on the same CPU by implementing a static cyclic scheduling where each virtual partition is associated with a fixed number of CPU cycles. At runtime, each partition is activated based on the static schedule and gets a predefined amount of processor time. Applications of different criticality level are mapped on different virtual partitions. However, the hypervisor is not able to ensure temporal isolation of the critical partitions when they access a shared bus. To eliminate the inter-core interference we have designed a new bus-arbiter scheme that preserves the isolation properties for critical virtual partitions and provides better utilization of shared resources for non-critical ones. The arbiter is based on a hierarchical, two-layer arbitration scheme that switches between a critical and non-critical mode.

The paper is organized as follows. The next section describes the hardware architecture of the MultiPARTES platform, the new bus-arbiter scheme and presents preliminary results on arbiter performance. Section 3 describes the XtratuM hypervisor and its integration to the multi-core platform. Temporal interference and preformance evaluation of the whole platform are shown in Section 4. Section 5 concludes the paper.

II. Hardware Platform for Mixed-Criticality Multicore Systems

The hardware used in MultiPARTES consists of a multi-core LEON3 processor that is interconnected with a memory controller and

[1] www.multipartes.eu

I/O resources through a shared AMBA bus. LEON3 is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. Multi-core LEON3 systems are highly customizable with respect to the processor core and periphery features. LEON3 has a seven-stage pipeline with a Harvard architecture, separate caches for instructions and data, a hardware multiplier and divider, on-chip debug support, an MMU with a configurable TLB, and multi-processor extensions [7].

The interconnection between cores with main memory and I/O components is realized through the Advanced Microcontroller Bus Architecture (AMBA). AMBA is an open standard specification that defines an on-chip communication standard for designing high-performance embedded systems [8]. It is widely used in network interconnect chips, RAM controllers, Flash memory controllers and SoCs (System on Chips) [9]. It consists of a high-performance system backbone bus (AHB) on which the CPUs, on-chip memories and other DMA devices reside, and a low bandwidth bus (APB) to which most of the system peripheral devices are connected. The APB is optimized for low power consumption and reduced interface complexity.

The AHB and the APB are connected via a bus bridge. Components which can initiate read or write operations are called AHB master while those which respond to a read/write operation are called AHB slave. Since the bus is shared between all master components, only one master is allowed to initiate a transfer at any time. AMBA uses a centralized arbiter to determine which master gets the right to commence a transfer and also ensures that at any given time at most a single transfer is in progress. The decision for granting the bus is based on an arbitration algorithm. The basic requirements for an arbitration policy are that it should guarantee the fairness of accesses between master components and it has to prevent the starvation of the masters.

A. Dual-Layer Arbitration Scheme

The MultiPARTES platform uses a two-layer arbitration scheme for accesses to the shared-memory bus. The first layer is based on *time division multiple access* (TDMA) and is responsible for guaranteeing the temporal properties of critical partitions, while the other layer employs a *round-robin* (RR) arbitration policy that controls the bus access for non-critical partitions.

TDMA is an arbitration policy that guarantees a fixed bus bandwidth by a priory assigning time slots of fixed length to each master. In contrast, the RR arbitration scheme grants bus access to every master on the bus in a circular manner. A master relinquishes control over the bus when it no longer has data to transfer. When a master has completed its transfer, it passes control to the next master in line [10].

The TDMA scheme is the main bus access driver. Each core that runs time-critical applications has an a-priori assigned time slot to guarantee access on the bus. Individual cores that need higher bandwidth can be assigned multiple slots within the scheduling frame. A few sequential time slots from TDMA arbitration are reserved for cores with non-critical applications. On top of these slots the RR is built. During this RR time interval, dynamic arbitration is activated and all transactions of non-critical cores are competing for the access to the memory bus. To ensure that non-critical cores cannot interfere with the critical ones, an additional rule has to be enforced. The dynamic arbiter (RR) must not serve any request that is issued after the start of the last timeslot in the sequence of the dynamic time slots. Otherwise, such a request could overlap with the next static time slot for the critical cores, thus invalidating the time predictability of the TDMA scheme.

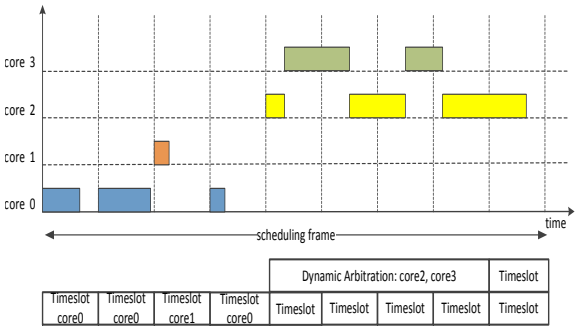


Fig. 2: Dual-policy arbitration scheme

An example of dual-layer arbitration for four-core hardware is presented in Figure 2. The first two cores (core 0 and core 1) are running time-critical applications while the other two cores (core 2 and 3) are executing non-critical ones. The scheduling frame has nine slots where four are assigned to critical cores (core 0 is assigned three slots under the assumption that it needs more bandwidth), the next four slots are reserved for non-critical cores and the last slot is reserved for the completion of ongoing non-critical transactions. Core 0 and 1 are granted to start bus transactions only at the beginning of their assigned slots in order to guarantee the non-interference of transactions. In contrast, cores 2 and 3 are accessing the bus when dynamic arbitration (RR) is active. The last slot, does not allow for any bus access but only serves the completion of the ongoing transaction of core 2.

B. Hardware Architecture of the Dual-Layer Arbiter

The arbiter architecture consists of a TDMA and an RR component (Figure 3). The TDMA components has a wrap incremental counter, a shift register and a controller. The wrap counter is incremented at each clock cycle with a maximal value equal to the number of cycles for one slot. The function of this unit is to shift the token in a shift register at the end of the time slot. The shift register keeps record of the slots. Depending on the active slot, the controller either grants access to an critical request (HBUSREQ0 or HBUSREQ1) or activates/deactivates the RR arbiter.

All request signals from non-critical partitions are connected to the RR-arbiter queue. When the RR arbiter is activated, it grants partitions bus access in the same order as the requests have arrived (HBUSREQ2 or HBUSREQ3).

C. Evaluation of the Arbiter

We implemented and deployed the dual-layer bus arbiter on an FPGA development kit (terasic DE2-115) to demonstrate the feasibility of the Dual-Layer arbiter and to evaluate its performance. The arbiter is written in VHDL language and deployed as part of the

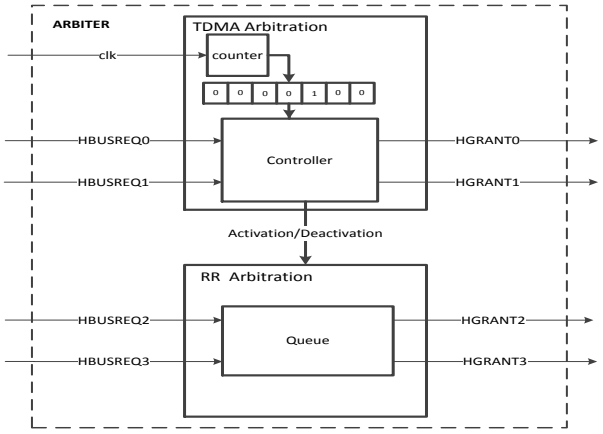


Fig. 3: Arbiter architecture for mixed-criticality systems

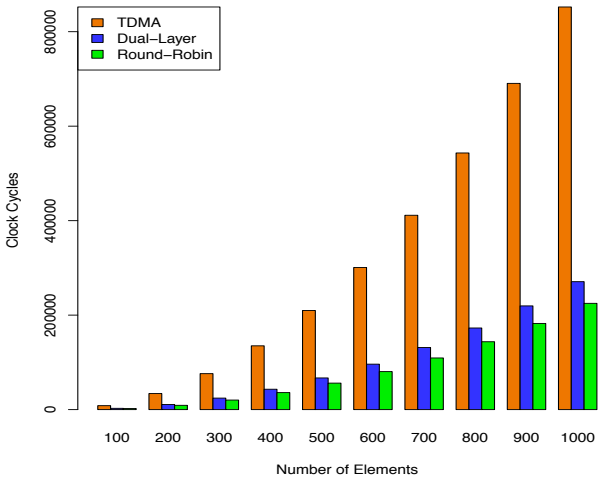


Fig. 4: Execution time of bubble sort algorithm

Grlib IP library [11]. The interconnection with the memory controller and the I/O resources is done through the AMBA bus. The bus is configured not to support split transactions.

For this evaluation a bubble sort algorithm was chosen. The algorithm sorts vectors of sizes from 100 to 1000 elements. In this evaluation, the algorithm is executed on a bare-metal processor, without any operating system in order to avoid the overhead generated from the hypervisor. All four cores were executing the same code. The goal was to test the temporal isolation for time-critical cores and also to compare the execution performance of the noncritical partitions. For comparison we used three different bus arbitration policies: pure TDMA, RR and our Dual-Layer (DL) policy.

The TDMA scheduling frame consisted of four slots and each slot was assigned to a core. The DL arbiter used five slots where the critical CPUs (CPU0 and CPU1) were assigned to the first two slots (slot 0 and 1) while the non-critical CPUs were allowed on the following two slots. No arbitration was allowed on the fifth slot (see above). The slot size was 20 clock cycles (the longest transfer was taking 18 cycles).

Figure 4 shows the observed execution time of the bubble sort algorithm for different vector sizes. The execution time of non-critical partitions is illustrated for TDMA, RR and DL. From the results we see that the execution time of the algorithm in non-critical cores with DL arbitration is 3.1 time shorter than in a system with pure TDMA. Compared to pure RR, the execution of bubble sort with DL lasts 1.2 times longer. For critical partitions the execution time with TDMA and DL is the same.

In summary, the results of this experiment suggest that DL arbitration improves the performance of non-critical partitions without affecting the time properties of the critical ones.

III. The XtratuM Hypervisor

A. Introduction of XtratuM

XtratuM [12], [13] is a bare-metal hypervisor specifically designed for embedded real-time systems that uses para-virtualization techniques to emulate hardware behaviour. The para-virtualized model offers potential performance benefits when a guest operating system or application is aware that it is running within a virtualized environment, and it has been modified to exploit this. One potential downside of this approach is that such modified guests cannot ever be migrated back to run on physical hardware.

XtratuM has been designed to achieve real-time constraints with a set of properties that strongly follow certification issues. These properties can be summarised as:

- Spatial isolation: A partition is completely allocated to isolated memory regions. The hypervisor guarantees the spatial isolation of the partitions.
- Temporal isolation: A partition is executed at specified and fixed temporal intervals. A cyclic scheduling policy is implemented by the hypervisor. The temporal allocation of time to a partition is not impacted by the execution of other partitions, although shared resources could produce interference in the execution time duration of its activities.
- Predictability: A partition with real-time constraints has to execute its code in a predictable way. It can be influenced by the underlying layers of software (guest-OS and hypervisor) and by the hardware. From the hypervisor point of view, the predictability applies to the provided services, the operations involved in the partition execution and internal operations (partition context switch, interrupt management, etc.).
- Fault isolation and management: Fault management is a fundamental aspect in critical systems, and it is strongly related with certification issues. Faults, when they occur, are detected and handled via Health Monitor which is statically configured.
- Static resource allocation: The system architect is responsible for the system definition and resource allocation. This

system definition is detailed in the system's configuration file, which specifies all system resources, namely number of CPUs, memory layout, peripherals, partitions, the execution plan of each CPU, etc.

B. Integration on the Multicore Platform

XtratuM has been adapted to multi-core systems [14] based on LEON4 processors and x86 and LEON3-bicore in the MultiPARTES project [15], [16]. In the multi-core approach, the hypervisor can provide several virtual CPUs to the partitions. A partition can be mono or multi-core. Different partitions (from the point of view of the number of cores) can coexist in the system. This approach allows profiting from a multi-core platform, even if the partitions are not multi-core by building multi-core or monocore partitions.

In order to handle the underlying multi-core hardware it can be configured following an Asymmetric Multiprocessing (AMP) or Symmetric Multi-Processor (SMP). In AMP, there is an instance of the hypervisor running on each core, which executes the allocated partitions. In the SMP approach, one single hypervisor instance manages all hardware resources. While the AMP software architecture simplifies the hypervisor, the SMP approach permits the use of mono or multi-core execution environments, and offers higher flexibility to assign partitions to different cores. Moreover, mono-core partitions in SMP architectures can be permanently allocated to one core, or several on different partition activations with no temporal overlap.

In the adaptation of XtratuM to multi-core platforms, the hypervisor model has been re-designed to support the concept of virtual CPU (vCPU). Virtual CPUs are abstractions that model hardware CPU behaviour and are managed in an analogous way, but can be allocated to any of the existing cores. There are as many virtual CPUs on the system as physical cores and they behave on a similar manner: when a partition starts its execution, only one vCPU is active, being responsibility of the partition to initialize the remaining vCPUs. To this end, it has been necessary to extend XtratuM with new hypercalls that allow the partitions to manage virtual CPUs operation.

In a multi-core partitioned system partitions can use one core (mono-core partitions) or several cores (multi-core partitions). Several mapping schemes can be considered:

- Each monocore partition is mapped to one core
- Several monocore partitions are mapped to one core
- A mono-core partition is mapped to different cores at different time intervals
- A multi-core partition is mapped to several cores at the same temporal intervals

Figure 5 shows a configuration with a 3 mono-core and 1 multi-core partitions. Partitions P1, P2 and P3 allocates their virtual core (vCPU0) to one of the real cores (CPU0 or CPU1). P3 uses both cores at different intervals. P4 is a multi-core partition (two virtual cores) and requires the use of both real cores.

C. Hypervisor scheduling

XtratuM can associate a scheduling policy to each core or group of cores. Two scheduling policies are implemented: cyclic scheduling and priority based. The policy is statically specified in the configuration file. In the cyclic scheduling, the configuration file

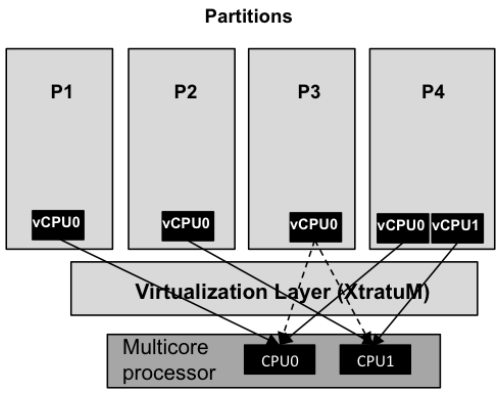


Fig. 5: Partitioned architecture in multi-core platforms.

specifies the temporal windows or slots in a major frame (MAF) where partitions will be scheduled. Each slot details the partition, the execution interval (as offset from the start of the major time frame and the duration) and the virtual CPU. Partitions definition includes the number of virtual CPUs to be used.

In case of priority based scheduling, partitions, allocated to the core that uses this policy, have to specify the priority, period and budget.

Next listing shows the specification of a cyclic schedule of four partitions in two cores according to the partition mapping shown in Fig 5.

XML configuration file: schedule specification

```
<ProcessorTable>
 <Processor id="0">
  <CyclicPlanTable>
   <Plan id="0" majorFrame="20ms">
    <Slot id="0" start ="0ms" duration="3ms" partitionId ="1" vCpuId="0"/>
    <Slot id="1" start ="3ms" duration="3ms" partitionId ="3" vCpuId="0"/>
    <Slot id="2" start ="7ms" duration="3ms" partitionId ="1" vCpuId="0"/>
    <Slot id="3" start ="12ms" duration="3ms" partitionId ="1" vCpuId="0"/>
    <Slot id="4" start ="15ms" duration="5ms" partitionId ="4" vCpuId="0"/>
   </Plan>
  </CyclicPlanTable>
 </Processor>
 <Processor id="1">
  <CyclicPlanTable>
   <Plan id="0" majorFrame="20ms">
    <Slot id="0" start ="0ms" duration="6ms" partitionId ="2" vCpuId="0"/>
    <Slot id="1" start ="7ms" duration="3ms" partitionId ="3" vCpuId="0"/>
    <Slot id="2" start ="10ms" duration="3ms" partitionId ="2" vCpuId="0"/>
    <Slot id="3" start ="13ms" duration="2ms" partitionId ="3" vCpuId="0"/>
    <Slot id="4" start ="15ms" duration="5ms" partitionId ="4" vCpuId="1"/>
   </Plan>
  </CyclicPlanTable>
 </Processor>
</ProcessorTable>
```

Figure 6 draws the execution chronogram of this example. P1 and P2 allocate their vCPU0 to CPU0 and CPU1, respectively. P3 allocates its vCPU0 to CPU0 and CPU1 at different time intervals (no overlap). P4 maps its virtual cores to the real cores at the same time intervals.

The configuration file is statically defined off-line. A set of techniques and tools are required to generate the schedule, according to the partition temporal requirements, criticality level, platform needs, etc., and to verify the coherence and correctness of the final schedule.

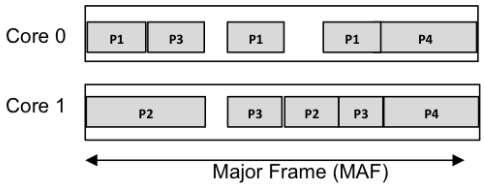


Fig. 6: Scheduling scheme.

In [17], a configuration and scheduling tool is presented. The hypervisor requires a binary representation of a verified configuration file to execute the system.

The main challenge in multi-core hypervisor is to deal with the temporal inteference when several cores are executing code at the same time. From the scheduling point of view, the temporal isolation in multi-core can consider two aspects:

- temporal allocation of systems resources: partition execution is statically defined (temporal windows).
- temporal interference: impact of the shared resources use by other cores.

While the hypervisor can guarantee the temporal allocation of resources, it requires the hardware support to deal with the temporal interference.

IV. Temporal Interference and Performance Analysis

In this section, the performance of the hypervisor layer comparing both architectures proposed in the MultiPARTES project for the AMBA bus: RR and TDMA is evaluated. The target is a LEON3 dual-core at 50MHz with DDR memory.

A. Temporal Interference Analysis

Temporal interference is produced when partitions in different cores use shared resources. We focus on this evaluation on the temporal impact that a target partition suffers when another partition is executed in other core and perform intensive access to memory.

To analyse this impact, a scenario with different levels of overlapping in partitions running in diffeent cores is defined. The scenario is defined with two partitions. P_1 is the target of evaluation and perform a fixed payload that is measured in an isolated environment. P_1 performs the following steps: read the clock (t_1), perform the payload, read the clock (t_2) and computes the differences $t_2 - t_1$ (execution time). P_2 is a dummy partition that performs a loop that read and modify the contents of a table. P_1 is executed in core 0 and P_2 is executed in core 1.

In order to analyse the effects in the worst conditions, cache management (instructions and data) is disabled for both partitions forcing both partitions to access physically to memory.

This scenario is executed under the following scheduling plan:

- MAF: 300 msec
- Payload cost 20 msec (approx).
- P_1 slot duration: 150 msec.
- P_2 slot duration: 150 msec.
- Experiments:
  - S0: No interference.
  - S25: 25% of interference.
  - S50: 50 % of interference.
  - S75: 75 % of interference.
  - S100: 100 % of interference.

Fig. 7 shows the schedule of S25.

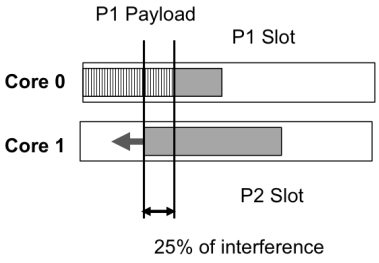


Fig. 7: Schedule of the S25

Table I shows the results of the impact of P_2 on the execution of P_1 in the LEON3 bi-core platform with RR bus arbitration. Under an approximated overlapping of the 25%, the increment of the execution time is in average 2899 μsecs (23544 - 20645). Next table presents the statistics of 100 executions of each scenario.

TABLE I: Impact of the memory accesses in the $P1$ execution time RR bus arbitration

Exec. Time (μsecs)	S0	S25	S50	S75	S100
Avg	20645	23544	27181	30577	33418
Max	20700	23557	27326	30614	33691
Min	20698	23305	27253	30535	33596
Stdev	0.63	12.46	16.77	17.19	22.81
Inteference	0%	14%	32%	48%	62%

Results show the impact of the interference when partitions allocated to different cores are executed with a level of overlapping. If both partitions are executed at the same time, the interference can produce an increment of 62% in the execution of P1.

Same experiment has been executed in the LEON3 bi-core platform with TDMA bus arbitration. Table II shows the results.

TABLE II: Impact of the memory accesses in the $P1$ execution time TDMA bus arbitration

Exec. Time (μsecs)	S0	S25	S50	S75	S100
Avg	120099	120099	120099	120099	120099
Max	120100	120100	120099	120100	120100
Max	120097	120096	120098	120098	120098
Stdev	0.99	1.03	0.97	0.96	0.96
hline Inteference	0%	0%	0%	0%	0%

These results show that the impact, as expected, depends on the overlapping interval and the bus arbitration policy. While the

temporal interference has a very high impact in the first hardware platform, the new design based on the TDMA arbitration policy fully achieves the temporal isolation of partitions. This is a relevant result for temporal and spatial partitioning platforms that permits to execute independently applications in multi-core systems.

B. Performance Analysis

In this section, a comparison of the performance of two hardware solutions is performed. Performance analysis includes two parameters: temporal cost of the same activity and hypervisor partition context switch in both platforms.

Table III summarises the temporal costs of the same payload executed in previous experiments.

TABLE III: Temporal cost comparison

	Avg Time (μsecs)
TDMA bus arbitration	120099
RR bus arbitration	20689
Increment cost factor	5.80

The use of a TDMA based arbitration policy introduces delays in the partition execution. For the configuration experimented in this paper, the computation time of a payload is incremented 5.8 times which can be relevant depending on the timing requirements of real-time tasks.

On the other hand, the partition context switch of the hypervisor measures the time required by the hypervisor to switch from one partition to another in a core. This cost has been experimentally measured by instrumenting the hypervisor code to annotate the entry and exit to the partition context switch service. Table IV shows the results in μsecss for both platforms.

TABLE IV: PCS comparison

	Avg Time (μsecs)
TDMA bus arbitration	1042
RR bus arbitration	224
Increment cost factor	4.65

The observed increment of the PCS is 4.65 times. While 224 μsecss for the partition context switch in space applications with period ranges in the order to dozens of milliseconds can be acceptable, the cost of 1 millisecond for the PCS introduces high overheads and reduces the period ranges to hundreds of milliseconds.

V. Conclusion

In mixed-criticality systems, critical applications are subject of certification. Without proper isolation of time-critical applications the process for certifying can become complex and time consuming.

In this paper we describe a memory architecture that provides temporal isolation between virtual partitions. Implementing the dual-layer memory-bus arbiter helps the hypervisor to guarantee time bounds for critical applications and to improve the performance of non-critical applications when they access the shared memory bus. We also demonstrate the feasibility of the proposed memory hierarchy by implementing it in an FPGA and running XtratuM on top of that hardware. The evaluation proves that even when critical partitions execute with full temporal overlap, the temporal properties of each of these partitions are preserved.

In future work we will run parallel applications on the multi-core hardware in order to evaluate the utilization of the hardware at the entire multi-core system level and experiment with the dual-layer arbiter.

Acknowledgment

This research was partially funded under the European Union's 7th Framework Programme under grant agreement no. 287702: Multicores Partitioning for Trusted Embedded Systems (MultiPARTES).

References

[1] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, 2010, pp. 13–22.

[2] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, 2011, pp. 34–43.

[3] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "Idamc: A many-core platform with run-time monitoring for mixed-criticality," in *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*. IEEE, 2012, pp. 24–31.

[4] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS). IEEE*, 2014.

[5] D. A. Patterson and J. L. Hennessy, "Computer organization and design," *Morgan Kaufmann*, 2007.

[6] M. Masmano, I. Ripoll, A. Crespo, and J.-J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, 2009.

[7] A. Gaisler and S. Göteborg, "Leon3 multiprocessing cpu core," *Aeroflex Gaisler, February*, 2010.

[8] A. A. Specification, "Multi layer ahb specification,(rev2. 0)," 2001.

[9] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of amba ahb from formal specification: a case study," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 585–601, 2013.

[10] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[11] Leon3. [Online]. Available: http://www.gaisler.com/

[12] M. Masmano, I. Ripoll, S. Peiró, and A. Crespo, "Xtratum for leon3: an open source hypervisor for high integrity systems," in *European Conference on Embedded Real Time Software and Systems. ERTS2 2010.*, Toulouse (France), 19-21 May 2010.

[13] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *Eleventh Real-Time Linux Workshop*, Dresden (Germany), 28-30 September 2009.

[14] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo, "Xtratum hypervisor redesign for LEON4 multicore processor," *SIGBED Review*, vol. 11, no. 2, pp. 27–31, 2014. [Online]. Available: http://doi.acm.org/10.1145/2668138.2668142

[15] "Multipartes: Multi-cores partitioning for trusted embedded systems," 2011. FP7 ICT 287702 European Project. (http://www.multipartes.eu).

[16] S. Trujillo, A. Crespo, and A. Alonso, "Multipartes: Multicore virtualization for mixed-criticality systems," in *2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013*, 2013, pp. 260–265.

[17] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, and P. Balbastre, "Xoncrete: a scheduling tool for partitioned real-time systems," in *Embedded Real-Time Software and Systems*, 2010.

Incorporating The Notion of Importance into Mixed Criticality Systems

Tom Fleming
Department of Computer Science,
University of York, UK.
Email: tdf506@york.ac.uk

Alan Burns
Department of Computer Science,
University of York, UK.
Email: alan.burns@york.ac.uk

***Abstract*—Mixed criticality systems offer the opportunity to integrate system components with different levels of assurance that previously may have been placed on different nodes. While the vast majority of mixed criticality work features a HI and a LO criticality level, LO criticality tasks should not be mistaken for tasks with little value. Such tasks might contain mission critical functionality and are still vital for the correct and efficient operation of the system. A large portion of earlier work immediately suspends all LO criticality functionality upon a criticality change. It is clear that suspending tasks at all is highly undesirable, let alone all tasks of a single criticality level at the same time. In this work we consider this issue, we propose a scheme to maintain the operation of lower criticality tasks for as long as possible, even when the system is executing in a higher mode. We introduce the notion of importance as a means of deciding which tasks are suspended first. This is done with the aim of allowing the system designer to make these decisions and have greater control over the way their system degrades. We conclude that by using essentially the same analysis and functionality that facilitates multiple criticality levels we are able to provide improved lower criticality performance.**

I. Introduction

The area of Mixed Criticality Systems builds upon an increasing desire to consolidate functionality of different criticality levels onto one platform. This is driven further by the development of more powerful hardware and pressure from industrial sectors, such as aerospace and automotive. A system that consolidates its functionality onto one platform looks to save space, power, weight and reduce hardware costs. Such systems must seek to satisfy two properties, efficient utilisation and isolation of high integrity tasks. Systems may be subject to certification against safety standards such as IEC 61508 or DO-178B, therefore the challenge is providing a means of producing a certifiable system with high overall utilisation.

The initial mixed criticality model proposed by Vestal [11] and used by others such as [3] has a strict notion of a criticality change. A dual criticality system begins executing in the LO criticality mode, if any HI criticality tasks overruns its LO criticality Worst Case Execution Time (WCET) then a criticality change occurs. LO criticality tasks are suspended (although active jobs are allowed to complete), HI criticality tasks are allowed to run to their maximum HI WCET. It is becoming increasingly apparent that the original notion of completely dropping all LO criticality tasks when a criticality change occurs is unacceptable. Although a task might be considered LO criticality, it might still contain mission critical functionality, as such these tasks should not be suspended unless absolutely necessary. It is also clear that simply dropping LO criticality tasks provides the system designer with little control over how their system degrades in the event of an overrun.

The work presented here seeks to address both the issue of immediately dropping all LO tasks and the lack of control over system degradation during overload. This is done by introducing the notion of importance, I. Importance levels are assigned to all tasks except those at the highest criticality level. Importance provides a greater degree of control and granularity over how a system degrades. This controlled degradation is facilitated by a more realistic view on the behaviour of a task during an overrun. When a task overruns its LO criticality WCET, it is unlikely that it will execute to its HI WCET. It is more likely that such a task might only overrun by a small margin. Rather than immediately dropping all LO criticality tasks our approach seeks to only drop tasks when absolutely required. Control over this degradation is given back to the system designer as the order in which tasks are dropped is determined by the assignment of importance (least important tasks are dropped first). We extend this further by considering groups of tasks as a single application. In this case applications are assigned an importance value, applications are dropped according to importance rather than individual tasks. We show the effectiveness of this technique via experimental results on randomly generated sets of tasks.

The remainder of this document is structured as follows: Section II covers related work, Section III introduces and describes Importance, Section IV evaluates the technique via experimental results and Section V provides some concluding remarks.

II. Related Work

In this section we consider related work where the primary focus is investigating the possible use of existing system slack to improve the level of service provided to LO criticality tasks.

Santy et al. [8] consider situations where LO criticality tasks do not need to be dropped during a criticality change. They showed that some slack often exists before any LO criticality task must be suspended. This period of slack is known as the 'allowance'. Allowance is shared across all HI criticality tasks, if it is used up LO criticality tasks must be suspended. Their work is based on the observations made about a previously developed technique OCBP [2], they claim that: it is never necessary to drop jobs that have a lower criticality but a higher priority than the current level. Although this assumption seems counter-intuitive, it must be noted

that OCBP is an older technique which considers a finite set of jobs where 'no job is allowed to execute for more than its WCET at is own specified criticality'[4]. In short the schedulability test considers all jobs executing in each level, jobs may execute up to their own $C_j(L_j)$ or their $C_j(L_i)$ where C_j is the WCET and L_j is the criticality level of task j. They use sensitivity analysis to calculate the allowance (slack) available for HI criticality tasks. On top of this, the possibility of returning the system to the LO criticality mode is considered. The system looks for a level-ℓ (where ℓ is the criticality level) idle time, a time where no jobs of at least criticality level ℓ are waiting to execute. They use this point as the time where the criticality of the system may be reduced.

Su and Zhu [9] consider an Earliest Deadline First based technique ER-EDF (Early Release - EDF). This technique seeks to allow LO criticality tasks to release earlier than their maximum period if there is slack available from the HI criticality tasks. This work is based upon an Elastic Mixed Criticality task model which allows for variable periods from a desired T_i to a maximum T_i^{max}. The minimum service requirement for LO tasks can be determined by its largest period, LO tasks may execute more frequently if there is no impact on HI criticality tasks. They evaluate their work by assuming reductions of 2 or 5 times to LO criticality periods. The performance of ER-EDF is compared against EDV-VD and is shown to have improved performance. Su et al. [10] extended this work to a multi-core platform.

Both of these techniques represent an attempt to provide some level of service for LO criticality tasks when a system enters the HI mode. Our work goes one step further, as well as considering the potential of using slack to schedule all LO tasks if the overrun is small we also consider the way and order tasks are dropped during a more severe overload.

III. Importance

In this section we will introduce and expand on the notion of importance. We use an adaptation of the standard system model initially defined in [11]. A system constitutes a finite set of applications K. Each of these applications is assigned a criticality level, L (designated by the system designer) and consists of a finite set of sporadic tasks[1]. Each task, τ_i, is defined as $\tau_i = \{\overrightarrow{C_i}, T_i, D_i, L_i\}$ where $\overrightarrow{C_i}$ is a vector of WCETs (one for each criticality), T_i is the period, D_i is the deadline and L_i is the criticality level. Each task gives rise to an unbounded series of jobs. Additionally we consider I_i as the importance of a LO criticality τ_i. Importance might also be assigned to LO criticality applications, which in turn applies this importance level to a group of tasks. In this work we constrain ourselves to consider only criticality dependent WCETs and dual criticality systems with $C_i(LO) \leq C_i(HI)$.

In this work we group sets of tasks at the same criticality level into applications. Applications are designed to better represent a more realistic system model where a group of tasks contribute to a single application. In this case if one task of an application must be suspended, then all other tasks associated with the application must also be suspended. Applications are assigned an importance value, rather than each task individually. It is worth noting that although tasks within an application are of the same criticality, priorities may be interleaved between applications and criticality levels. In the description and examples presented below we consider the case of systems composing of tasks rather than applications. This is to allow for simpler examples that show more clearly the effect of importance.

A. Overview

This work introduces the notion of *importance*, each task within the LO criticality level[2] is assigned an importance value, this provides an order in which tasks might be suspended during an overload. The purpose of this is to provide the designer of the system with more control over the graceful degradation of their system during an overrun. Whereas criticality levels typically involve the assignment of a SIL (Safety Integrity Level) or equivalent, importance is assigned according to how the designer wishes the system to degrade. The difference between criticality and importance is discussed[3] in [6] which considers the properties of a criticality change compared to other mode changes. As mentioned above, the typical response to a criticality change is to drop all tasks of a lower criticality. By assigning importance we suspend tasks in order, lowest importance first. This is done only when the HI criticality overrun is severe enough to warrant the suspension of a task. If the overrun is not severe and there is sufficient slack in the system, it is possible that the system might move into the HI criticality mode, while maintaining all of its LO criticality tasks. As soon as an overrun reaches the point in which a LO criticality task must be dropped, the task with the lowest importance is suspended. Importance provides an extra level of granularity within a level of criticality. The key difference between these two assignments is that criticality is typically assigned due to certification requirements, whereas levels of importance are decided by the system designer. We are using 'importance' as an *ordinal* scale [7] which allows questions such as 'is application A more important than application B?' to be answered. We do not extend the notion to an *interval* or stronger scale that would allow an answer to the following question to be used at run-time: 'Is application A and B more important than application C?'

Sensitivity analysis is used to determine the severity of the overrun required to drop a particular level of importance. Such analysis begins by checking if the system has any initial slack. It seeks to find a point during the HI criticality overrun at which the system is unschedulable and such a LO criticality task must be dropped. Once a point is found at which a task must be dropped, that point is recorded and the analysis begins to search for the next point by increasing the severity of the overrun. This process is repeated until a point is recorded at each time a LO criticality task must be dropped. It is possible that not all LO criticality tasks will be dropped, this depends on the properties of a task set, in particular its HI criticality utilisation. During runtime if a HI criticality task overruns its LO WCET up to the first recorded point, the

[1] All tasks within one application are of the same criticality level.

[2] For a dual criticality system, LO/HI, a system with greater than two criticality levels would see importance assigned within all but the highest level.

[3] Although importance is not named explicitly.

least important LO criticality task is suspended. This process continues if the overrun continues to increase.

We note here that in the case of high priority, high importance LO criticality tasks, we mostly consider their effect on the schedulability of the system even after they have been suspended. In order to do this we must consider their bounded interference on the task set up to the point that they are suspended. This is similar to the way AMC [3] (Adaptive Mixed Criticality) bounds the interference created by high priority LO criticality tasks during a criticality change. Consider the set of tasks in Table I:

	L	P	I
τ_k	LO	1	1
τ_j	LO	2	2
τ_i	HI	3	-

TABLE I
BASIC EXAMPLE.

Initially we consider the schedulability of the LO and HI mode as well as the criticality change [4]. The calculations for the LO mode and the change are shown in Equations (1) and (2) respectively, we exclude the HI mode as there is only one HI criticality task:

$$R_i(LO) = C_i(LO) + \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (1)$$

$$R_i(HI) = C_i(HI) + \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (2)$$

Such that:

$$R_i(HI) <= D_i$$

Where $R_i(HI)$ and $R_i(LO)$ are the response times of τ_i in the HI and LO criticality modes respectively.

We increase the overrun of τ_i from $C_i(LO)$ until we determine a point at which τ_j, the least important task, τ_j at $I = 2$, must be dropped.

$$\begin{aligned} R_i^{I_2}(LO) &= C_i^{I_2}(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \\ R_i(HI) &= C_i(HI) + \left\lceil \frac{R_i^{I_2}(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \end{aligned} \quad (3)$$

Next we attempt to increase the overrun until τ_k must be dropped. We use the response time just calculated, $R_i^{I_2}(LO)$ to bound the possible interference of τ_j.

[4]According to AMCrtb [3].

$$\begin{aligned} R_i^{I_1}(LO) &= C_i^{I_1}(LO) + \left\lceil \frac{R_i^{I_1}(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \\ R_i(HI) &= C_i(HI) + \left\lceil \frac{R_i^{I_1}(LO)}{T_k} \right\rceil C_k(LO) + \left\lceil \frac{R_i^{I_2}(LO)}{T_j} \right\rceil C_j(LO) \end{aligned} \quad (4)$$

In this way we account for the possible LO criticality, high priority interference up to the point at which a task is suspended.

B. Priority assignment

During sensitivity analysis, as the HI criticality tasks overrun is increased, it is likely that a particular LO criticality task will miss its deadline and cause the system to be seen as unschedulable. As we must drop our LO criticality tasks in order of Importance we may not be able to drop the task that might make the set immediately schedulable again. In extreme cases several other LO tasks might need to be dropped before the offending LO task can be dropped and the system can be seen as schedulable at a particular overload level. In a fixed priority system it is highly likely that the task which misses its deadline will be at the lowest priority. By slightly adapting Audsley's priority assignment technique [1] we can attempt to place tasks of lower importance at a lower priority. This is much the same as aiming to give lower criticality tasks lower priorities. This approach would work as follows:

For each priority level, beginning at the lowest. Check the schedulability of each task at this level. If more than one task is schedulable first differentiate by assigning the lower priority to the lower criticality. If many tasks that might be assigned a particular priority are of LO criticality, assign the priority to the task with the lowest importance value.

```
for Each priority level do
    for Each task do
        if criticalityLevel < currentTask then
            if importanceLevel < currentTAsk then
                currentTask=task;
            end
        end
    end
    Assign priority level to currentTask;
end
```

Algorithm 1: Audsley's Approach [1] with importance.

By assigning lower importance tasks lower priorities this reduces the chance of having to drop multiple tasks in order to make the system schedulable again.

C. Examples

A simple example can be used to illustrate the basic functionality of Importance. Consider the task set in Table II:

	C(LO)	C(HI)	T=D	L	P	I
τ_1	2	6	8	HI	1	-
τ_2	1	-	6	LO	2	1
τ_3	2	-	6	LO	3	2

TABLE II
A SIMPLE EXAMPLE.

If τ_1 were to exceed its $C_1(LO)$ by 2, τ_3 would have to be dropped as it would miss its deadline and cause the system to be unschedulable. Finally at an overrun of 4, τ_2 must be suspended. It is worth noting that such tasks would need to be suspended at time 3 for τ_3 and 5 for τ_2 if τ_1 does not signal completion at each of these times.

If the HI criticality tasks are also HI priority, they do not need to worry about interference from LO criticality tasks during their execution. Importance provides us with a set of points at which LO criticality tasks will be prevented from executing. These points give the system designer greater control over HI criticality degradation and allow the system resources to remain highly utilised. Crucially, regardless of the priority levels involved, this approach provides an improved level of service for LO criticality tasks, potentially reducing their likelihood of being suspended.

A second example can be used to highlight some interesting behaviour. Table III shows an example task set with importance assigned to the LO criticality tasks. Sensitivity analysis has been carried out on this set to determine the points at which each task must be dropped during an HI criticality overrun.

	C(LO)	C(HI)	T=D	L	P	I
τ_1	5	15	25	HI	3	-
τ_2	5	-	20	LO	4	3
τ_3	2	-	8	LO	1	2
τ_4	1	-	5	LO	2	1

TABLE III
A MORE COMPLEX EXAMPLE.

The least important task, τ_2 must be dropped when an overrun of HI criticality task τ_1 reaches 5 units of execution without signalling completion. In other words, τ_2 must be dropped as soon as an overrun is detected. Tasks τ_3 and τ_4 may continue to execute, if τ_1 does not complete after 10 units of execution, τ_3 and τ_4 must be suspended.

At each stage of the sensitivity analysis we re-check the schedulability of the system. For example if we assume an overrun of 1 to τ_1 then essentially we do the following calculation.

$$R_2(LO) = 5 + \left\lceil \frac{22}{25} \right\rceil 6 + \left\lceil \frac{22}{8} \right\rceil 2 + \left\lceil \frac{22}{5} \right\rceil 1 = 22 \quad (5)$$

The result of this shows that τ_2 will overrun its deadline if τ_1 exceeds its LO criticality execution by 1. As such it is clear that τ_2 must be suspended as soon as τ_1 reaches 5 units of execution without signalling completion.

The lowest priority task is now τ_1. At an overrun of 10 (9 without signalling completion) the task set is unschedulable.

$$\begin{aligned} R_1(LO) &= 10 + \left\lceil \frac{21}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 21 \\ R_1(HI) &= 15 + \left\lceil \frac{21}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 26 \end{aligned} \quad (6)$$

Thus giving the result of 26, and making an overrun of 10 with both tasks being unschedulable. In this case τ_3 is suspended leaving just τ_1 and τ_4 executing. If we just include these two tasks it would seem that τ_4 does not need to be suspended, the calculation would be as follows:

$$\begin{aligned} R_1(LO) &= 15 + \left\lceil \frac{19}{5} \right\rceil 1 = 19 \\ R_1(HI) &= 15 + \left\lceil \frac{19}{5} \right\rceil 1 = 19 \end{aligned} \quad (7)$$

However when calculating the schedulability of this situation we must include the prior interference from τ_3, we use the previously calculated LO response time at overrun 9[5] to cap the possible interference caused by τ_3.

$$R_1(LO) = 9 + \left\lceil \frac{19}{8} \right\rceil 2 + \left\lceil \frac{19}{5} \right\rceil 1 = \mathbf{19} \quad (8)$$

$$\begin{aligned} R_1(LO) &= 10 + \left\lceil \frac{\mathbf{19}}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 21 \\ R_1(HI) &= 15 + \left\lceil \frac{19}{8} \right\rceil 2 + \left\lceil \frac{21}{5} \right\rceil 1 = 26 \end{aligned} \quad (9)$$

Here it is clear that if τ_1 reaches an overrun of 9 without signalling completion both τ_3 and τ_4 must be suspended in order to allow τ_1 to meet its deadline. This can be shown further by considering the execution trace shown in Figure 1. This trace shows the situation where the interference from τ_3 is not considered. However, it is clear that τ_4 may not remain scheduled as the example then shows τ_1 executing until time 26 and exceeding its deadline. If both τ_3 and τ_4 were dropped after an overrun of 9[6], τ_1 would meet its deadline. As such, it is clear that when considering higher priority LO criticality tasks that are suspended, we must account for their impact throughout the execution. This is similar to including the bounded impact of LO criticality tasks on HI criticality tasks during a criticality change.

This example raises a very interesting point. As in the first instance, when trying to find a time in which τ_3 must be dropped we maximised the possible overrun of τ_1, it is clear that at this point τ_4 must also be suspended. This is because we must include the interference suffered from τ_3 until it is suspended. We know that all 3 tasks are not schedulable beyond point 9, as such both must be suspended at the same instant. This is due to the fact we must include the same amount of interference from τ_3 as the previous calculation. It seems likely that as we include interference from previously suspended higher priority LO criticality tasks, most higher priority LO criticality tasks will be dropped at the same instant. It is worth noting that even if such high priority LO tasks are only able to remain schedulable up to a HI criticality overrun of 10%, the likelihood of the system overrunning by 10% may be relatively low.

D. Further points

One of the fundamental assumptions of this work is that HI criticality overruns are not likely to be as severe as their HI WCET suggests. To support this assumption we can look

[5]The last schedulable point before suspension.
[6]Time 19.

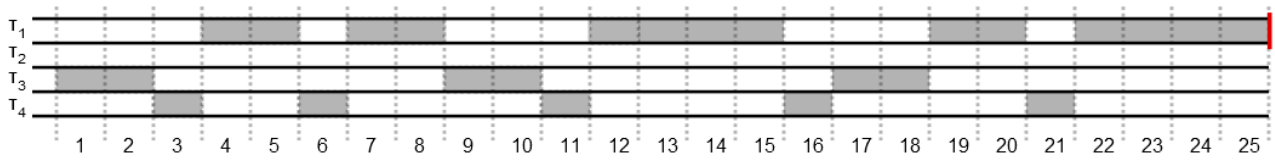


Fig. 1. An execution trace of Table III.

at work carried out on probabilistic real-time systems. Rather than the traditional view of a LO and HI criticality value, this work considers the space between the two values as a large number of points. Each point has its own degree of confidence. This set of points is known as the pWCET distribution. If such a pWCET distribution for a system showed that the likelihood of a task overrunning by more than 60% was extremely small (for example: 10^{-9} failure rate per hour), then simply dropping all LO criticality tasks when a criticality change occurs is poor use of the system resources. Importance is able to improve on this by allowing LO criticality tasks to continue execution, providing they do not effect the execution of HI criticality tasks. Although some LO criticality tasks may have to be dropped, it is likely that a good percentage of these tasks will be able to continue to execute throughout the HI criticality mode. If such a pWCET is known for a system, it would be possible to make predictions on the likelihood of a particular task being dropped. In this way the use of importance and probabilistic reasoning could help provide more detailed guarantees of system performance.

Importance is a useful means of providing a more detailed picture of system performance under HI criticality/overload conditions while passing more control over system degradation to the designer.

IV. Evaluation

The notion of importance is relatively easy to explain, however, its effectiveness is not so easy to quantify. As schedulability is not improved via the use of this technique another means of showing its effectiveness is required. In our evaluation we firstly illustrate simply how our approach is able to stagger the process of dropping LO criticality tasks and secondly we consider a probabilistic view that considers the severity of an overrun. Both illustrations show how our approach is able to reduce the chance of LO criticality tasks being suspended.

Our experimental data was produced from 10,000 randomly generated tasks with a total LO utilisation of 85%, these were created as follows. Utilisation values were generated via the UuniFast Algorithm [5], periods between X and Y were generated in a log uniform distribution. Our task sets are dual criticality, $C(LO)$ values were created from the periods and utilisations generated $C = U^*T$, $C(HI)$ values were 2 times $C(LO)$. 25 LO criticality tasks and 5 HI criticality tasks were generated per task set. The LO criticality tasks we randomly grouped into 5 applications (5 tasks per application), each application was randomly assigned a level of importance. HI criticality tasks were left as individual tasks rather than applications and no assignment of importance is required for this level. Priorities were assigned via our version of Audsley's algorithm [1] as seen in Section III, part B.

It is worth noting that in experiments such as this, there are a huge number of parameters which will affect what the results look like. The total number of tasks will effect the results, as will the distribution of these tasks between HI and LO criticality. The relative utilisation of the HI and LO modes has a big impact as to when LO criticality applications must be dropped, as does the difference between a HI criticality task, τ_i's, $C_i(LO)$ and $C_i(HI)$. We have described the values we used in our experimentation, different parameters will produce different looking graphs. However, the key result remains the same regardless of the parameters used, introducing levels of importance will provide a better level of service for LO criticality tasks.

In our work we also introduced the notion of groups of tasks as applications. Tasks of one application share a level of importance and will therefore be suspended as a group. It is worth noting that although applications share a level of importance, the priorities of individual tasks may be interleaved. The purpose of this is to better capture the nature of applications as groups of tasks, although these tasks might be interconnected we only consider independent tasks in this work.

Our experiments firstly ran each generated task set through the schedulability test AMCrtb [3] to ascertain schedulability and if schedulable, provide a priority ordering. Each task set that passed this test was then run through sensitivity analysis to determine the points at which LO criticality applications must be dropped, these points were recorded and used later to present the results. In some cases a task set might only need to drop one application and in others it might need to drop all 5. During sensitivity analysis all HI criticality WCET values are increased by the same percentage, this seemed like a reasonable assumption as it is difficult to model the likelihood of each HI criticality task individually overrunning. On top of this, a single HI criticality task overrunning its LO WCET is not likely to have that much of an impact on the system, especially as our HI criticality tasks do not have particularly high utilisations individually.

Figure 2 shows the number of applications dropped on the Y axis against the severity of the overrun as a percentage increase from $C(LO)$ on the X axis. The graph clearly shows that our approach is able to maintain LO criticality functionality for a significantly increased amount of time. This is even more apparent if you consider that the probability of an overrun occurring even beyond the 5% initial slack becomes exponentially more unlikely. Figure 3 shows the potential likelihood of an overrun reaching each point and causing a task to be suspended. The probabilities used here are merely designed to illustrate the point and are not meant as realistic values.

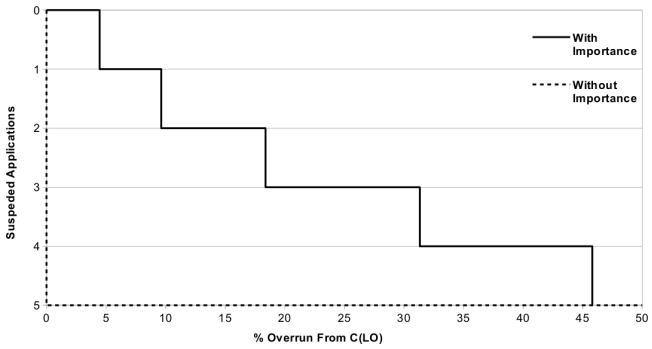


Fig. 2. Results from 10,000 random task sets.

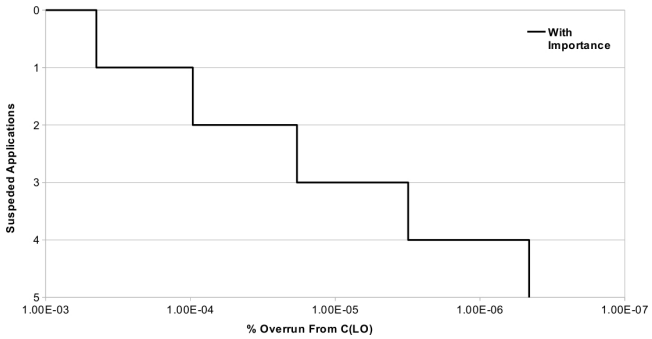


Fig. 3. Example probability of overrun in log scale.

If we consider Figure 3 with a linear scale it is easy to see that, even if the system can maintain all LO tasks during an overrun of 5%, this is still a significant improvement when taking into account the probability of the overrun actually reaching that level. This is shown in Figure 4:

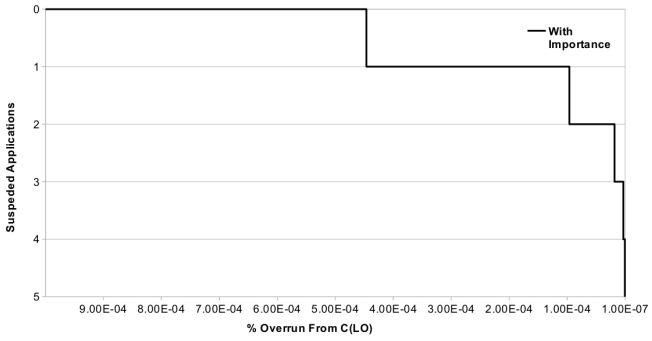


Fig. 4. Example probability of overrun in linear scale.

V. Conclusion

It is clear that as we move to consider more realistic mixed criticality implementations, simply dropping LO criticality tasks when a criticality change occurs is unacceptable. In this work we have introduced the notion of importance, we discussed the reasoning and illustrated the benefits through discussion and experimental results. Importance provides the designer of a system with a greater level of control and knowledge over the likely behaviour of their system during a criticality change. We show the effectiveness of importance by considering the reduced number of tasks dropped and the increased HI criticality system utilisation. This is done via experimental evaluation on randomly generated task sets. During the experimentation we introduced the notion of several tasks grouped as applications, applications aim to provide a more realistic system model. Further work might consider importance at greater than two criticality levels or it might consider a means of re-introducing LO criticality tasks when recovering from an overrun. To summarise, we have introduced importance as a means to provide a greater level of control and guarantees for LO criticality tasks during a criticality change.

Acknowledgements

The authors acknowledges the support and funding provided for this work by BAE Systems, and the ESPRC (UK) via MCC grant (EP/K011626/1).

References

[1] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

[2] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. In P. Hlinn and A. Kuera, editors, *Mathematical Foundations of Computer Science 2010*, volume 6281 of *Lecture Notes in Computer Science*, pages 90–101. Springer Berlin Heidelberg, 2010.

[3] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34 –43, 29 2011-dec. 2 2011.

[4] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13 –22, april 2010.

[5] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[6] A. Burns. System mode changes - general and criticality-based. volume WMC RTSS 2014, 2014.

[7] D. Prasad, A. Burns, and M. Atkin. The measurement and usage of utility in adaptive real-time systems. *Journal of Real-Time Systems*, 25(2/3):277–296, 2003.

[8] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 155 –165, july 2012.

[9] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 147–152, 2013.

[10] H. Su, D. Zhu, and D. Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 352–357, Aug 2013.

[11] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239 –243, dec. 2007.

Scheduling Mixed-Criticality Real-Time Tasks with Fault Tolerance

Jian (Denny) Lin[1], Albert M. K. Cheng[2], Douglas Steel[1], Michael Yu-Chi Wu[1]

[1]Department of Management Information Systems, University of Houston - Clear Lake, Houston, TX 77058 USA
[2]Department of Computer Science, University of Houston, Houston, TX 77024 USA

***Abstract* - Enabling computer tasks with different levels of criticality running on a common hardware platform has been an increasingly important trend in the design of real-time and embedded systems. On such systems, a real-time task may exhibit different WCETs (Worst Case Execution Times) in different criticality modes. It has been well-known that traditional real-time scheduling methods are not applicable to ensure the timely requirement of the mixed-criticality tasks. In this paper, we study a problem of scheduling real-time, mixed-criticality tasks with fault tolerance. An off-line algorithm is proposed to enhance the performance of the system when it runs into a high-criticality mode from a low-criticality mode. A novel on-line slack-reclaiming algorithm is also proposed to recover from as many faults as possible before the jobs' deadline. Our simulations show that an improvement of about 30% in performance can be seen between our algorithm and a regular slack-reclaiming method.**

I. Introduction

THE integration of multiple functionalities on a single hardware platform is an increasing trend in the design of embedded systems with the consideration of reducing cost. While the tasks running on these systems share resources they do not share the same importance (criticality). Therefore, the concept of *mixed-criticality* has risen. Some widely-discussed cases about mixed-criticality are the application domains that need to be certified correct by Certification Authorities (CA's) [1]. In these cases, different computer tasks performed on the same system require different levels of assurance. This difference probably produces different WCETs of the the critical tasks estimated by the CA's and the system designers. The CA's may be more concerned about some tasks' assurance than the system designers do. As a result, it brings issues to the scheduling of real-time tasks with different levels of criticality in a timely manner. It has been well-known that conventional scheduling methods cannot satisfactorily address these issues.

Mixed-criticality systems recently become one of the research focuses in the community of real-time and embedded systems. For examples, Sanjoy Baruah *et al.* demonstrate the intractability of determining whether a mixed-criticality system can be scheduled to meet all its certification requirements, and then two scheduling techniques are proposed to scheduling such mixed-criticality systems [1]. De Niz *et al.* study the *criticality inversion* problem and propose a new scheduling scheme called *zero-slack scheduling* which can be used with priority-based preemptive schedulers (e.g., RMS) [2]. Later, the work is extended to work with solutions for a distributed system [3]. For solving the problems under a dynamic priority scheduler EDF (Earliest Deadline First), Baruah *et al.* [4] propose an effective and efficient scheduling algorithm, namely EDF-VD (virtual deadline), in which two different deadlines are used for some tasks if they may exhibit two different WCETs during the run-time. In order to guarantee the timeliness of high-criticality tasks most of existing algorithms completely sacrifice the executions of low-criticality tasks [1-6]. This strategy is too conservative and not necessary in most cases. Also, too many jobs abandoned can seriously degrade the system's performance or even cause service abrupt.

Real-time systems not only have temporal constraints to meet, computation quality constraints are also clearly important for a system with critical tasks. In a mixed-criticality system, faults or errors may happen during tasks' execution which can either produce incorrect results or cause critical tasks to miss deadlines, both of which may be catastrophic. It has been shown that in a computer system transient faults occur much more frequently than permanent faults do [7, 8]. Transient faults can be tolerated by adding redundancy where a task will be re-executed if it completes with errors. There is little work to study both fault-tolerance and mixed criticality systems. In [14], the author studies the fixed-priority schedulablity test condition for a mixed-criticality system. In [15], Huang et al describes a method to convert the fault-tolerant problem into a standard scheduling problem in a mixed-criticality system.

In this work, we consider a problem that schedules a set of real-time, fault-tolerant tasks in a mixed-criticality system using EDF. Each task in the system is periodic and characterized by a 4-tuple of parameters: $\tau_i = (p_i, X_i, c_i(LO), c_i(HI))$. Without loss of generality, all tasks are assumed to be active at time 0. The p_i is a period which is the length of the interval between any two τ_i's consecutive job releases and also the relative deadline of the task. The $X_i \in \{LO, HI\}$ denotes the criticality of τ_i. A HI-criticality task τ_i may exhibit two WCETs $c_i(LO)$ and $c_i(HI)$ during the run-time where $c_i(HI) \geq c_i(LO)$. A LO-criticality task τ_i has only the $c_i(LO)$ defined and its $c_i(HI)$ is defined as *none*. After a system starts to run, all tasks have an infinite sequence of jobs to execute. The j^{th} job of τ_i is released at $(j-1)p_i$. Initially, all HI-criticality and LO-criticality tasks in the system are scheduled using their $c_i(LO)s$ and in this stage the system is said to be in the LO-criticality mode. During the execution, a HI-criticality task may signal that its execution time exceeds its $c_i(LO)$. At this point, all HI-criticality tasks will assume

[2]Supported by the National Science Foundation under Awards No. 0720856 and No. 1219082.

their $c_i(HI)s$ and the system will go into the HI-criticality mode. In order to enhance the system's reliability, the primary and re-execution approach [9, 10] is used and a re-execution may be performed after errors are detected at the completion of the primary execution.

II. An Off-Line Algorithm

In our work, we adopt similar notations as used in [4] for utilization parameters for tasks. A general format is defined as follows:

$$U_x^y(a) = \sum_{a \in \{pri,re\} \wedge X_i = x} \frac{c_i(y)}{p_i} \tag{1}$$

The superscript and the subscript next to the U denote the system mode and type of task, respectively. If with a, it indicates that it is for primary execution or re-execution.

In [4], without a consideration of primary or re-execution, a virtual deadline used for each HI-criticality task in the LO-criticality mode is obtained off-line with a scaling factor x, where the virtual relative deadline of the HI-criticality τ_i is equal to $x \times p_i$. The x is a value defined as:

$$x \in [\frac{U_{HI}^{LO}}{1 - U_{LO}^{LO}}, \frac{1 - U_{HI}^{HI}}{U_{LO}^{LO}}] \tag{2}$$

Any value in the range can be used for x. When a system is signaled with the HI-criticality mode, all HI-criticality tasks τ_i use their $c_i(HI)$ and all LO-criticality tasks are abandoned immediately. If such a range is not found, for example, $\frac{U_{HI}^{LO}}{1-U_{LO}^{LO}} > \frac{1-U_{HI}^{HI}}{U_{LO}^{LO}}$, the HI-criticality tasks cannot be guaranteed to meet their deadline during the mode conversion.

The aspect of our work is different. We assume that the low-criticality tasks are not trivial but the reliability of them can be traded for schedulability. In our method, we not only assure the primary execution and re-execution for all HI-criticality tasks, but also maximize the executions of LO-criticality tasks. If the resource is sufficient, the re-executions for LO-criticality tasks are scheduled as well in order to enhance the overall reliability.

In this section, we introduce an off-line algorithm to calculate (maximize) the number of tasks, including HI- and LO-criticality tasks, with the primary and re-executions reserved. The system starts with every task having both primary and re-executions reserved in the LO-criticality mode. HI-criticality tasks are required to reserve their primary execution and re-execution in HI-criticality modes to maintain high reliability. Next, we try to reserve the primary executions then the re-executions of LO-criticality tasks. During the HI-criticality mode, the primary and re-executions of the HI-criticality tasks have the equal importance and they must be reserved, but for LO-criticality tasks the primary executions are reserved before the re-executions. The re-executions of LO-criticality tasks are reserved only after all other primary executions are reserved. The order to reserve the executions is as follows: primary and re-executions of HI-criticality tasks, primary execution of LO-criticality tasks and then re-execution of LO criticality tasks. For the problem to be non-trivial, we assume that the re-executions cannot be guaranteed for all LO-criticality tasks in the HI-criticality mode.

According to the schedulability test of EDF, for the system schedulable in the LO-criticality mode with every task's primary and re-execution reserved, the total utilization is at most 1:

$$U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re) \leq 1 \tag{3}$$

During the execution, the system goes into the HI-criticality mode if there is any HI-criticality task exceeding its $c(LO)$. We call the event as mode conversion. In a mode conversion, all HI-criticality tasks must maintain their fault-tolerance and both of their primary and re-execution will use their $c(HI)$ for the WCET. In order to walk through the mode conversion, the x is calculated as in (2), where $x \times d_i$ is used for the HI-criticality task's relative deadline in LO-criticality mode. It switches back to the original relative deadline while the system goes into the HI-criticality mode. The following conditions calculate the two boundaries in (2) for the scaling factor to ensure that all HI-criticality tasks' primary and re-execution are reserved in the HI-criticality mode without missing their deadlines:

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)}{1 - U_{LO}^{LO}(pri) - U_{LO}^{LO}(re)} \tag{4}$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re))}{U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)} \tag{5}$$

An important observations in above inequalities is that there is a room between these two boundaries so that it can be used to keep some LO-criticality tasks' executions in the system's HI-criticality mode. The significance of utilizing this room is two-fold. First, having more executions of LO-criticality tasks performed increases a system's overall value and improves the system's performance. Second, when more tasks have a re-execution reserved it makes the system more reliable and predictable. With doing a small modification, the inequalities 4 and 5 can be re-written for including the reserved re-executions for some LO-criticality tasks in the HI-criticality mode:

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri)^{'}}{1 - U_{LO}^{LO}(pri)^{''} - U_{LO}^{LO}(re)} \tag{6}$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re) + U_{LO}^{LO}(pri)^{'})}{U_{LO}^{LO}(pri)^{''} + U_{LO}^{LO}(re)} \tag{7}$$

The $U_{LO}^{LO}(pri)^{'}$ in inequality 6 and the $U_{LO}^{LO}(pri)^{''}$ in inequality 7 denote, of the LO-criticality tasks, the utilization for the reserved primary execution and the utilization of the primary execution not reserved, respectively. If the resource sufficiently allows all LO-criticality tasks to have their primary executions reserved, we can try to reserve more re-executions for LO-criticality tasks.

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)^{'}}{1 - U_{LO}^{LO}(re)^{''}} \tag{8}$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)^{'})}{U_{LO}^{LO}(re)^{''}} \tag{9}$$

The $U_{LO}^{LO}(re)^{'}$ in inequality 8 and the $U_{LO}^{LO}(re)^{''}$ in inequality 9 denote, of the LO-criticality tasks, the utilization for the reserved re-execution and the utilization of the re-execution not reserved, respectively.

We notice that when an execution of a LO-criticality task is moved from a denominator to a numerator, The gap between the two boundaries is narrowed. A similarity can be found between the problem and the bin-packing problem. Intuitively, the smallest first approach is a good way to optimize the solution. We demonstrate our solution in Algorithm 1 (Max. Re-executions). In the input, U_1 represents the numerator of inequality 6 or 8, U_2 represents the sub-tractor in the numerator of inequality 7 or 9, and U_3 is equal to the remaining, total utilization for un-reserved executions. The x_1 and x_2 are the left and right boundaries of x and when the final boundaries are found we set $x = x_2$. Utilizations of LO-criticality tasks that are not reserved are sorted from small to large as part of the input. The primary executions are allocated for reservations before the re-executions. Line 8 and Line 9 tell that before the loop starts if $x_2 < x_1$, the system is not able to reserve all HI-criticality tasks' primary executions and re-executions. From Line 11 to Line 24, the loop adds one primary execution of a LO-criticality execution a time, from small to large, to the reserved ones. If all LO-criticality primary executions can be reserved, it will try to reserve more LO-criticality re-executions. It calculates x_1 and x_2 in each iteration by using the updated utilizations. The x is returned while the gap between x_1 and x_2 is minimized. If a system is schedulable with the virtual deadlines, during LO-criticality mode the value of $x \times d_i$ is assigned as the relative deadlines to all HI-criticality tasks for both primary and re-executions, and to all LO-criticality tasks for their reserved executions. The executions from LO-criticality tasks that are not reserved, use their original deadlines in the LO-criticality mode and are abandoned or run in background in the HI-criticality mode.

Compared with using the approaches discarding all LO-criticality tasks, Algorithm 1 not only guarantees the HI-criticality tasks' deadlines and fault-tolerance in both HI and LO-criticality modes, some LO-criticality tasks' executions are also reserved if the resource is sufficient. The guarantee of meeting the reserved executions' deadlines during both criticality modes and the mode conversion is proved in Theorem 1.

***Theorem*1**: There is no deadline miss if it uses the x returned by Algorithm 1 to calculate the virtual deadlines used in LO-criticality mode for all reserved executions and discard the un-reserved re-executions during the HI-criticality mode.

The proof is supported by two facts. The first one is the mapping of our task set to the task set used in the proof in [4] to support the validity using virtual deadlines. Algorithm 1 in fact makes the following executions' behavior as HI-criticality: primary and re-executions of HI-criticality tasks, and reserved LO-criticality executions. By using the calculated virtual deadlines, every task's reserved execution does not miss its deadline. The correctness directly follows the Theorem 1 and Theorem 2 in [4]. The second fact is that by using EDF, any reduction of execution does not cause a deadline violation. In our problem, if a primary execution completes correctly, its reserved re-execution is not performed. This is the same as reducing the re-execution's running time to be zero which will not cause any deadline violation.

Input : τ, Γ, ξ, U_1, U_2, x_1, x_2;
Output: the scaling factor x;
1 initialization:
2 $\Gamma = \tau_{HI}(pri) \cup \tau_{HI}(re)$, $\xi = \tau_{LO}(pri) \cup \tau_{LO}(re)$
3 $U_1 = U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)$
4 $U_2 = U_{HI}^{HI}(pri) + U_{HI}^{HI}(re)$
5 $U_3 = U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)$
6 ξ is sorted from small to large using utilization for the primary and re-executions, respectively. The primary executions are ordered before the re-executions.
7 $x_1 = \frac{U_1}{1-U_3}$, $x_2 = \frac{1-U_2}{U_3}$, x=x_2;
8 **if** *($x_2 < x_1$)* **then**
9 | not schedulable;
10 **else**
11 | **while** $|\xi| > 1$ **do**
12 | | U_i = the first execution's utilization in ξ ;
13 | | $U_1 = U_1 + U_i$;
14 | | $U_2 = U_2 + U_i$;
15 | | $U_3 = U_3 - U_i$
16 | | $x_1 = \frac{U_1}{1-U_3}$, $x_2 = \frac{1-U_2}{U_3}$;
17 | | **if** $x_1 \leq x_2$ **then**
18 | | | $\Gamma = \Gamma \cup \tau_{LO}(U_i)$;
19 | | | x = x_2;
20 | | | $\xi = \xi - \tau_{LO}(U_i)$;
21 | | **else**
22 | | | return x;
23 | | **end**
24 | **end**
25 | return x;
26 **end**

Algorithm 1: Maximize Re-executions

	p	X	c(LO)	c(HI)	x	d_{pri}	d_{re}
τ_1	30	HI	3	4.5	0.8	24	24
τ_2	100	HI	5	12	0.8	80	80
τ_3	200	LO	10	none	0.8	160	160
τ_4	50	LO	3	none	0.8	40	50
τ_5	50	LO	7	none	0.8	40	50

Table 1. Calculated relative deadlines of primary and re-executions of a 5-tasks set

Table 1 gives an example of using Algorithm 1 to calculate the virtual deadlines for a task set of 5 tasks. In the task set, there are two HI-criticality tasks and three LO-criticality tasks. It is not possible to schedule all tasks for guaranteeing both of their primary and re-executions during the HI-criticality mode because the total utilization is larger than 1. Instead of discarding all LO-criticality tasks in the HI-criticality mode, we can ensure all of the primary executions for all tasks and the re-executions of τ_1, τ_2 and τ_3. The calculated virtual deadlines are shown in the table.

III. An On-Line Slack Reclaiming Algorithm

The strategy of using re-executions to preserve the reliability is relatively conservative. It is expected that most job instances in a system do not produce errors and therefore the resource for the reserved re-executions is wasted. Also, it is well adopted that in most of the time a real-time task does not use up its WCET. Clearly, slack is generated in these two cases. When a fault is detected by a LO-criticality job without a re-execution reserved, the slack may not be large enough to re-execution. However, if we accept a small probability of a sacrifice of the predictability, we may be able to borrow some slack from a future execution to increase the amount of the available slack at that time. Figure 1 shows an example of

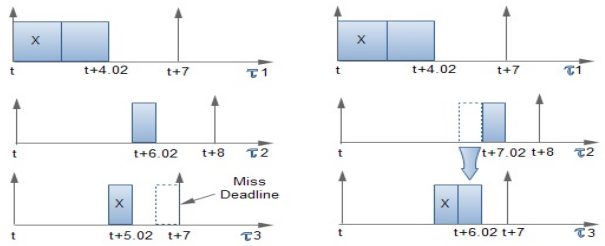


Fig. 1. Borrowing another job's future re-execution to the current re-execution

three tasks in the HI-criticality mode: τ_1 is a HI-criticality task with $c_1(HI) = 2.01$, and τ_2 and τ_3 are two LO-criticality tasks with the same $c(LO) = 1$. Both τ_1 and τ_2 have a re-execution reserved. Assume that time *t* is a LCM (Least Common Multiple) of the three tasks and their periods are 7, 8 and 7. So, for each task there is a job arriving at *t*. The HI-criticality mode is assumed to be the mode at and after time *t*. Now suppose that both the first jobs of τ_1 and τ_2 in the figure have a fault detected at their completion. The left schedule shows that the first job of τ_3 will miss its deadline at time t+7 if a re-execution is performed using the slack generated from the no-operation of τ_2's re-execution. The right schedule shows a possible solution that a future re-execution of τ_2 can be borrowed to re-execute the job of τ_3 early. If the job of τ_2 completes without errors, no impact is posed to it. Even if the job of τ_2 ends with an incorrect result due to a lack of the re-execution, the consequence is not catastrophic because both jobs are not HI-criticality. Considering the small likelihood of a fault actually occurring, the idea behind this solution has the potential to increase the overall reliability and system's performance.

Now, we are ready to present our on-line slack reclaiming algorithm. When an additional re-execution not reserved is needed to perform, the slack generated from the early completion of jobs (including no-operations of reserved re-executions) and system idle time are firstly used because they are most safe. A borrowing from the future is performed only if the slack in the previous cases is not sufficient. The borrowing only happens at a future re-execution of a LO-criticality task because HI-criticality tasks require the highest level of fault-tolerance. Algorithm 2 is based on the approach in [5, 6] (CASH) and a modification to allow the borrowing of future re-executions is applied.

1) Each task τ_i is associated with a server S_i characterized by a current budget b_i and an ordered pair (q_i, p_i), where q_i is the maximum budget and p_i is the period of the server and the task. If a task has a re-execution statically reserved, $q_i = 2 \times c_i$; otherwise, $q_i = c_i$.
2) When the system in the LO-criticality mode, the c_i for a HI-criticality task is the LO one, and the p_i is equal to its virtual relative deadline. If a mode conversion occurs, all HI-criticality tasks will use their HI-criticality execution time for c_i and their original relative deadline as p_i.
3) At each time $(k-1) \times p_i, k \in \mathbb{N}^+$, the server budget b_i is recharged at the maximum value q_i and the new deadline of the server is $d_{i,k} = k \times p_i$. The $\tau_{i,k}$ with a total execution time equal to b_i is inserted into a waiting queue, and scheduled using the server's deadline.
4) A server S_i is said to be active at time *t* if there are jobs associated with it pending; otherwise, it is said to be idle.
5) There is a slack queue such as the one used in [11]. Whenever $\tau_{i,k}$ is scheduled for execution, it always uses the slack with the earliest deadline such that $d_q \leq d_{i,k}$. Otherwise, its own budget b_i is used. When $\tau_{i,k}$ is executed, the slack in the slack queue or the server budget is decreased by the same amount. When a slack capacity in the queue becomes zero, it is removed from the queue.
6) When a fault of a completion of $\tau_{i,k}$'s primary execution is detected at *t*, the total execution time of $\tau_{i,k}$ is increased by c_i.
7) When the server S_i is active and b_i becomes zero at *t*, it finds the first LO-criticality job with an execution reserved in the queue which has not finished its primary execution (if any). Then, do the following:
   a) The job's server donates a budget of its re-execution to S_i,
   b) S_i's deadline is set to $d_d - c_d^{'}$, where d_d is the deadline of the donating server of the job and $c_d^{'}$ is the remaining primary execution time of the server from the donating task.

   If no such LO-criticality jobs in the waiting queue exist, the re-execution of $\tau_{i,k}$ only runs while the system becomes idle.
8) When a job's deadline is reached, even though it has not been finished yet, the job is terminated and its new instance in the next interval starts as in step 3.
9) When a job finishes or is terminated, the residual budget of its server, if any, is inserted into the slack queue using its server's deadline.
10) Whenever the system becomes idle for an interval of time, the slack with the earliest deadline, if any, in the slack queue is decreased by the same amount of time until the queue becomes empty.

Algorithm 2: CBS-FT (CBS-Fault Tolerance)

The step 7 states how the server S_i borrows and uses a

capacity of time from a donating server by setting a new deadline, $d_d - c_d^{'}$, which is the latest time to use that capacity. We restrict to use our on-line algorithms in the cases where all LO-criticality tasks have their primary executions reserved. Otherwise, we just execute the un-reserved executions using the regular slack.

We use the same example in Figure 1 to explain the process. At time t+5.02, a fault is detected for τ_3 and another WCET (1.0) is added to the execution time requirement. At this point S_3's server budget is equal to zero and it finds S_2 from the waiting queue. S_2 donates a capacity of time of 1.0 to S_3 so that now $b_3 = 1$ and $b_2 = 1$. S_3's new deadline is set to be t+8-1=t+7 so τ_3 is scheduled before τ_2 and the re-execution of τ_3 completes in time. Later, τ_2 completes its primary execution correctly and the budget for its reserved re-execution is not wasted. In this case, all tasks finally complete with the correct results.

IV. Perfomance and Discussions

In this section, we perform simulations to evaluate the performance of our off-line Max. Re-executions and on-line CBS-FT algorithm. For the algorithm Max. Re-execution, each time a task set of ten periodic tasks is randomly generated. The total utilization of the task set is larger than 1.0 so that not all tasks have their primary and re-executions reserved. In these ten tasks, four are HI-criticality and the other six are LO-criticality. We apply the off-line Max. Re-execution algorithm to the task set so that the reserved executions for the LO-criticality tasks can be maximized. We perform this experiment 100 times and each time the number of reserved primary and re-execution for the LO-criticality tasks are recorded. The results are shown as in Figure 2 and Figure 3. It can be seen that instead of discarding the executions of all LO-criticality tasks, we can nicely reserve a great number of their primary executions. The average is 4.26 of 6 LO-criticality tasks. Since we reserve a re-execution only if all primary executions of the LO-criticality tasks are reserved, the average number of the reserved re-execution is relatively small as 1.32 task per 6 tasks. Despite, there are about 40% of the cases in which at least one LO-criticality task becomes fault-tolerant during the HI-criticality mode.

While all primary executions are reserved, we can use the *borrow from future* technique as implemented in Algorithm 2. A task set of five periodic tasks is generated randomly and the task set is selected if all tasks have their primary execution reserved. Among the five tasks, two are randomly selected as HI-criticality tasks and the rest are LO-criticality tasks. Algorithm 1 is used to determine which re-executions are reserved. A valid test case used to evaluate the Algorithm 2 should contain at least one LO-criticality task with a re-execution reserved. The tasks' periods range from 30 to 200 and they are scheduled within an interval of one million time units. Within that interval each time the total number of job instances generated is expected between 60,000 to 100,000. A fault occurrence rate between 0.05 and 0.5 is used to control whether or not a job completes with a fault in its primary execution. A fault is recorded if the job's re-execution does not successfully complete. For each experiment, we perform it 10 times using the same parameters and the average is used for our results.

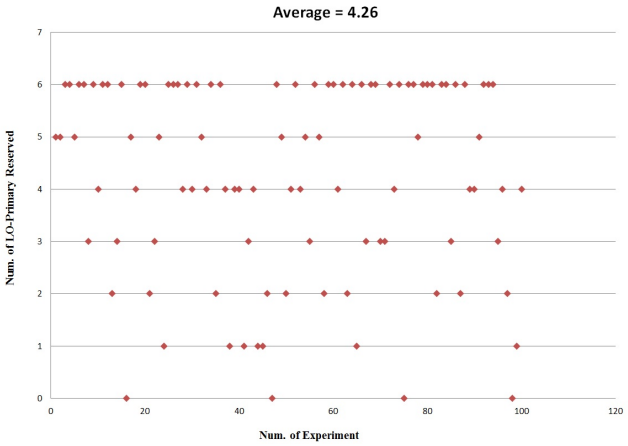


Fig. 2. The Number of Reserved Primary Executions using Algorithm 1

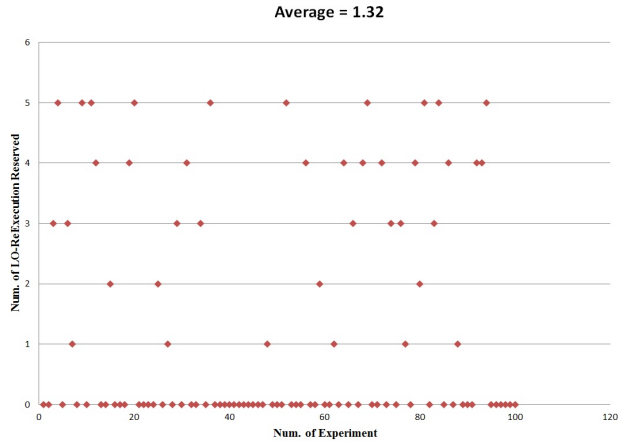


Fig. 3. The Number of Reserved Re-Executions using Algorithm 1

In the experiments, the fault rate has a big impact to our results and another factor is the lower bound of each job's actual execution time. The smaller the fault occurrence rate/the lower bound of the actual execution times, the larger amount of regular slack that can be used to recover faults and the lower risk of missing deadline for the jobs lending their re-execution slack. We investigate the effects of these two factors in our experiments and the results are shown in Table 2 and Table 3. In Table 2, each row is a set of data for a specific fault rate. The data compares of a slack reclaiming method using the regular slack only (no borrow from future) and our CBS-FT algorithm. The number of final faults are recorded when these two approaches are used. The percentages of faults recovery are also calculated. It can be seen that the CBS-FT with the borrowing of future slack always outperforms the approach using regular slack only. If we compare the number of faults recorded, an improvement of about 30% in performance can

Fault Rate	Faults in Primary Execution	Faults Recorded (Regular Slack)	Faults Recovered (Regular Slack)	Faults Recorded (CBS-FT)	Faults Recovered (CBS-FT)	Faults on Jobs w/ Re-execution Originally Reserved
0.05	4,379	1,178	73%	843	81%	0.00%
0.2	17,647	4,989	72%	3,611	80%	0.14%
0.3	26,554	7,944	70%	5,484	79%	0.5%
0.4	35,324	10,905	69%	7,685	78%	1.42%
0.5	43,832	14,074	68%	9,868	77%	2.95%

Table 2. Experimental results based on the fault occurrence rate (lower bound of actual execution times is 1.0)

Execution Times' Range	Faults in Primary Execution	Faults Recorded (Regular Slack)	Faults Recovered (Regular Slack)	Faults Recorded (CBS-FT)	Faults Recovered (CBS-FT)	Faults on Jobs w/ Re-execution Originally Reserved
0.9 - 1.0	43,298	13,935	68%	9,715	78%	2.20%
0.8 - 1.0	44,021	13,401	70%	9,366	79%	0.4%
0.7 - 1.0	43,589	11,887	73%	8,319	81%	0.06%
0.6 - 1.0	43,420	10,816	75%	7,607	82%	0.00%
0.5 - 1.0	43,489	9,922	77%	7,022	84%	0.00%
0.2 - 1.0	44,019	5,291	88%	3,828	91%	0.00%

Table 3. Experimental results based on the lower bound of actual execution times of jobs (fault-rate = 0.5)

be seen by using our CBS-FT.

With using our algorithm, it is possible that a LO-criticality task originally having its re-execution reserved violates its deadline because it lends its budget to another task to recover from fault. The last column in the table shows the percentages of the faults caused by this scenario. As we explained it earlier, since later jobs have a bigger chance of not executing the re-execution or using the regular slack generated later, a borrowing of slack from future re-execution does not significantly degrade the overall performance a lot. While the fault rate is small, nearly no lending jobs miss their deadline for recovering from their faults. Even if the fault rate is as high as 0.5, the number of faults on the jobs with the re-execution originally reserved is relatively small so that the system's performance is well maintained. Table 3's results are similar, based on the lower bounds of the actual execution times. The results are obtained by using a fixed fault occurrence rate of 0.5. A specific range that represents the lower bound of the actual execution time is used in each set. For example, 0.5 - 1.0 means that the actual execution time is between 50% and 100% of the WCET, using an even distribution. While tasks are more likely to use less time for execution, more faults can be recovered by using both approaches but our CBS-FT always does the job better. Also, while the lower bound of the actual execution time becomes lower and lower, the difference of the fault recovery rate between the two approaches becomes smaller and smaller. This is because when the jobs complete earlier, their slack can be used by other jobs earlier. While more jobs use the regular slack to recover faults, fewer jobs need to borrow slack from the future re-executions.

V. Discussion and Summary

This paper studies a novel problem of scheduling a set of mixed-criticality, real-time tasks and considering the fault-tolerance issue. Two algorithms are proposed. The static algorithm works for maximizing the reserved executions of the LO-criticality tasks including their primary and back-up executions. The on-line algorithm improves the regular approach by exploiting a technique to borrow slack from future executions. The evaluation results show that both of our algorithms significantly improve the performance. For the sake of simplicity, we assume that the system has two levels of criticality but the results can be generally expanded to have multiple levels. Also, we assume that using one re-execution sufficiently satisfies the requirement of fault-tolerance. How to handle with multiple number of faults will become the target to extend this paper. Finally, another common technique used in fault-tolerance is checkpointing. It will be an interesting problem how to use checkpointing in mixed-criticality systems for fault-tolerance.

References

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow and L. Stougie, *Scheduling Real-Time Mixed-Criticality Jobs*, IEEE Transactions on Computers, Vol. 61, No. 8, August 2012.

[2] D. de Niz, K. Lakshmanan, and R. Rajkumar, *On the Scheduling of Mixed-Criticality Real-Time Task Sets*, in RTSS, pages 291-300, 2009.

[3] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno, *Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems*, in ICDCS pages 169178, 2010.

[4] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster and L. Stougie, *The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Tas Systems*, in ECRTS, pages 147-155, 2008.

[5] S. Baruah, H. Li, and L. Stougie, *Towards the design of certifiable mixed-criticality systems*, In RTAS, pages 13-22, 2010.

[6] F. Santy. L. George, P. Thierry, and J. Goossens, *Relaxing mixedcriticality scheduling strictness for task sets scheduled with fp*. In ECRTS, pages 155-165, 2012.

[7] X. Castillo, S.R. McConnel, and D.P. Siewiorek, *Derivation and Caliberation of a Transient Error Reliability Model*, IEEE Transactions on Computers, Vol. 31, No. 7, 1982.

[8] R.K. Iyer and D.J. Rossetti, *A Measurement-Based Model for Workload Dependence of CPU Errors*, IEEE Transactions on Computers, Vol. 35, No. 6, 1986.

[9] S. Ghosh, R. Melhem and D. Mosse, *Fault-tolerant scheduling on a hard real-time multiprocessor system*, in IEEE Parallel Processing Symposium, Page 775-782, 1994.

[10] R. Al-Omari, A. K. Somani, and G. Manimaran, *Efficient Overloading Techniques for Primary-backup Scheduling in Real-Time Systems*, in Journal of Parallel Distributed Computing, Vol. 64, Issue 5, 2004.

[11] M. Caccamo, G. Buttazzo, and L. Sha, *Capacity Sharing for Overrun Control*, in RTSS, pages 295-304, 2000.

[12] C. Lin, and S. Brandt, *Improving Soft Real-Time Performance Through Better Slack Reclaiming*, in RTSS, pages 410-421, 2005.

[13] C. L. Liu and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, in Journal of the ACM, Vol 20, Issue 1, Pages 4661.

[14] R. M. Pathan, *Fault-tolerant and real-time scheduling for mixed-criticality systems*, in Real-Time Systems, Vol 50, Issue 4, pages 509-547.

[15] P. Huang, H. Yang and L. Thiele, *On the Scheduling of Fault-Tolerant Mixed-Criticality Systems*, in DAC, Pages 1-6, 2014.