The Application of the Original Priority Ceiling Protocol to Mixed Criticality Systems

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

***Abstract*—Mixed Criticality Systems (MCSs) have been the focus of considerable study over the last six years. This work has lead to the definition of a standard model that allows processors to be shared efficiently between tasks of different criticality levels. This model, however, often assumes that the tasks are independent of each other; which is an unrealistic restriction. In this paper tasks of the same criticality are allowed to shared resources that require mutually exclusive access. Such resources are usually protected by some form of priority ceiling protocol (PCP). But it is not clear that the more common form of this protocol (sometimes called the Immediate Priority Ceiling Protocol, or the Stack Resource Protocol) is appropriate for MCS. Here we evaluate the original form of the PCP and propose a new MCS-aware protocol called MCS OPCP. This protocol allows lower criticality tasks to transfer budgets when a task runs out of its own, or the resource, budget before it has completed its use of the resource. Without this provision the task would need to be suspended whilst holding a resource lock. We show that higher criticality tasks are not impacted by the protocol. An assessment of MCS OPCP in terms of its impact on schedulability analysis is provided.**

I. Introduction

Although the formal study of mixed criticality systems (MCSs) is a relatively new endeavor, starting with the paper by Vestal (of Honeywell Aerospace) in 2007 [13], a standard model has emerged (see for example [3], [2], [6], [11], [8]). This model usually defines the tasks to be independent of each other. This is clearly an unacceptable restriction for realistic industrially relevant software. It is however far from clear to what extent data should flow between criticality levels. There is a strong objection to data flowing from low to high criticality applications unless the high criticality component is able to deal with potentially unreliable data – this happens with some security protocols [4]. Even with data flowing in the other direction there remains the scheduling problem of not allowing a high criticality task to be delayed by a low criticality task that has either locked a shared resource for longer than expected or is executing at a raised priority ceiling level for too long. In this paper we look at sharing within a criticality level. We restrict ourselves to single processor systems scheduled using fixed priorities; a general sporadic task model is assumed.

For non-MCSs there are a number of priority inheritance protocols [12] that allow for the efficient sharing of resources between tasks. Here 'efficient' means delivering bounded priority inversion. Most effective amongst these protocols are those based on priority ceilings [9], [12], [7], [1]; with these protocols:

- A job is blocked at most once during its execution.
- Deadlocks are prevented.

There are two main variants of the priority ceiling protocol (PCP): the original one (described below and identified by the acronym OPCP) and the one in which the task has its priority immediately raised to the ceiling priority of the resource when it accesses the resource. This later version is the one supported by programming languages such as Ada and Java, many operating systems and many standards such as OSEK and Autosar. It is known as the Stack Resource Protocol, or the Immediate Priority Ceiling Protocol (IPCP) or Priority Ceiling Emulation. The protocol has the following additional properties:

- If a job experiences blocking it will take place prior to the job actually executing.
- Once a job starts executing, all the resources it needs will be available.
- No additional preemptions are introduced.

In the following we argue that for mixed criticality systems with inter-criticality level resource sharing, there are some advantages in re-evaluating the original form of the protocol. We will assume, initially, that there are just two criticality levels: LO and HI (with LO $<$ HI).

II. MCS Model

For dual criticality systems the standard model for MCSs has the following properties [13], [2]:

- A mixed criticality system is defined to execute in either of two modes: a HI-criticality mode or a LO-criticality mode.
- Each task is characterised by its criticality level, L (equal to either LO or HI), 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 for HI-criticality tasks, denoted by $C(HI)$ and $C(LO)$; and just a single $C(LO)$ value for LO-criticality tasks).
- The system starts in the LO-criticality mode, and remains in that mode as long as all jobs execute within their low criticality computation times ($C(LO)$).
- If any job execute for its $C(LO)$ execution time without completing then the system immediately moves to the HI-criticality mode.

- As the system moves to the HI-criticality mode all LO-criticality tasks are abandoned. No further LO-criticality jobs are executed.
- The system remains in the HI-criticality mode.
- Tasks are assumed to be independent of each other (they do not share any resource other than the processor).

In this paper we remove the assumption that tasks are independent of each other, we also allow LO-criticality tasks to run (in some form) in the HI-criticality mode as long as their impact on HI-criticality tasks is constrained so as to prevent any deadline misses in these tasks.

III. MCS Model with Resource Sharing

The standard model for MCS is easily extended to include resources. Contained within the system are m shared resources $(r^1, ..., r^m)$. Tasks may access (under mutual exclusion) these resources, but we make no assumptions as to when or how often each job accesses these shared resources during its execution. We do assume however that tasks do not self-suspend whilst accessing a resource[1]. The worst-case execution time of task τ_i when using resource r^k is denoted by c_i^k. For resources used in both critically modes there will be (as with tasks) two views as to the resource access time: $c_i^k(LO)$ and $c_i^k(HI)$ with $c_i^k(LO) \leq c_i^k(HI)$[2].

For single criticality systems using a PCP there is, for each task, τ_i a blocking term, B_i which is the maximum c_j^k over all resources (r^k) that are used by a lower priority task (τ_j) and a task of equal or higher priority than τ_i. For a dual criticality model there would therefore be two versions of this blocking term: $B_i(LO)$ and $B_i(HI)$. Note the actual task and the actual resource that leads to these maximum blocking terms may differ in the two criticality modes. Although it is possible for $B_i(HI)$ to be less than $B_i(LO)$ the analysis involves less stages if we assume $B_i(HI) \geq B_i(LO)$.

We use the Adaptive Mixed Criticality Method 1 (AMC-rtb) approach [2] to illustrate how a blocking term is introduced into the analysis model. The analysis first computes the worst-case response times for all tasks in the LO-criticality mode (denoted by $R_i(LO)$). This is accomplished by solving, via fixed point iteration, the following response-time equation for each task τ_i:

$$R_i(LO) = B_i(LO) + C_i(LO) + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO). \quad (1)$$

If any task executes for more than $C_i(LO)$ or any resources takes more than $c_i^k(LO)$ to complete then a criticality mode change occurs. Note that a resource request taking more than $c_i^k(LO)$ may not result in the blocking term $B_i(LO)$ being violated for some task, but it is the quality that can be monitored at run-time.

During and after the criticality change we are only concerned with HI-criticality tasks, so for these tasks:

$$R_i(HI) = B_i(HI) + C_i(HI) + \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \quad (2)$$

which 'caps' the interference from LO-criticality tasks. Note the summations are now over $\mathbf{hpH}(i)$ and $\mathbf{hpL}(i)$, the sets of higher and lower criticality (higher priority) tasks.

Although many papers assume that LO-criticality tasks are abandoned once the system moves to the HI-criticality, this is not a realistic strategy. Some form of recovery scheme is needed for these LO-criticality tasks. One simple approach would be to lower their priorities to, in effect, a background level. But this would not impact on a task executing with an inherited higher priority from executing with a resource. Some other form of robust protocol is required. For example:

1) abandon the resource;
2) suspend the LO-criticality task.

With the first approach, when the budget on the resource is exhausted or the $C(LO)$ budget on the entire job is exhausted (while it holds the resource) the resource is 'pulled' from the job. The resource is returned to its previous state and the job must re-access it. Other jobs may get it first of course, but the resource is not locked unnecessarily. The disadvantage of this approach is that the resource update must be implemented as an atomic action. Also, an error in which $c_j^k(LO)$ is actually insufficient will mean that the task τ_j will loop indefinitely accessing the resource, having its code aborted and then retrying (and re-failing).

The second scheme is equivalent to the job itself running out of budget. It will not execute again until the task's next release. When next released it will be allowed to continue to execute within the resource (for up to $c_j^k(LO)$ again). This has the advantage that any lower priority HI-criticality task will have a blocking term no higher than $B_i(LO)$. So the constraints of the LO-criticality execution mode continue to be satisfied; however, this behaviour is extremely detrimental to the LO-criticality tasks as a lock on a shared resource is being held by a task that is in effect suspended. To ease this situation one can envisage a protocol in which capacity (budget) is transferred from a stalled LO-criticality job to the out-of-budget LO-criticality task that has the lock. Such a protocol is relatively easy to support with OPCP; and we define one in Section V-A.

IV. The Original Priority Ceiling Protocol, OPCP

The original form of the priority ceiling protocol protocol builds on the simpler priority inheritance protocol which just says that if τ_i has locked[3] a resource which is then required

[1] A task running out of budget and being de-scheduled whilst accessing a resource is not considered to be self-suspending. This is however a situation that is far from ideal and techniques are developed in this paper to prevent this happening.

[2] For simple resources it is likely that, in practice, $c_i^k(LO) = c_i^k(HI)$, but the general model would allow $c_i^k(LO)$ to be assigned a lower value.

[3] We use the term *lock* although a physical OS lock is not strictly necessary with a priority ceiling protocol.

by higher priority task τ_j then τ_i will inherit the priority of τ_j for the duration of the time it holds the lock. In addition to this rule, for OPCP, a task is only allowed to lock a resource if its active priority is higher than the current system ceiling. And this value is the maximum ceiling value of all currently locked resources (excluding any the task has locked itself).

Consider the example represented in Table I. There are four LO-criticality tasks (L1, ..., L4) and two high criticality tasks (H1 and H2). There are also three resources (r^1, r^2 and r^3). The table is ordered by the task's priorities (L1 has the highest priority) and gives the usage relation between the tasks and resources. The execution times for the resources are: $c_i^1 = 5\ \forall i$, $c_i^2 = 7\ \forall i$ and $c_i^3 = 10\ \forall i$. The priority ceilings of the

Task	Priority	User r^1	Uses r^2	Uses r^3
L1	1	•		
H1	2		•	
L2	3	•		•
H2	4		•	
L3	5	•		•
L4	6			•

TABLE I
EXAMPLE SYSTEM OF TASKS AND RESOURCES

three resources are: $P(r^1) = 1$, $P(r^2) = 2$ and $P(r^3) = 3$. The blocking terms introduced by IPCP or OPCP, for use in the scheduling analysis, are: $B_{L1} = 5$, $B_{H1} = 7$, $B_{L2} = 10$, $B_{H2} = 10$, $B_{L3} = 10$ and $B_{L4} = 0$. These are easily computed; for example, H2 can be blocked by L3's use of r^1 or L4's use of r^3. Any priority ceiling protocol ensures that only one block per job is actually experience, so $B_{H2} = \max(5, 10) = 10$.

An illustrative execution of this example is when L3 executes first and locks r^1, H1 is then released and preempts L3 as it has a higher priority. H1 then attempts to access r^2, OPCP prevents this access (even though the resource is free) as H1's priority (2) is not greater then the current system ceiling (which has the value 1 as r^1 with priority ceiling 1 is locked by L3). L3 will now execute with priority 2 until it unlocks r^1; it priority then returns to 5, H1 will preempt and be allowed to lock r^2.

V. MCS PRIORITY CEILING PROTOCOLS

In this section we shall explore two properties of OPCP that are advantageous for MCS. First a partitioning of the resources using criticality, and secondly capacity sharing between LO-criticality tasks; these both contribute to the definition of an MCS OPCP.

A. An MCS OPCP

For a dual criticality system let the collection of m resources be split into two sets, h and l. With $h = h^1, ..., h^m 1$ and $l = l^1, ..., l^m 2$, and $m1 + m2 = m$. The first set is used only by high criticality tasks, the second by only low criticality tasks. The resource execution times for HI-criticality task τ_i when accessing resource k in h is denoted, as before, by $c_i^k(LO)$ and $c_i^k(HI)$. For a resource s in l there is just one execution time per task – $c_j^s(LO)$, where τ_j is the LO-criticality task accessing the resource.

The proposed MCS OPCP protocol has the following characteristics:

- All resources are assigned a ceiling priority.
- A task can only access a resource if the task and resource are of the same criticality.
- At run-time the system maintains two system ceiling values, one per criticality level; they each represent the current highest ceiling priority of a locked resource.
- At run-time a task can only lock a resource if its priority is strictly greater than the system ceiling for its criticality level (again excluding resources that the task has locked itself).
- A task will inherit the priority of any task of higher priority that it is blocking.

The consequences of these rules is that a task can only be directly blocked by a lower priority task of the same criticality. So in the earlier illustration from the example (in Table I), H1 would be able to lock r^2 even while L3 has the lock on r^1. However, the worst-case blocking value (in the scheduling analysis) does increase as it must now involve two terms, one derived from set h the other from set l. Equation (1) becomes:

$$R_i(LO) = Bh_i(LO) + Bl_i(LO) + C_i(LO) + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \quad (3)$$

where, $Bh_i(LO)$ is the maximum LO-criticality resource execution time for a resource used by HI-criticality tasks (one with priority less than τ_i, one with priority equal or higher); $Bl_i(LO)$ is defined similarly. Equation (2) is updated in the same way:

$$R_i(HI) = Bh_i(HI) + Bl_i(LO) + C_i(HI) + \sum_{\tau_j \in \mathbf{hpH}(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in \mathbf{hpL}(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \quad (4)$$

The example (in Table I) illustrates some interesting properties of MCS OPCP. First the resources are appropriately partitioned with $r^1, r^3 \in l$ and $r^2 \in h$. The priority ceiling values remain the same. If the execution times allocated earlier are interpreted as implying LO-criticality values then $c_i^1(LO) = 5;\ i = L1, L2\ or\ L3$, $c_i^2(LO) = 7;\ i = H1\ or\ H2$ and $c_i^3(LO) = 10;\ i = L2, L3\ or\ L4$. Finally, the single required HI-criticality value is given by $c_i^2(HI) = 12;\ i = H1\ or\ H2$.

The blocking terms for equation (3) (i.e. the LO-criticality mode analysis) are as given in Table II. The final column gives

the blocking value if a single OPCS protocol is used, it is the maximum of the values in columns 2 and 3. Clearly the blocking terms have increased for some tasks, but not all. For the HI-criticality mode analysis (equation (4)), $Bh_{H1}(HI) = 12$ and $Bh_{H2}(HI) = 0$, and $Bl_{H1}(LO) = 5$ and $Bl_{H2}(LO) = 10$.

Task	$Bl(LO)$	$Bh(LO)$	$Bl(LO) + Bh(LO)$	OPCP B
L1	0	5	5	5
H1	5	7	12	7
L2	10	7	17	10
H2	10	0	10	10
L3	10	0	10	10
L4	0	0	0	0

TABLE II
BLOCKING TIMES

The use of these (increased) blocking times represents the worst-case behaviour. But there are other considerations in terms of an overall evaluation of MCS OPCP. Consider the impact on H2 in the LO-criticality mode. Under OPCP its access to r^2 could be blocked if either r^1 and r^2 were locked. But under MCS OPCP it does not need to check, it is the lowest priority of the users of r^2 and hence if it is executing r^2 must be free. Under MCS OPCP a task can only directly block tasks of the same criticality and hence priority inheritance only occurs between tasks of the same criticality. This simplifies implementation (and therefore certification). Note with IPCP this separation is not possible, L4 executing with r^3 will execute with priority 3 and hence directly impact H2. Of course the worst-case behaviour via indirect blocking is the same (e.g. H2 get interference from L2, L2 can be blocked by L4, and hence L4 could have its priority raised to 3).

Returning to the illustrative execution involving H1; when it attempts to lock r^2 it can now only be prevented from making progress if its priority is not strictly higher than the system ceiling for h. This can only be the case if H2 has locked the resource. The state of r^1 and r^3 are immaterial to the behaviour of H1. This would not be the case with OPCP or IPCP. Again this helps certification as it increases the separation between criticality levels.

The property of MCS OPCP that a task can only be directly blocked by tasks of its own criticality level, and that within that level an ordinary priority ceiling protocol is designated implies that deadlock free execution of task sets that adhere to MCS OPCP is guaranteed.

In terms of average-case behaviour, OPCP has always had an advantage over IPCP. With IPCP a priority change (to ceiling value and back) occurs for all resources usages. With OPCP it only occurs if there is an actual priority inversion. With MCS OPCP the chances of both HI and LO priority inversion occurring is even more unlikely. It might be possible to computed that the probability of a lower priority task being preempted while it held the lock on a resource, and that a task that needs the resource subsequently executes, is less than, say, 10^{-3}. With MCS OPCP the probability of this happening to both a resource from h and a resource from l will be close to 10^{-6} (as the events can reasonably be defined to be independent). So, from Table II, the blocking term for, for example, L2 would be 17 with probability 10^{-6} or 10 with probability 10^{-3}. Probabilistic scheduling analysis [5], [10] could easily factor this into its analysis framework. Note for an increased number of criticality levels this probability decreases significantly. For five levels the probability of a task suffering blocking from all the levels could be argued to be less than 10^{-15}!

B. Budget Inherence with MCS OPCP

If a LO-criticality task is suspended due to an exhausted resource budget then it will have no impact on other tasks until a further LO-criticality task attempts to use the resource. With OPCP there is a direct relationship between the lock holding tasks and the blocked task. And with MCS OPCP the blocked task must be of the same criticality. Hence is it relatively straightforward to define a protocol for LO-criticality task's use of resources:

- Only allow a LO-criticality task (τ_j) to attempt to lock a resource (r^k) if its remaining budget (from $C_j(LO)$) is greater than $c_j^k(LO)$.
- If the resource budget ($c_j^k(LO)$) is exhausted τ_j is suspended.
- If another task (τ_i) attempts to access r^k then budget capacity is transferred from τ_i to τ_j.

Enough budget is transferred to allow τ_j to complete its operation on the resource. Note τ_j and τ_i are of the same LO-criticality.

This protocol has some consequences for both HI and LO criticality tasks:

- LO-criticality task τ_i has had budget 'misappropriated' and this may impact on its ability to meet its deadlines. But this is clearly a less serious impact than suspending τ_i until τ_j at some future point releases the lock on r^k.
- If the priority of τ_i is greater than that of τ_j then budget is passing from high priority to lower priority; this cannot have a negative impact on any other task in the system (HI or LO).
- If the priority of τ_i is less than that of τ_j then as τ_i is executing it must have the highest priority of any runnable job. Hence there are no runnable HI-criticality tasks with priority greater than τ_i or τ_j. The movement of capacity from τ_i to τ_j therefore cannot impact on any runnable HI-criticality task.

None of these consequences seem to be problematic.

The result of using budget inheritance is that LO-criticality tasks execute more robustly, whilst HI-criticality tasks are unaffected. At run-time no tasks have their priorities changed, and hence static unchangeable priorities can be used. A system that allows LO-criticality tasks to change their priorities is open to a security breach in which they are changed to unacceptable levels. Budgets are only moved between LO-criticality tasks and this happens using a protocol that cannot impact on HI-criticality tasks. The only real down-side to using

MCS OPCP is the extra blocking terms in the scheduling analysis.

VI. More than two Criticality Levels

Many papers on mixed criticality constrain themselves to only address dual criticality system. This restriction helps to clarify descriptions, and has been utilised here. However, it is important that protocols do generalise to a realistic number of criticality levels. Perhaps up to five levels may be needed (see, for example, the IEC 61508, DO-178B, DO-254 and ISO 26262 standards).

MCS OPCP naturally generalises to more than two levels. A set of resources per level is defined and a current system ceiling level per set is used at run-time. Each set can give rise to a blocking term in the schedulability analysis. But as indicated earlier, the likelihood of any job actually suffering this worst-case scenario decreases with the number of levels and opens the way to a probabilistic interpretation for the blocking term.

Budget inheritance is only between tasks of the same criticality and hence this also scales to multiple levels of criticality. There is an inevitable increase in run-time complexity, but for up to five levels this should not be a significant issue. Note where there are five levels of criticality the four lowest need to be monitored and could benefit from budget inheritance. The highest criticality level is always dealt with differently; there is nothing to be gained from constraining its tasks. If they overrun the best response is to allow them to continue and hopefully recover without further intervention other than removing the impact of all lower criticality tasks.

With a number of distinct levels it is actually unlikely that any resource will have, say, five different estimates of the worst-case execution time for its operators/methods. These operators are ideally very simple and short compared with the execution behaviours of the tasks in the system. Hence it is quite likely that even with five criticality levels there may only be, say, two levels for the resources.

VII. Conclusions

This paper has focused on single processor systems on which, for single criticality systems, the Immediate Priority Ceiling Protocol (IPCP) is almost universally accepted as the protocol to use for protected access to shared resources (e.g. critical sections that must be accessed under mutual exclusion). For mixed criticality systems it is not clear that IPCP is the right protocol. However it is clear that resource sharing within critically levels is a necessary part of any usable task model.

In this paper we argue that the issue of priority ceiling protocols for MCS requires further study. We attempt to contribute to this study by considering the properties of the original version of PCP. We define a new protocol for mixed criticality systems based on this original version which we refer to as MCS OPCP. This protocol separates all resources into sets, one per criticality level. An OPCP is then defined per set. With MCS OPCP a task can only be directly blocked if a resource is locked by a lower priority task of the same criticality level.

An analysis of MCS OPCP shows that worst-case blocking is increased (over global IPCP or OPCP), but there are a number of advantages to its use. It aids separation, and its average run-time overhead is likely to be significantly less than a global IPCP scheme. It could also form a component of a probabilistic view of schedulability analysis for mixed criticality systems. Finally, it retains the property of deadlock-free behaviour.

References

[1] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[2] S. 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.

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

[4] K. Biba. Integrity considerations for secure computer systems. Mtr-3153, Mitre Corporation, 1977.

[5] J. L. Díaz, D. F. García, K. Kim, C.-G. Lee, L. L. Bello, J. M. López, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *RTSS*, pages 289–300, 2002.

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

[7] J. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *Proc. of the 2nd International Workshop on Real Time Ada Issues, ACM Ada Letters*, 8(7):22–31, 1988.

[8] 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.

[9] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *CACM*, 23(2):105–117, 1980.

[10] D. Maxim, O. Buffet, L. Santinelli, L. Cucu-Grosjean, and R. I. Davis. Optimal priority assignment algorithms for probabilistic real-time systems. In *RTNS*, pages 129–138, 2011.

[11] F. Santy, P. Richard, M. Richard, 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.

[12] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[13] 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.