Washington University in St. Louis
Washington University Open Scholarship

All Theses and Dissertations (ETDs)

6-15-2012

MCFlow: Middleware for Mixed-Criticality Distributed Real-Time Systems

Huang-Ming Huang
Washington University in St. Louis

Follow this and additional works at: http://openscholarship.wustl.edu/etd

Recommended Citation
Huang, Huang-Ming, "MCFlow: Middleware for Mixed-Criticality Distributed Real-Time Systems" (2012). *All Theses and Dissertations (ETDs)*. 961.
http://openscholarship.wustl.edu/etd/961

This Dissertation is brought to you for free and open access by Washington University Open Scholarship. It has been accepted for inclusion in All Theses and Dissertations (ETDs) by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Christopher D. Gill, Chair
Roger Chamberlain
Chenyang Lu
Paul Min
David Peters
Bill Smart

MCFlow: Middleware for Mixed-Criticality Distributed Real-Time Systems

by

Huang-Ming Huang

A dissertation presented to the Graduate School of Arts & Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2012
Saint Louis, Missouri

copyright by

Huang-Ming Huang

2012

ABSTRACT OF THE DISSERTATION

MCFlow: Middleware for Mixed-Criticality Distributed Real-Time Systems

by

Huang-Ming Huang

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2012

Research Advisor: Professor Christopher D. Gill

Traditional fixed-priority scheduling analysis for periodic/sporadic task sets is based on the assumption that all tasks are equally critical to the correct operation of the system. Therefore, every task has to be schedulable under the scheduling policy, and estimates of tasks' worst case execution times must be conservative in case a task runs longer than is usual. To address the significant under-utilization of a system's resources under normal operating conditions that can arise from these assumptions, several *mixed-criticality scheduling* approaches have been proposed. However, to date there has been no quantitative comparison of system schedulability or run-time overhead for the different approaches.

In this dissertation, we present what is to our knowledge the first side-by-side implementation and evaluation of those approaches, for periodic and sporadic mixed-criticality tasks on uniprocessor or distributed systems, under a mixed-criticality scheduling model that is common to all these approaches. To make a fair evaluation of

mixed-criticality scheduling, we also address some previously open issues and propose modifications to improve schedulability and correctness of particular approaches.

To facilitate the development and evaluation of mixed-criticality applications, we have designed and developed a distributed real-time middleware, called MCFlow, for mixed-criticality end-to-end tasks running on multi-core platforms. The research presented in this dissertation provides the following contributions to the state of the art in real-time middleware: (1) an efficient component model through which dependent subtask graphs can be configured flexibly for execution within a single core, across cores of a common host, or spanning multiple hosts; (2) support for optimizations to inter-component communication to reduce data copying without sacrificing the ability to execute subtasks in parallel; (3) a strict separation of timing and functional concerns so that they can be configured independently; (4) an event dispatching architecture that uses lock free algorithms where possible to reduce memory contention, CPU context switching, and priority inversion; and (5) empirical evaluations of MCFlow itself and of different mixed criticality scheduling approaches both with a single host and end-to-end across multiple hosts. The results of our evaluation show that in terms of basic distributed real-time behavior MCFlow performs comparably to the state of the art TAO real-time object request broker when only one core is used and outperforms TAO when multiple cores are involved. We also identify and categorize different use cases under which different mixed criticality scheduling approaches are preferable.

Acknowledgments

A number of people have helped me over the past few years of my graduate student life in Washington University in St. Louis. Without their help, I would have never fulfilled my ambition of earning a doctorate degree.

First and foremost, I am truly indebted and thankful to my parents. Without their love and support, I couldn't have came to the United States pursuing the degree. I would also like to thank to my brother Huang-Yuan and sister-in-law Yi-Chen who have taken over the responsibility to look after the family during my absence.

I am sincerely and heartily grateful to my advisor, Christopher D. Gill, for the support and guidance he showed me throughout the entire period of my graduate life. Even after the dismal result for my first attempt in the dissertation topic, he still encourage me not to drop out and provide me continuing financial support. The other person who I owe a debt of gratitude in my academic studying is Dr. Chenyang Lu. He helped me to choose MCFlow as the foundation of my final dissertation topic and co-authored most of my publications.

I am grateful to my dissertation defense committee, Dr. Roger D. Chamberlain, Dr. Christopher D. Gill, Dr. Chenyang Lu, Dr. Paul Min, Dr. David Peters, and Dr. Bill Smart, for their time and efforts spent reviewing, commenting on, and making suggestions for improvement to this dissertation. I am also obliged to many of the people who collaborate with me for my publications, they are Shirley Dyke, Venkita Subramonian, Xiaorui Wang, Xiuyu Gao, Terry Tidwell.

I am indebted to my friends for their support and encouragement. They are Yi-Hsin Liu, Li-Wei Chang, Waiman Suen, Yiting Zheng, Chien-Yuan Hong, Ching-Yi Wu, Annabelle Pan, Tsai-Chen Lin, Boyu Tsai, Liang-Jui Sheng, Chiming Ho, Ty Thai, Tianyu Zhao, Nanbor Wang, Yuanfang Zhang, Sisu Xi, Zhixiang Xu. I would also like to thank the staff (Jean Grothe, Myrna Harbison, Peggy Fuller, and Sharon Matlock, Kelli Eckman, Madeline Hawkins, Jayme Moehle) of the Department of Computer Science for their help and generosity over the years I have been at Washington University.

Support for this research was provided in part by NSF grants CNS-0834755 (CSR-DMSS), CNS-0821713 (MRI), and CCF-0448562 (CAREER).

Huang-Ming Huang

Washington University in Saint Louis
August 2012

Dedicated to my parents.

Contents

Abstract . . . . . **ii**

Acknowledgments . . . . . **iv**

List of Tables . . . . . **x**

List of Figures . . . . . **xi**

1 Introduction **1**
1.1 Mixed-Criticality Scheduling 1
1.2 MCFlow 3
1.3 System Model 4
1.3.1 Mixed-Criticality Uniprocessor Systems 5
1.3.2 Mixed-Criticality End-to-End Systems 7
1.4 Basic Middleware Requirements 8
1.5 Contributions of This Work 10
1.6 Dissertation Organization 11

2 Basic MCFlow Design and Performance Evaluation **13**
2.1 Encapsulation via Components 13
2.1.1 Component Interfaces 15
2.1.2 Interface Type Safety 17
2.1.3 Component Communication 18
2.1.4 Deployment Plan and Code Generation 19
2.2 Middleware Architecture 19
2.2.1 Task Management Subsystem 21
2.2.2 Dispatching Subsystem 22
2.2.3 Subtask Release Mechanism 26
2.3 Performance Evaluation 27
2.3.1 Latency Comparison 29
2.3.2 Speedup Comparison 32
2.3.3 Real-Time Performance 33

3 Mixed-Criticality Scheduling **35**
3.1 Classification of Mixed-Criticality Scheduling Approaches 35
3.1.1 Run-Time Enforcement Mechanisms 36
3.1.2 Priority Assignment 36

3.2 Zero-Slack Scheduling and our Improvements . . . 40
3.2.1 Execution After a Missed Deadline . . . 42
3.2.2 Design Challenges . . . 43
3.2.3 Solution Approach . . . 43
3.2.4 Unaccounted Interference . . . 44
3.2.5 Worst Case Alignment . . . 46
3.2.6 Refining the ZSI Calculation . . . 48
3.2.7 Normal Mode Time Demand Function . . . 49
3.2.8 A Four-Task Example . . . 50
3.2.9 Critical Mode Time Demand Function . . . 53
3.2.10 Available Slack Function . . . 55
3.2.11 Improved ZSI Calculation Algorithm . . . 57
3.3 Fixed Task Priority AMC Scheduling (FTP-AMC) . . . 58
3.3.1 Basic Analysis . . . 58
3.3.2 Improved Analysis . . . 59
3.4 Fixed Job Priority AMC Scheduling (FJP-AMC) . . . 62
3.5 Mixed-Criticality End-to-End Task Sets . . . 66
3.5.1 Priority Assignment for Mixed-Criticality End-to-End Tasks . 68

4 Mixed Criticality Evaluations . . . 73
4.1 Schedulability Evaluation for Mixed-Criticality Tasks on Uniprocessor Systems . . . 73
4.1.1 Task set generation parameters . . . 73
4.1.2 Scheduling approaches and analysis investigated . . . 74
4.1.3 Simulation . . . 75
4.2 Schedulability Evaluation for Mixed-Criticality End-to-End Tasks . . 83
4.2.1 Workload generation . . . 83
4.2.2 Scheduling approaches investigated . . . 84
4.2.3 Simulation . . . 84
4.3 Mixed-Criticality Runtime Enforcement Mechanism Implementation and Evaluation . . . 88
4.3.1 Zero-Slack Scheduling Implementation . . . 88
4.3.2 Period Transformation Implementation . . . 89
4.3.3 AMC Implementation . . . 90
4.3.4 Empirical Evaluation . . . 91

5 Related Work . . . 100
5.1 Related Work on Middleware . . . 100
5.2 Related Work on Mixed-Criticality Scheduling . . . 102

6 Conclusions . . . 106

Appendices . . . 109

Appendix A Original Zero Slack Instant Calculation Algorithms . 110

References 113

Vita 122

List of Tables

1.1 A two task example from [63] 2

2.1 Valid preparers for MCFlow worker components 19
2.2 (CPU, workload in μs) for each task's subtasks 34
2.3 Tasks deadline miss ratios 34
2.4 Average response times in μs 34

3.1 Summary of approaches for periodic or sporadic mixed-criticality tasks 39
3.2 A two task ZSRM example 42
3.3 Example task set for worst case phasing condition 45
3.4 Example 4-task set for ZSI calculation 50
3.5 An example task set where the calculation of Z_2 is based on Equation 3.8 53

4.1 Number of task sets that are uniquely schedulable by each scheduling approach with proportional deadline assignment (DP=1.6) 87
4.2 Time stamp log (in μs) for the example in Table 1.1 where only the first jobs of the tasks are overloaded 93
4.3 A 4 tasks example (in ms) 94
4.4 The average and maximum cost in cycles of manager thread invocation 98

5.1 The DO-178B standard 102

List of Figures

1.1 Two tasks example under various scenarios (from [63]) 2
1.2 Task model in MCFlow 5

2.1 MCFlow component model 14
2.2 MCFlow host architecture (the squiggly arrows represent threads) . . 20
2.3 Example MCFlow initialization flow 21
2.4 Real-time communication via priority lanes 23
2.5 TAO Real-Time Event Service ITC mechanism 23
2.6 MC-ORB ITC mechanism 23
2.7 MCFlow ITC mechanism 25
2.8 Data merge in MCFlow 25
2.9 Single task experiment setup 28
2.10 Average server response time latency per invocation 30
2.11 Average parallel subtask release to termination latency per invocation 30
2.12 Round trip latency comparison 31
2.13 Latency comparison server result 33

3.1 Two tasks example under ZSRM scheduling policy 40
3.2 Zero-slack rate-monotonic scheduling of the task set in Table 3.2 . . . 42
3.3 Illustration of zero-slack scheduling with demotion-on-deadline rule for the task set in Table 3.2 44
3.4 Zero-slack scheduling of the task set in Table 3.3 45
3.5 Illustration of Theorem 1 47
3.6 Illustration of $r_j^{i,n}$ and $\delta_i^n(\zeta_m, \tau_j, t)$ where both τ_j and τ_k are in $H_i^{\zeta<\zeta_i}$ and $\pi_k > \pi_j$ 49
3.7 Illustration of worst phasing schedule for the tasks set in Table 3.4 . . 52
3.8 A schedule for the task set in Table 3.5 which shows ZSI calculation based on Equation 3.8 can cause τ_2 to miss deadline 54
3.9 The illustration of ϕ_j^c using the example task set in Table 3.5 55
3.10 The relationship between the slack function $S_4^n(t)$ and $t - \Delta_4^n(\zeta_4, \Gamma_4^n, t)$ for τ_4 in Table 3.4 56

4.1 Schedulability evaluation for fixed task priority SMC/AMC analyses with 20 tasks (NT=20, CF=1.5, NC=2) 76
4.2 Schedulability evaluation for fixed task priority SMC/AMC analyses with varying numbers of tasks (CF=1.5, NC=2) 77

4.3 Schedulability evaluation for fixed task priority SMC/AMC analyses with varying CF (NT=20, NC=2) 78
4.4 Schedulability evaluation for fixed job priority analyses (NT=4, NC=4, CF=1.5) 79
4.5 Schedulability evaluation for MC scheduling approaches with 20 tasks (NT=20, CF=1.5, NC=4) 79
4.6 Schedulability evaluation for MC scheduling approaches with varying number of tasks (CF=1.5, NC=4) 80
4.7 Schedulability evaluation for MC scheduling approaches with varying CF (NT=20, NC=4) 81
4.8 Schedulability evaluation for MC scheduling approaches with varying number of criticalities (NT=20, CF=1.5) 81
4.9 Comparison between different priority assignments using proportional deadline 85
4.10 The improvement of slack reallocation 86
4.11 Comparison between different priority assignments with and without slack reallocation using proportional deadline 87
4.12 Comparison between PD and NPD for FJP-HGL 87
4.13 Illustration of the time stamp log in Table 4.2 93
4.14 Busy period time comparison between different scheduling mechanisms for the task set in Table 4.3 95
4.15 Preemption overhead for Linux RT kernel 96
4.16 The cost of priority adjustment 97

Chapter 1

Introduction

1.1 Mixed-Criticality Scheduling

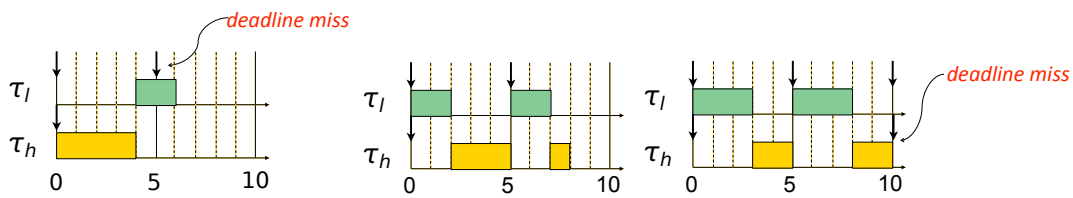
Traditional fixed-priority scheduling analysis for periodic task sets assumes that all tasks are equally critical to a system's correct operation; thus, every task has to be schedulable under a chosen scheduling policy. To meet this assumption, the estimation of worst case execution time for tasks has to be conservative in order to accommodate the special case when a task runs longer than average. Such a conservative approach can in turn lead to under-utilization of system resources (e.g., CPU cycles) under normal operating conditions.

Mixed criticality models. To address this issue, Vestal et al. [95, 21] and de Niz et al. [39, 63] have developed alternative *mixed-criticality* models for systems in which tasks are not equally critical. In the first model [95, 21], each task τ_i may have a *set* of alternative execution times $C_i(\ell)$, each having a different level of confidence ℓ. A task τ_i is also assigned a criticality level ζ_i, which corresponds to the required level of confidence for the task and is used in schedulability analysis.

The second model [39, 63] (de Niz Model) is a special case of the first one, where each task can specify only two execution times: a *normal worst case execution time* C_i^n and an *overload budget* C_i^o. Assuming all confidence and criticality levels are positive integers, with larger values indicating higher confidence and higher criticality, the

Table 1.1: A two task example from [63]

Task	C^n	C^o	Period	Criticality
τ_h	4	6	10	High
τ_l	2	3	5	Low



(a) Criticality Monotonic (b) Rate Monotonic (c) Rate Monotonic under overload

Figure 1.1: Two tasks example under various scenarios (from [63])

execution times of a task τ_i are

$$C_i(\ell) = \begin{cases} C_i^n & \text{if } \zeta_i > \ell, \\ C_i^o & \text{otherwise.} \end{cases} \tag{1.1}$$

Mixed criticality enforcement. To ensure that no lower-criticality task prevents a higher-criticality task from meeting its deadline, a scheduler could use the criticality of a task directly as its scheduling priority. However, this would unnecessarily penalize lower criticality tasks when the system is not overloaded.

Lakshmanan [63] gave an example with 2 tasks to illustrate this issue, which is shown in Table 1.1 where the deadlines and periods of tasks are equal and the priorities are assigned in rate-monotonic order (with higher priority assigned to higher frequency tasks). Each task in the mixed-criticality model has two execution time specifications: worst case execution time under non-overload conditions (C^n) and an overload execution budget (C^o) which gives an upper bound for how much a task can be overloaded.

Figure 1.1a shows that lower-criticality task τ_l misses its deadline under non-overloaded conditions when the task priorities are assigned according to their criticality levels. However, if the rate-monotonic policy is used, both τ_l and τ_h can meet their deadlines as shown in Figure 1.1b. On the other hand, when τ_l is overloaded, the purely

rate-monotonic scheduling policy can lead to *criticality inversion* by allowing τ_l to meet its deadline while the higher-criticality task τ_h misses its deadline (shown in Figure 1.1c).

To improve the schedulability of lower-criticality tasks while preserving the mixed-criticality scheduling guarantee, numerous theoretical approaches [95, 21, 39, 66, 19, 50] have been proposed. Despite their potential to improve schedulability of mixed-criticality task sets, however, to date there has been no practical comparison of the system schedulability or run-time implementation overhead implications for these different approaches. In addition, some of the proposed mixed-criticality scheduling approaches contain unresolved issues which can affect the correctness of system schedulability analysis. In Chapter 3, we discuss the details of existing mixed-criticality approaches, their issues, and how we can improve them.

1.2 MCFlow

Real-time middleware, which is software that mediates the interactions between real-time applications and operating systems that support them, offers a potentially suitable setting in which to implement and evaluate diverse mixed-criticality real-time scheduling approaches. However, as computers have evolved from uni-processor to multi-core platforms, traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution. For example, traditional real-time middleware requires explicit concurrency management and synchronization control which may scale poorly as the number of cores in a host increases.

The emergence of multi-core platforms also makes new applications, such as high fidelity real-time testing of civil structures [87], possible through parallel execution of subtasks of computations. As was noted in [92, 54], for example, the scale of civil structures such as buildings and bridges often makes it infeasible to test them fully through empirical techniques alone so that rate dependent physical elements often must be integrated with complex numerical computations, for which parallel execution of their subtasks is crucial to meet timing (e.g., to preserve realistic physical dynamics) and scalability constraints.

A new generation of real-time middleware is thus needed that can support and optimize parallel execution of subtasks from multiple (potentially mixed-criticality) real-time tasks, in distributed systems spanning multiple hosts and multiple processor cores within each host. The contributions of this dissertation are implemented within MCFlow, a middleware designed specifically to: (1) execute and coordinate mixed-criticality directed acyclic graphs of soft real-time subtasks efficiently and in parallel across multi-core processors; (2) provide a simple and intuitive programming model within which an application's subtasks may be implemented and inter-connected in a type-safe manner according to their data dependences and other precedence constraints; (3) facilitate system integration and deployment through automatic code generation from a deployment plan specification; and (4) provide mechanisms needed to implement and evaluate a wide range of mixed-criticality scheduling policies.

1.3 System Model

We consider a distributed real-time environment consisting of a collection of hosts. Each host may have one or more processor cores, and the hosts are connected by a common network. We consider distributed real-time applications made up of tasks, each of which can be represented as a directed acyclic graph (DAG) in which the vertexes of each DAG represent subtasks and the edges represent precedence constraints among the subtasks. We say that the subtasks from the same task are *dependent* due to their precedence constraints (e.g., a subtask may depend on data output by other subtasks for its inputs). In our system model, the subtasks may be allocated freely to different cores on different hosts. Thus, dependences between subtasks may need to be enforced within a single processor core, between cores on the same host, or between hosts. We use the term *team* to denote a set of dependent subtasks allocated on the same host, which belong to the same end-to-end task.

As Figure 1.2 illustrates, an *initial subtask* does not depend on any other subtask, an *intermediate subtask* has at least one other subtask on which it depends and at least one other subtask that depends on it, and a *terminal subtask* has no other subtasks that depend on it. Our system model allows tasks to have multiple initial, intermediate, and terminal subtasks, so that arbitrary DAGs are supported.

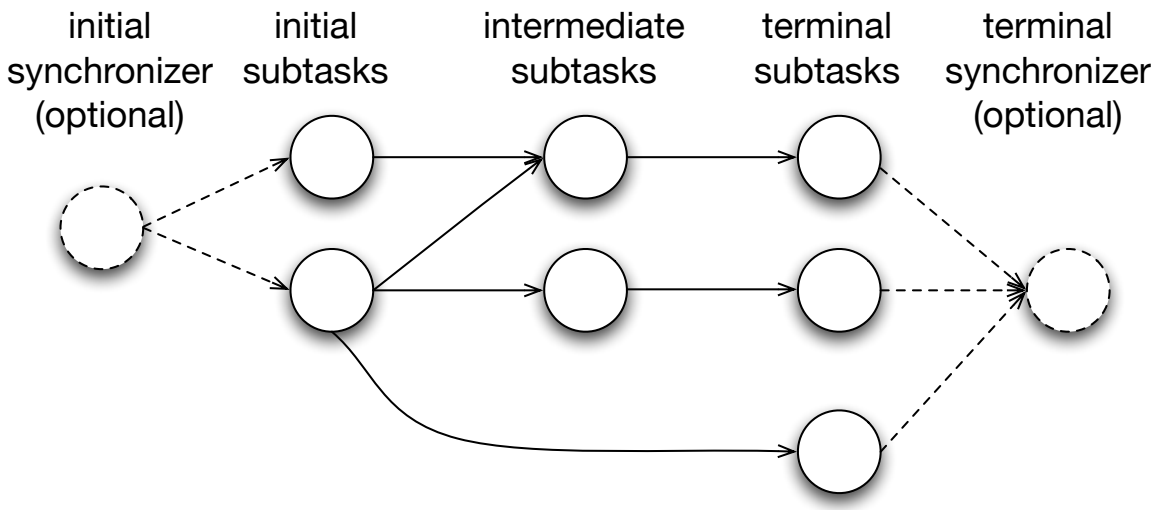


Figure 1.2: Task model in MCFlow

However, to meet application synchronization requirements or even to simplify analysis, for many systems it may be useful to consider the common special cases where a task has a single initial subtask, a single terminal subtask, or both. The component encapsulation model and middleware architecture discussed in Sections 2.1 and 2.2 make it straightforward to adapt any general task DAG to have a single initial subtask and a single terminal subtask by connecting an optional *initial synchronizer* subtask to the task's initial subtasks, and connecting the task's terminal subtasks to an optional *terminal synchronizer* subtask, as shown in Figure 1.2. For simplicity, we henceforth assume that each task has a single initial subtask and a single terminal subtask, unless stated otherwise.

1.3.1 Mixed-Criticality Uniprocessor Systems

In this subsection, we present the mixed-criticality task model for a single processor in a single host as was formalized by [50] and explain related terms and concepts that are used in this dissertation.

We consider a system which consists of a set of mixed-criticality (MC) recurrent tasks scheduled preemptively on a single processor. Let L be a positive integer indicating

the number of confidence levels in the system. A mixed-criticality recurrent task is defined as a 4-tuple: $\tau_i \equiv \langle T_i, D_i, C_i, \zeta_i \rangle$, where

- $T_i \in \mathbb{R}_+$ is the period of τ_i.
- $D_i \in \mathbb{R}_+$ is the relative deadline of τ_i; we assume $D_i \leq T_i$ unless otherwise specified.
- $\zeta_i \in \{0, 1, 2, \cdots, L-1\}$ is the criticality level of τ_i with larger value indicating higher criticality.
- $C_i \in \mathbb{R}_+^L$ is a vector of worst case execution time (WCET) estimations of τ_i in each confidence level. An estimation of a higher confidence level represents a more pessimistic estimation. That is, the constraint $C_i(\ell_1) \leq C_i(\ell_2)$ holds if $\ell_1 < \ell_2$.

Each mixed-criticality recurrent task τ_i consists of an infinite number of jobs, where each MC job $\tau_{i,j}$ can be characterized by a 4-tuple: $\tau_{i,j} \equiv \langle a_{i,j}, d_{i,j}, C_i, \zeta_i \rangle$, where

- $a_{i,j} \in \mathbb{R}_+$ is the arrival time of $\tau_{i,j}$.
- $d_{i,j} \in \mathbb{R}_+$ is the deadline of $\tau_{i,j}$ with the constraint $d_{i,j} = a_{i,j} + D_i$.
- $\zeta_i \in \{0, 1, 2, \cdots, L-1\}$ is the criticality level which inherits from τ_i.
- $C_i \in \mathbb{R}_+^L$ is a vector of worst case execution time estimations of $\tau_{i,j}$ in each confidence level, inheriting from the WCET vector of τ_i.

A mixed-criticality recurrent task τ_i is called *periodic* if every two consecutive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ satisfy the condition $a_{i,j+1} = a_{i,j} + T_i$. A mixed-criticality recurrent task τ_i is called *sporadic* if any two consecutive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ satisfy the condition $a_{i,j+1} \geq a_{i,j} + T_i$.

Given a collection of jobs, we say that the behavior of the jobs is *criticality-λ* if none of the execution times of any job $\tau_{i,j}$ exceeds $C_i(\lambda)$. Under a given scheduling algorithm, a job $\tau_{i,j}$ is *schedulable* if and only if for any possible criticality-λ behaviors of the system $\tau_{i,j}$ always completes before its deadline. A task τ_i is *schedulable* if and only

if all the jobs released by τ_i are schedulable. A mixed-criticality system is *schedulable* if and only if all the tasks in the system are schedulable.

1.3.2 Mixed-Criticality End-to-End Systems

We consider distributed real-time applications made up of *end-to-end tasks*, each of which consists of subtasks and can be represented as a directed acyclic graph (DAG) in which the vertexes of each DAG represent subtasks and the edges represent precedence constraints among the subtasks. We call a real-time system a *mixed-criticality end-to-end system* if it occupies of more than one processor and has end-to-end tasks of different criticalities. The following definitions further explain the mixed-criticality end-to-end model.

- The system consists of a set of processors and a set of tasks.
- Each task τ_i consists of a set of subtasks $\tau_{i,1}, \tau_{i,2} \cdots \tau_{i,n}$ which forms a chain where $\tau_{i,j+1}$ is directly dependent on $\tau_{i,j}$ [1].
- Subtasks are statically assigned to processors: given a subtask $\tau_{i,j}$, we use the notation $Pr(\tau_{i,j})$ to represent the processor to which $\tau_{i,j}$ is allocated. Given a processor PR_k, we use $\tau_{i,j} \in PR_k$ to represent the relation that $\tau_{i,j}$ is allocated to processor PR_k.
- Each task τ_i is periodic, with a period T_i and a criticality level ζ_i. That is, all subtasks of τ_i share the same period and criticality level. In addition, each task has a relative *end-to-end* deadline D_i, by which the last subtask in the chain must complete its execution.
- Each task τ_i has a criticality level ζ_i.
- Each subtask $\tau_{i,j}$ has a set of worst case execution times $C_{i,j}(\zeta_i)$ for each criticality level.

[1] The analysis can be extended to tasks in which subtasks form an acyclic graphs with the help of longest path traversal in graphs. We avoid discussing such extensions for simplicity of presentation.

- Resource sharing between subtasks is not explicitly considered. If resource contention arises between subtasks of the same processor, the priority ceiling protocol [86] can be used to bound the blocking factor which can then be added to the scheduling analysis.

When processors are connected via a network, the cost of communication must be explicitly considered. In the simplest case, the maximum communication delay can be directly added to the worst execution time and overload budget. If multiple subtasks share the same processor and a dedicated DMA controller is used for the network communication in the processor, one can use critical sections to model the access to the controller and again adopt the priority ceiling protocol [86] to bound the blocking time. If a prioritized shared bus is used communication, the bus itself can be modeled as a processor with multiple subtasks executed on it.

1.4 Basic Middleware Requirements

In this section we describe requirements for the design and implementation of real-time middleware for dependent task graphs on multi-core platforms. These requirements fall into three main categories: *encapsulation*, which guides the component design presented in Section 2.1, and *real-time capabilities* and *performance optimization* which motivate the middleware architecture presented in Section 2.2.

Encapsulation Real-time middleware must provide suitable mechanisms to encapsulate subtasks (hide unneeded details while revealing other necessary ones), under a model to which schedulability analysis can be applied readily. In addition, real-time middleware must separate distinct application concerns so that they can be configured independently. To facilitate software reuse, the subtask implementation should be independent from the graph topology. That is, a subtask implementation should be agnostic to whether another subtask is upstream or downstream from it, or to how many immediate upstream or downstream subtasks it has. Subtasks should be encapsulated to provide location transparency so that the communication between

subtasks is independent of how a subtask implementation obtains its input or generates its output. Deadlines and other constraints should be specified separately and used by the middleware rather than being entangled within the subtasks' implementations. Type safety also must be enforced between subtasks, i.e., the data sent from an upstream subtask must be acceptable to its downstream subtasks.

Real-time capabilities We adopt a fixed priority periodic task model under which real-time middleware must provide ways to specify subtask priorities and strictly enforce them when dispatching subjobs, including avoiding any possible priority inversions. In addition, a release guard protocol [91] is used to avoid scheduling anomalies due to subjob release time jitter. The release guard protocol thus enforces an individual release time for each subjob, according to the system model described in Section 1.3.

Performance optimizations In addition to the requirements for encapsulation and real-time capabilities, which are common to most kinds of real-time middleware, the following optimizations are specific for dependent task graphs running on distributed multi-core platforms. First, empirical studies [23, 98] have shown that the costs of thread migration on a multi-core platform can be unpredictable and can introduce meaningful overhead. Therefore, the middleware architecture should avoid thread migration if possible. Second, memory sharing among threads may require extensive synchronization and locking control, and may experience unpredictability and additional costs because of cache effects. Third, since resource allocation and deallocation times can be large and unpredictable, real-time middleware should provide suitable interfaces for reserving required resources in advance to avoid unnecessary resource allocation or deallocation. In addition, the middleware should be able to customize communication between subtasks on the same core or on different cores of the same host, rather than always using common inter-process communication (IPC) mechanisms such as sockets or pipes which can incur larger and more variable delays.

1.5 Contributions of This Work

The primary contribution of this work is a practical implementation and evaluation of different mixed-criticality scheduling approaches, atop a real-time capable version of the commonly available Linux operating system. To our knowledge this is the first system implementation and comparative evaluation of different mixed-critical scheduling approaches for periodic tasks. Specifically, our contributions are six-fold.

- Extensions and improvement of the zero-slack scheduling algorithm and analysis to (i) accommodate execution of tasks beyond their deadlines and (ii) refine the calculation of zero-slack instants to account for forms of interference not considered in previously published analyses.
- Improved scheduling analysis for fixed-task priority and fixed-job priority mixed-criticality scheduling approaches. Especially, our schedulability evaluations show that our improved analysis can greatly enhance the schedulability of fixed job priority mixed criticality task sets.
- Simulations of schedulability under different approaches to mixed-criticality scheduling, which show that fixed-job priority based mixed-criticality scheduling algorithms provide best schedulability. However, they do not dominate other approaches in the distributed end-to-end cases.
- A specially designed middleware, MCFlow, to help the development of real-time applications with mixed-criticality end-to-end task sets on multicore platforms. MCFlow provides an efficient component model through which computations can be configured flexibly for execution within a single core, across cores of a common host, or spanning multiple hosts. Application components do not need to be modified in order to cope with the different synchronization mechanisms between cores or hosts when the allocation of subtasks to CPUs changes. MCFlow does this for them transparently, and also enforces a strict separation of timing and functional concerns so that they can be configured independently. MCFlow provides a novel event dispatching architecture that where possible uses lock free algorithms to reduce memory contention, CPU context switching, and priority inversion. To our knowledge, no other real-time middleware

is specifically designed to support directed acyclic real-time task graphs in distributed platforms consisting of multi-core hosts or to support mixed-criticality task scheduling.

- Our empirical evaluation of MCFlow in comparison to TAO [57], a widely used standards-based middleware for distributed real-time applications, shows that MCFlow performs comparably to TAO when only one core is used, and outperforms TAO when multiple cores are involved, due to the multi-core specific optimizations available in MCFlow.

- Our further evaluation of mixed-criticality scheduling atop MCFlow, which demonstrates a wide-range of mixed-criticality scheduling policies can be realized efficiently and effectively in real-time middleware. Our results show that the mixed-criticality scheduling approach with highest run-time cost imposes only 0.3% additional overhead, which demonstrates the viability of such approaches in practice.

1.6 Dissertation Organization

This dissertation is structured as follows.

In Chapter 2, we introduce the general architecture of MCFlow. We discuss (1) how to map end-to-end tasks into MCFlow component model, (2) the interfaces of MCFlow components, (3) how MCFlow enforces type safety and (4) MCFlow deployment plans which specify interconnections between user defined components, and a configuration tool to generate programs from a deployment plan. Next, we describe key design and implementation details of MCFlow. MCFlow's dispatching subsystem, which enforces real-time execution of subtasks, is in particular a notable advance in the state of the art in real-time middleware design. We conclude the chapter with experiments that evaluate MCFlow's performance and validate its design and implementation.

In Chapter 3, we discuss the mixed criticality scheduling of periodic or sporadic tasks on uniprocessor and end-to-end systems. First, we introduce a classification of existing mixed-criticality scheduling approaches on uniprocessor systems. Next, we present existing mixed-criticality scheduling approaches in detail and discuss unresolved issues

in some of those approaches. We then provide improvements to address those issues. Last, we focus on scheduling mixed-criticality end-to-end tasks in distributed real-time systems where a task may span several processors or hosts with precedence constraints. Most of the mixed-criticality scheduling approaches for uniprocessors need certain adaptations before they can be applied to end-to-end tasks. We discuss those adaptations for the end-to-end task model, and what modifications are needed to the analysis of mixed-criticality scheduling on uniprocessors before it can be applied to end-to-end systems.

In Chapter 4, we present an evaluation of different approaches to mixed-criticality scheduling in terms of task schedulability. Subsequently, we describe mixed criticality support in MCFlow and an evaluation of enforcement mechanisms for the various mixed-criticality scheduling approaches. Chapter 5 surveys related work and Chapter 6 summarizes the contributions of this dissertation and offers concluding remarks.

Chapter 2

Basic MCFlow Design and Performance Evaluation

MCFlow is a distributed real-time middleware specially designed to support the development of mixed-criticality end-to-end tasks running on multi-core platforms. In this chapter, we introduce the component models used by MCFlow and how the model facilitate the separation of timing and functional concern so they can configured independently. Next, we discuss the general architecture of MCFlow and various optimizations to inter-component communications. Finally, we conclude this chapter with empirical evaluations of MCFlow under the traditional (i.e., single criticality) real-time environment.

2.1 Encapsulation via Components

As we have described in Section 1.4, real-time middleware must provide suitable mechanisms to encapsulate subtasks under a model to which schedulability analysis can be applied readily. To address the encapsulation requirement, MCFlow provides a *component* model within which subtask implementations written by application developers are encapsulated. Unlike other popular component middleware approaches, in which an abundance of features comes at a significant cost in code size and potential run-time overhead and jitter, MCFlow takes a minimalist approach in which each component is a class that must specify only its inputs, outputs, configuration parameters, and runtime execution code. Conceptually, each component is an object with special interfaces that determine how its input and output types should be initialized.

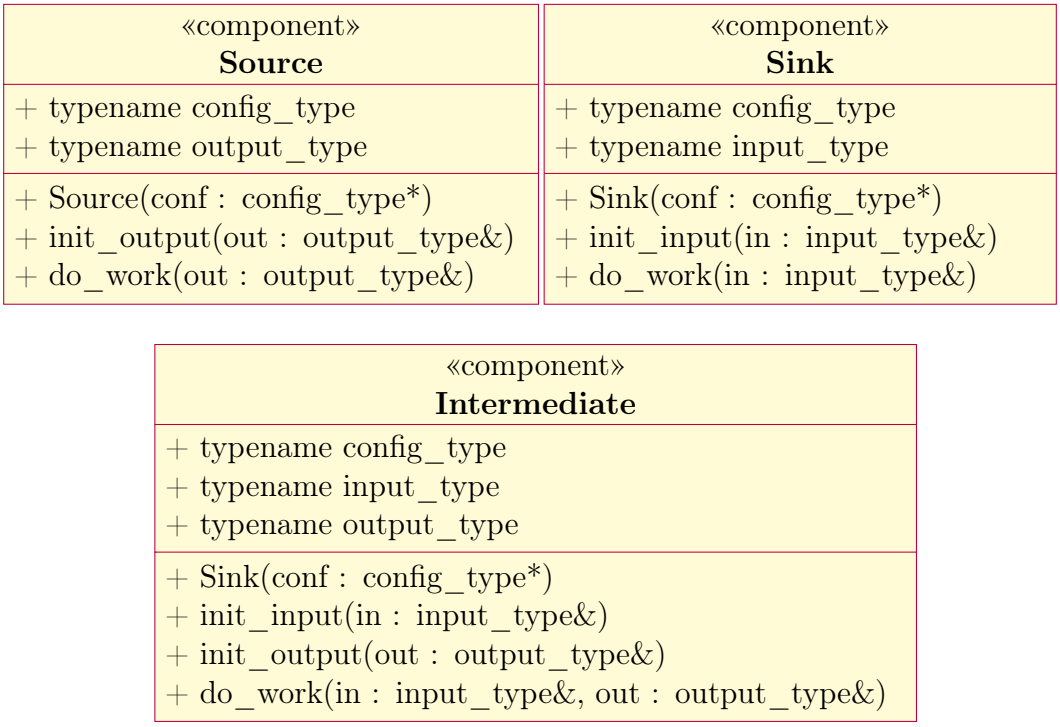


Figure 2.1: MCFlow component model

Unlike traditional object oriented frameworks, MCFlow does not enforce any inheritance hierarchy on the components but rather uses *interface polymorphism* based on template wrapper classes to encapsulate subtasks so that they can be invoked directly by a dispatcher as described in Section 2.2.3.

Components in MCFlow are classified into three categories (`source`, `intermediate` and `sink`) depending on whether they generate output and/or consume input data. Every component must provide an associated type called `config_type` and a constructor that accepts a pointer to its `config_type`. This allows developers to control the initial states of their components, such as the maximum size of a matrix or the parameters of a differential equation. The values of these configuration parameters are provided by a deployment plan as is described in Section 2.1.4.

2.1.1 Component Interfaces

MCFlow is designed to support both real-time performance and flexible component-based design. As described in Section 1.4, dynamic memory allocation may introduce high cost and jitter, and yet to forbid the use of dynamic memory at all could seriously impact the flexibility of component design. One standard way to address this issue is to estimate an upper bound on the size of memory required overall, and to preallocate enough memory at initialization time.

This rationale gives rise to the design of MCFlow's component interface: the separation of input and output initialization from the construction of the component. An application can utilize the `config_type` to set the memory size at configuration time. The input and output initialization interfaces allow MCFlow to provide appropriately sized (e.g., as in [24]) input and output *ring buffers* that support the optimizations described in Section 2.2.2. The separation of input and output initialization provides greater freedom for the framework to optimize the input and output buffers without complicating the component interface itself. Notice that whether to use a maximum memory reservation strategy at component initialization time or to use dynamic memory allocation is left to the discretion of the application developer. For systems with more stringent timing constraints advance reservation may be appropriate, while for applications with looser constraints the flexibility offered by dynamic memory allocation may be the dominant concern. MCFlow provides a convenient interface so that a developer can choose which memory allocation strategy to use based on the specific requirements of their application.

MCFlow application components are written in C++. To provide type safety, increase reusability, and avoid the overhead of virtual function calls, MCFlow does not enforce any inheritance hierarchy on the components but rather uses *interface polymorphism* based on template wrapper classes to encapsulate subtasks so that they can be invoked directly by a dispatcher as is described in Section 2.2.3.

Consider, for example, an intermediate component that may compute a Fast Fourier Transform (FFT). Note that without optimization such a component could lead to extensive memory movement among input and output buffers. There are two potential sources of such memory copying. First, an upstream subtask must copy its

Program 1 Example Intermediate Component

```
struct FFT_Component
{
  // declare parameters for component
  // configuration
  struct config_type {
    int max_size;
  };

  using std::vector;
  typedef vector<double> input_type;
  typedef vector<double> output_type;

  FFT_Component(config_type* conf)
    : max_size_(conf->max_size){}

  // used to initialize input buffer
  void init_input(vector<double>& i)
    { i.reserve(max_size_); }

  // used to initialize output buffer
  void init_input(vector<double>& o)
    { o.reserve(max_size_); }

  // used to define component behavior
  void do_work(vector<double>& input,
               vector<double>& output)
    { ... }

  int max_size_;
};
```

output result to the input buffer of its downstream subtask. Second, the downstream component may write to its own output buffer even if it can reuse its own input buffer.

Program 1 shows an example of an intermediate component that computes a Fast Fourier Transform (FFT).

One solution to this problem is to define the input and output types as pointers instead of value types. However, that would require both upstream and downstream components to change their declarations to use pointers. Declaring the `input_type` or `output_type` to be a reference is not an option, because a reference type in C++ is not default constructible or reassignable; for example it could cause compiler errors if the framework tried to create an array of references. To address this issue, MCFlow provides a simple template wrapper class `ref_t` that encapsulates a pointer so that it is reassignable and can be implicitly converted to a reference type.

In Program 1, if we change the line `typedef vector<double> input_type` to `typedef ref_t<vector<double> > input_type`, memory copying from the upstream subtask to the downstream subtask can be avoided. If we also change the `output_type` typedef and the second parameter of `do_work` to use the `ref_t` type and write `output = input;` as the last statement of the `do_work` function, we can achieve the effect of in-place memory modification.

2.1.2 Interface Type Safety

MCFlow also enforces compatibility of output and input between subtasks through its component interfaces. For example, the connection between two components is only valid if the upstream component's output type for that connection is assignable to the downstream component's input type. To overcome a potential limitation with this approach on the reusability of components, MCFlow also allows adapters for component connections to be specified. An adapter is a C or C++ function that takes the output of an upstream component and coverts it into the input of a downstream component. MCFlow's connection *ports* are essentially data members of the user defined types named `input_type` or `output_type` within each component class. Instead of copying the entire output from an upstream component to a downstream

component, a port allows the application developer to selectively connect part of an upstream component's output to all or part of a downstream component's input as long as the connection is type safe.

Since we allow components to share memory via their input and output interfaces, the lifetime of the validity of the shared memory becomes a potential issue. In MCFlow, each job execution is implicitly associated with a sequence number. The major purpose for the sequence number is to index the corresponding ring buffer for input and output queues. As long as the queue size is greater than the maximum pipeline level of any end-to-end task, the output data of the first subtask won't be overwritten until the last subtask of the end-to-end task has finished its work. Therefore the memory passed from an upstream subtask will always be valid until it finishes executing its current job. However, it is not safe to save the pointer to the memory and use it for the next occurrence of the job. In that case, the component should always copy the memory buffer's contents into its local state variables.

2.1.3 Component Communication

One of the most important features of MCFlow is that it allows communication between components to be automatically optimized regardless whether it involves intra-core, inter-core or network communication. Communication is implemented by a set of template wrapper classes for the components. These wrapper templates are highly modular and are specialized for different categories of components and their supported communication schemes. Table 2.1 shows the list of MCFlow wrappers for components. The `source_worker`, `intermediate_worker` and `sink_worker` are designed specifically for source, intermediate and sink components respectively. The `servant_worker` and `proxy_worker` templates are used for receiving and sending network messages. The `interthread_preparer`, `intrathread_preparer` and `servant_preparer` listed on the first row of Table 2.1 are used to customize and potentially optimize how a component gets its input. For example, the C++ expression `intermediate_worker<FFT_Component, interthread_perparer>` represents a subtask which accepts input and produces output when none of the communication with its upstream subtasks is through the network. If the inputs were from the network instead of intra-host communication, we would change the second template parameter

Table 2.1: Valid preparers for MCFlow worker components

	`interthread_preparer`	`intrathread_preparer`	`servant_preparer`
`source_worker`			
`intermediate_worker`	√	√	√
`sink_worker`	√	√	√
`servant_worker`			√
`proxy_worker`		√	

to be `servant_preparer`. Based on (1) the type of each component a wrapper template is designed for, and (2) the allowed type of inputs for the component, Table 2.1 shows the valid combinations to wrap a component class as a dispatchable subtask.

2.1.4 Deployment Plan and Code Generation

To make connections between subtasks even more transparent, MCFlow provides a *deployment* tool which reads the specification of a deployment plan and generates appropriate C++ source files and Makefiles according to its contents. A deployment plan's specification includes: (1) the hosts in the execution environment and their network addresses; (2) all end-to-end tasks and the subtasks they contain; (3) for each subtask the type of component used, the values for each field in the `config_type`, on which host and core the subtask should be executed, and the priority of the subtask; and (4) the connections between the subtasks.

2.2 Middleware Architecture

MCFlow enforces a crucial separation of concerns between its task management and dispatching subsystems, as Figure 2.2 illustrates. The task management subsystem creates, initializes and terminates the subtasks on each host. If a subtask throws an unrecoverable exception, the task management subsystem releases all resources previously acquired by that subtask's team. The dispatching subsystem is designed specifically to enforce real-time requirements and apply performance optimizations discussed in Section 1.4. In particular, all threads in the dispatching subsystem, and the memory resources they use, are strictly and unchangeably pinned to a specific

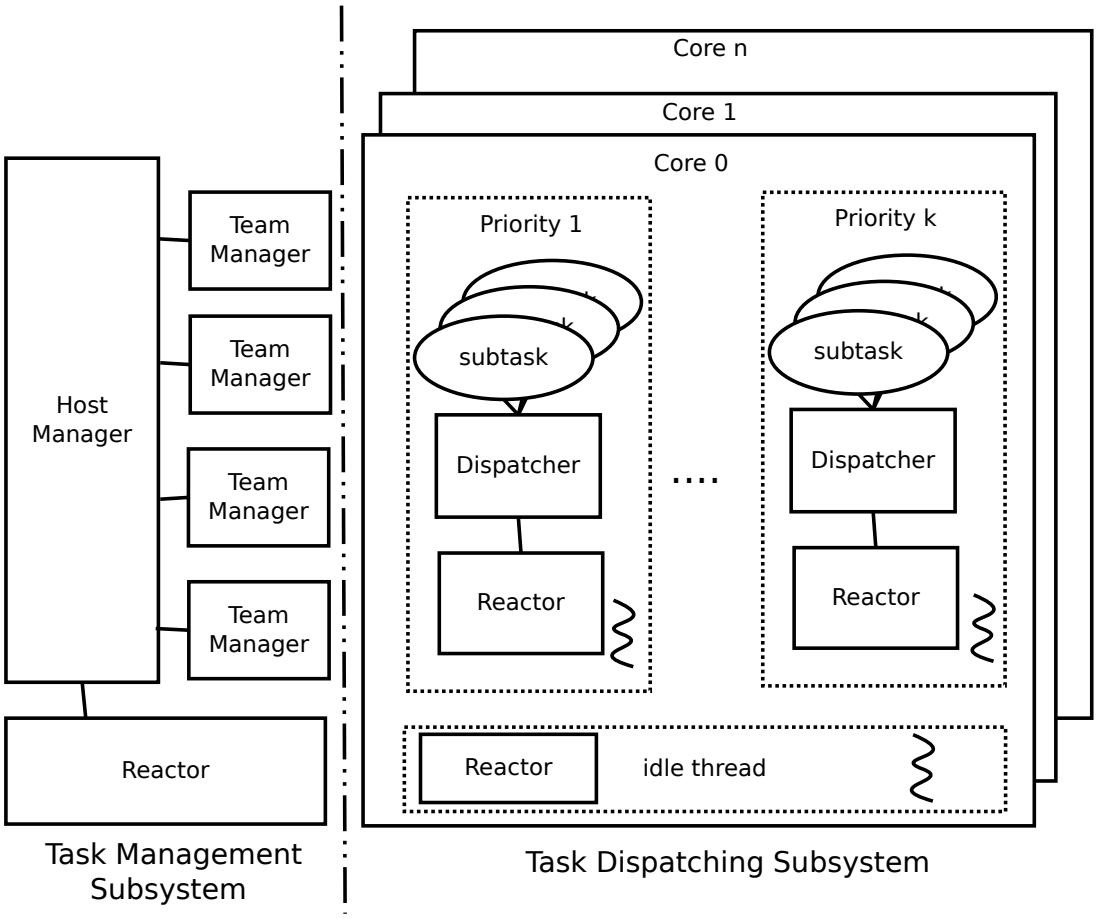


Figure 2.2: MCFlow host architecture (the squiggly arrows represent threads)

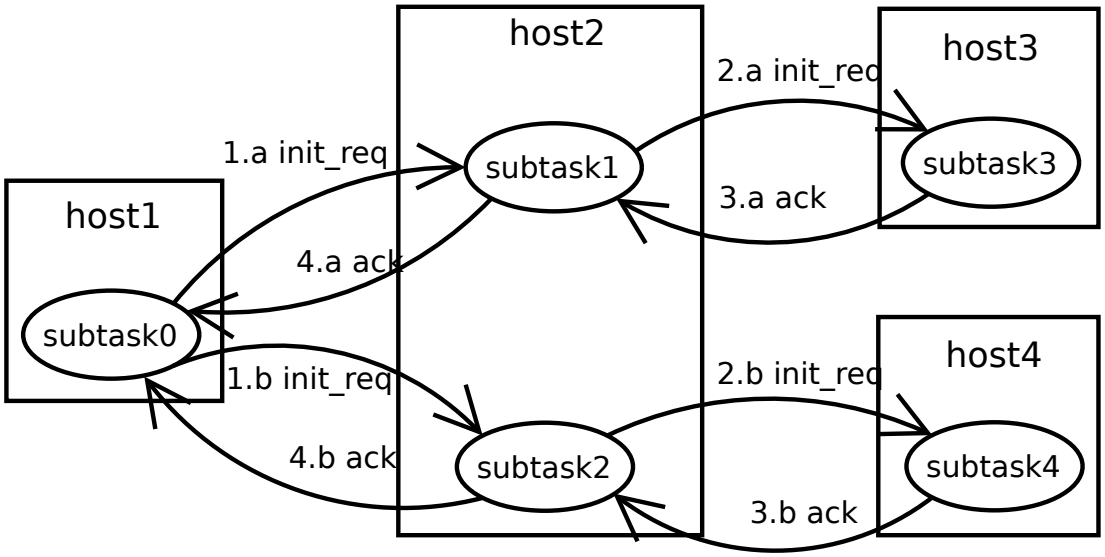


Figure 2.3: Example MCFlow initialization flow

priority and processor core. In both the task management and dispatching subsystems, events (e.g., from network I/O) are demultiplexed onto threads using the Linux `epoll` APIs according to the *reactor* architectural pattern [84] as is commonly the case in other real-time middleware architectures [57].

2.2.1 Task Management Subsystem

The creation and destruction of subtasks in a host is done by the task management subsystem. The host where an end-to-end task originates creates the subtasks assigned to it and issues initialization requests to other hosts in the system. Upon acknowledgement from the downstream hosts, it activates its local task dispatching subsystem to start real-time execution of the end-to-end task. Figure 2.3 shows an example initialization process for an end-to-end task with 5 subtasks spanning 3 hosts.

Termination of each end-to-end task is executed at its real-time priority. This is to ensure that all the subjobs have stopped executing and their subtasks can be safely deallocated. Termination may be initiated either by an upstream data source or by run-time exceptions from downstream subtasks. During team termination, the

team manager first sends a termination request to all the downstream subtasks, to be executed at their real-time priorities. Upon receiving the termination request, a subtask will stop accepting any new inputs and will pass the request to its successors. In addition, it sends an asynchronous notification[2] to the team manager to indicate that the subtask has stopped. Once the team manager receives all notifications from all subtasks in the team, it can deallocate resources reserved for the team.

2.2.2 Dispatching Subsystem

The dispatching subsystem prioritizes execution of subtasks using priority lanes as is shown in Figure 2.4. Each priority lane is a collection of threads which are allowed to share resources without contributing to priority inversion [80]. Instead of using a dedicated thread with a fixed priority to receive IPC messages, it allocates multiple threads to receive and handle messages at different priorities. Each supplier or consumer in the figure represents an upstream or downstream subtask, and each proxy or servant actually sends or receives data through an IPC channel. Priority lanes are widely used to avoid inter-process priority inversion, though different inter-thread communication (ITC) mechanisms may be used to deliver messages from servants to consumers.

For example, TAO's Real-Time Event Service [51] uses the ITC mechanism shown in Figure 2.5 to support event multicast, filtering, and correlation, through queues for consumers at different priorities within a single processor.

To support allocation of threads onto multiple processors, MC-ORB [98] uses a half-sync/half-async concurrency architecture [84] for receiving network requests, in which a dedicated thread decides in which core and at which priority each request should be handled, and then pushes each request to a designated thread pool as is shown in Figure 2.6. This supports thread migration and allows adaptive run-time load balancing of subtasks, but suffers two additional context switches for each subtask (in contrast to our system model in which requests are delivered directly to subtasks), which may impact timing guarantees especially with large numbers of subtasks.

[2] We use an `eventfd` that is provided by the current Linux kernel for lightweight event notification; it is also possible to implement asynchronous notification using a pipe in an older kernel, but with a higher cost.

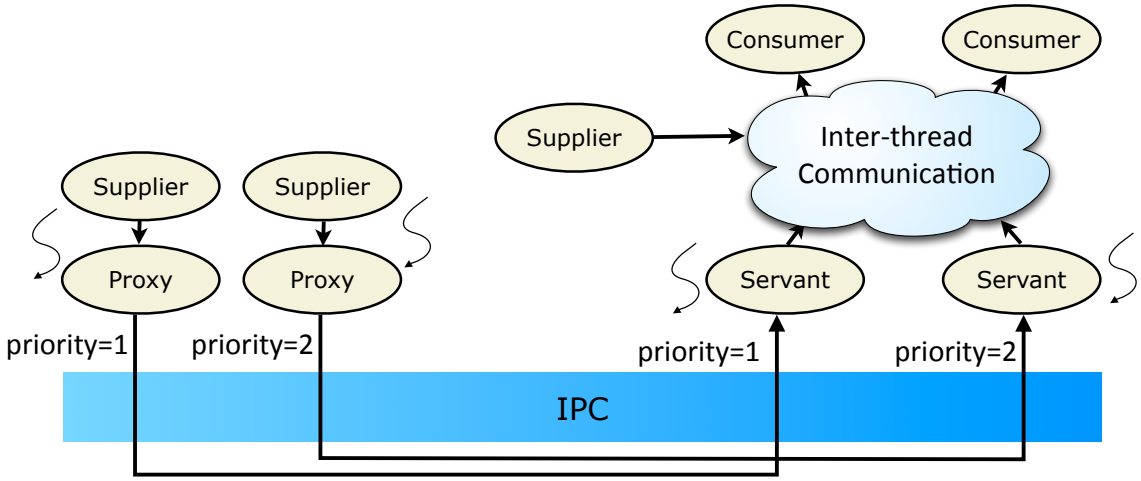


Figure 2.4: Real-time communication via priority lanes

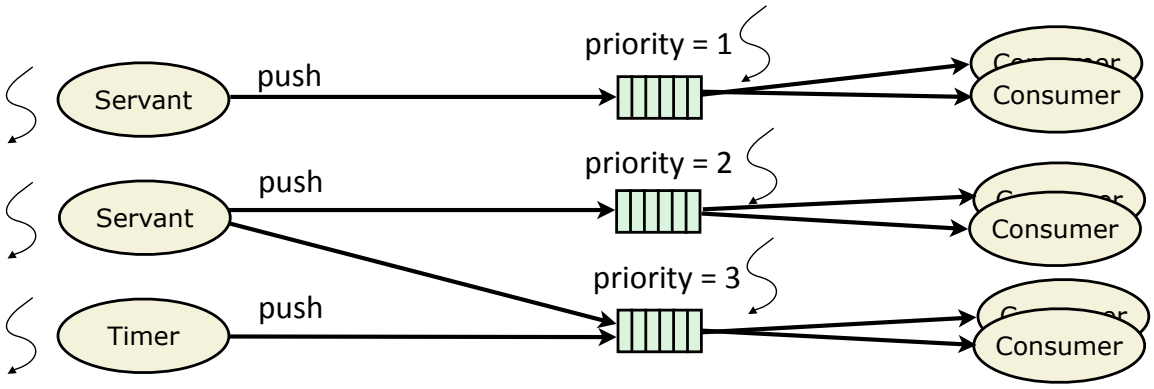


Figure 2.5: TAO Real-Time Event Service ITC mechanism

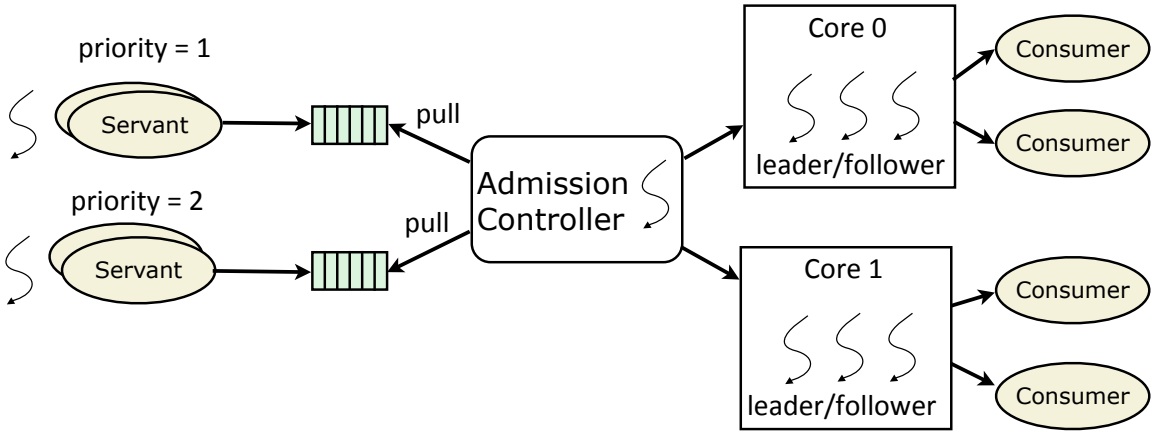


Figure 2.6: MC-ORB ITC mechanism

MCFlow instead relies on static task allocation at configuration time so that it can optimize inter-thread communication. For each subtask $T_{i,j}$, every subtask in $Pre(T_{i,j})$ is given a distinct network address and identifier for the host and core where $T_{i,j}$ should be run. The network request is then delivered directly to the appropriate core, rather than going through an intermediate thread to dispatch the request. This design choice allows efficient dispatching of network events without the extra context switches in MC-ORB.

Neither TAO's Real-Time Event Service nor MC-ORB optimizes inter-core communication, and therefore the data exchanged between subtasks must be marshaled and demarshaled even if the suppliers and consumers are on the same host. This can be very inefficient if a large amount of data needs to be exchange among dependent subtasks, which is a common case for computations like the numeric simulations mentioned in Section 1.2. Other alternatives like direct function calls passing native C++ pointers or reference counted smart pointers are possible within a single process, but they may reduce parallelism and incur variable costs for dynamic memory allocation and deallocation.

To alleviate the need for either data marshaling and demarshaling or memory allocation and deallocation, MCFlow uses the novel ITC mechanism shown in Figure 2.7. Unlike TAO's Real-Time Event Service or MC-ORB where consumer side queues are shared, in MCFlow each consumer has its own type-specific input queue and each supplier has its own type-specific output queue. Since both input queues and output queues are type-specific, data can be copied directly between them in a type-safe manner without marshaling and demarshaling.

MCFlow uses a simple lock free ring buffer to avoid mutex synchronization. If a consumer has multiple suppliers, the consumer will be dispatched if and only if all its suppliers have copied data into its input queue. Each entry in the consumer queue may have multiple data fields, where each field is used for a specific supplier as shown in Figure 2.8.

To synchronize data merging from multiple suppliers, each job has a sequence number that indexes the input and output queues. The supplier queues are also implemented as ring buffers, and use this index to retain the data for as long as the consumers need it. A sufficient size for the supplier queue can be calculated automatically [24] by

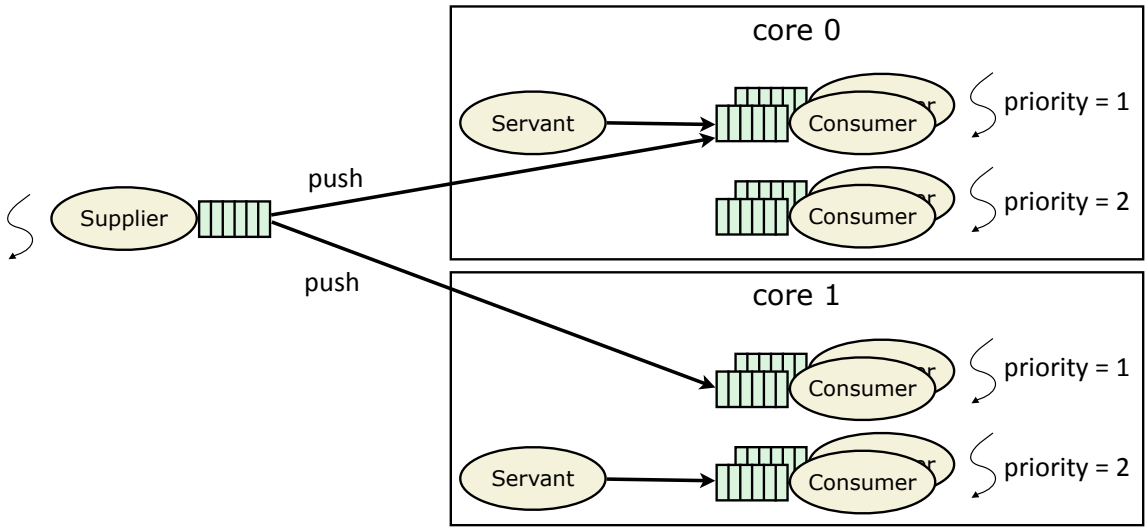


Figure 2.7: MCFlow ITC mechanism

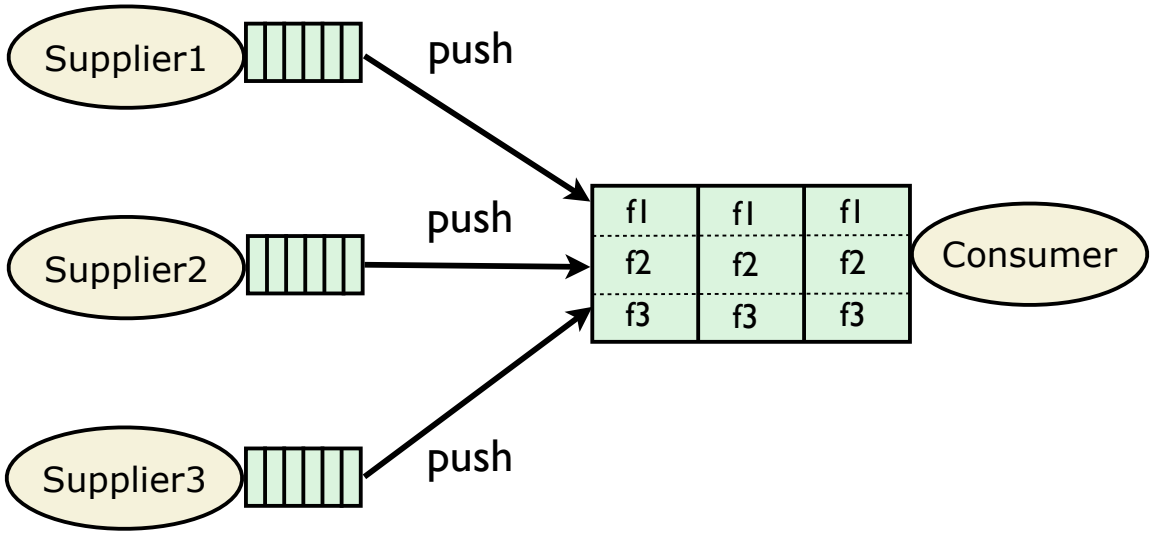


Figure 2.8: Data merge in MCFlow

the deployment tools described in Section 2.1.4, from the supplier subtask's relative deadline, its period, and the depth of the task graph in that host. The memory passed from an upstream subtask moves from queue to queue but will always be valid until the current job finishes execution. MCFlow also ensures that no two elements in a supplier or consumer ring buffer share the same cache line, by padding each element in the buffer so that its size is a multiple of the hardware cache line size, which thus avoids the *false sharing problem* [30] for access to the buffer.

2.2.3 Subtask Release Mechanism

In MCFlow a dispatcher coordinates all subtasks allocated to a specific core at a particular priority. As Figure 2.2 illustrates, each dispatcher has an associated demultiplexor for passing subtask invocation requests to it (according to the *reactor* architectural pattern [84] in which asynchronously arriving events, e.g., for network I/O, are demultiplexed by invoking specific event handlers that have registered to receive those events). In MCFlow subtasks are dispatched in this manner, and a demultiplexor can be configured either with a single thread (in which case the different subtasks are dispatched sequentially in FIFO order) or using the leader/followers pattern [84] with multiple concurrent threads waiting for events. The leader/followers configuration can be especially useful when subtasks can block in certain system calls.

Each dispatcher also manages a FIFO subtask queue and a timer queue to control when each subtask can be executed. When a subtask finishes its execution, it copies its outputs to the input queues of its immediate downstream subtasks, as described in Section 2.2.2, and then inserts those subtasks into the subtask queues of their corresponding dispatchers. After that, it sends asynchronous notifications to their designated demultiplexors. Once a demultiplexor thread picks up the notification, it processes the notification in the following steps: (1) the thread pops a subtask from subtask queue; (2) the thread checks whether the popped subtask is still being processed (by reading an atomic *in-processing* flag); this step is required when the leader/followers pattern is applied; in this case, multiple threads exist for the same core/priority and two subjobs may be executing concurrently if one thread is still processing a subjob when another notification is sent and is picked up by another thread; (3) if the subtask is not being processed, the thread then checks whether

the subtask is periodic and whether the release time for the subtask has expired; (4) if the release time has expired, the *in-processing* flag is set and the subtask is executed in the thread; otherwise the subtask is inserted into the timer queue, to be executed when the timer expires; (5) after the subtask finishes executing, it checks whether there are more inputs to be consumed and keeps executing until no inputs are available; (6) the *in-processing* flag is cleared before the thread waits again for events.

Step 3 is required to enforce release-guard semantics [91] across distributed or multi-core systems so that *intervals between release times of jobs in any subtask are never less than the period of the subtask* [69]. Besides waiting for the period boundary before the next subjob of a subtask can be executed, the release-guard protocol also allows a subjob to be executed earlier than the periodic boundary when the CPU becomes idle. To implement this feature, another *idle thread* with the lowest real-time priority for each CPU waits on a prioritized demultiplexor. Whenever the timer queue size changes, the dispatcher sends a notification to the idle thread with the new size of the queue. The idle thread can only receive those notifications when there are no other real-time subjobs executing in the CPU. Once the idle thread receives the notification, it will then send an idle notification to the highest priority dispatcher that has a nonzero timer queue size. That dispatcher will then dispatch the subtask with the earliest expiration in its timer queue. The dispatching scenario is similar for network triggered subtasks. In this case, the subtask is notified directly upon readability of the socket instead of upon the receipt of an asynchronous notification.

2.3 Performance Evaluation

In this section, we evaluate the performance of MCFlow for dispatching real-time subtasks in parallel. We examine the end-to-end latency for a single flow application and the deadline miss ratios for a multi-flow application.

Of the alternative middleware architectures discussed in Section 2.2.2, we compare MCFlow to TAO [57] RT-CORBA object request broker, to judge how much of an improvement MCFlow offers compared to a state of the art real-time middleware for distributed real-time systems. We chose TAO as a baseline because it is widely used

and is reasonably representative of other real-time ORBs such as nORB [90] and MC-ORB[3] [98].

In Section 2.3.1, we evaluate how well MCFlow can reduce the latency of a basic distributed real-time task with dependent parallel subtasks. Section 2.3.2 describes experiments designed to examine MCFlow's ability to parallelize subtasks across different number of cores. The multi-flow experiments in Section 2.3.3 evaluate how well MCFlow enforces priorities among different tasks.

The experiments described in this section were performed on two 6-core Intel core i7 980 3.3GHZ CPUs with hyper-threading enabled. Both machines ran Ubuntu Linux 10.04 with the 2.6.33-29-realtime Kernel (with the PREEMPT_RT Patch [82]). In this section, we use *CPUs* to refer to the number of logical processors recognized by the operating system, rather than the number of physical cores.

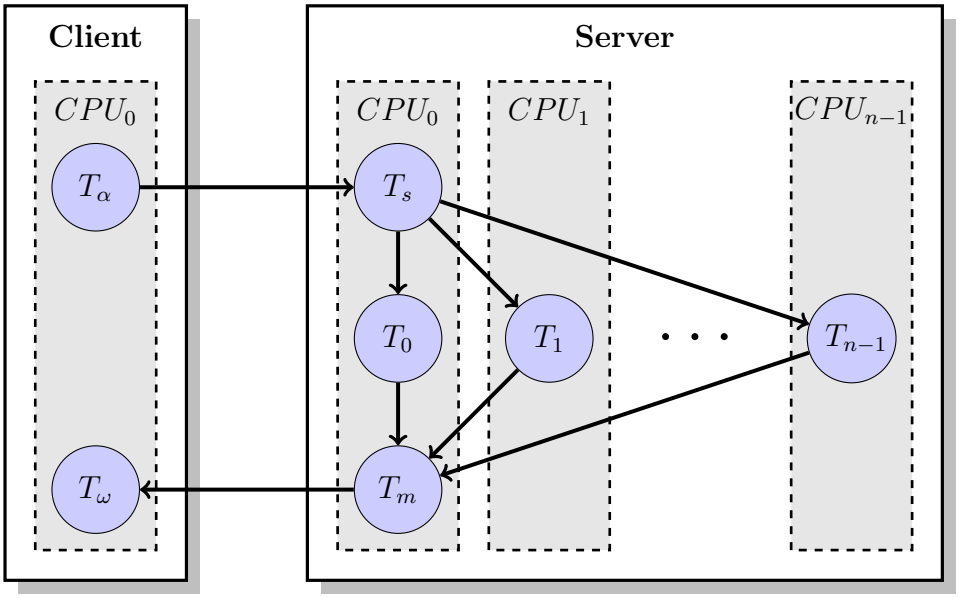


Figure 2.9: Single task experiment setup

We measure the latency of an end-to-end task spanning two hosts, as is shown in Figure 2.9. The server receives data from a client and splits the subtask computations onto a number of CPUs, merges their output into a combined result, and sends it

[3]While MC-ORB supports core-specific assignment of threads, it also may migrate threads dynamically to balance utilization at run-time, which would violate our system model assumptions described in Section 1.3.

back to the client. To evaluate how computation splitting itself affects performance, we vary the number of subtasks and the number of CPUs used together. We use T_α and T_ω to represent the data source and sink subtasks on the client; T_s and T_m to represent the data splitting and merging subtasks on the server; and T_i (where $i = 0$ to $n - 1$) for the parallel subtasks on the server. The data transmitted from T_s to each T_i, and from each T_i to T_m are 64 bytes long. The data transmitted from T_α to T_s and from T_m to T_ω are $64n$ bytes long. No extra computation is done in T_α and T_ω. The computation times for parallel subtasks T_0 throughT_{n-1} and also T_s and T_m are all 5 μs.

2.3.1 Latency Comparison

We compare the latency of these applications using MCFlow and TAO. The MCFlow version has two configurations: the first uses all of the optimizations described in Section 2.2, while the second configuration (denoted MCSock) uses only sockets for inter-component communication. These configurations are compared to examine how much the inter-core communication optimizations can improve system performance.

The TAO version also consists of two different configurations. The first uses one ORB per CPU (denoted as TAO-MORB), with each ORB allocated only one thread. Each thread is pinned to a particular CPU to avoid migration. All subtasks are assigned to their corresponding ORBs. The collocation strategy [79] used for this configuration is "per-ORB" which means the requests are optimized to use direct function calls when the caller and callee are registered with the same ORB. The second configuration uses the leader/followers pattern with only one ORB per application and n threads (denoted as TAO-SORB). In this configuration, a subtask can't be run on a fixed CPU. No collocation optimization is used for this configuration; if it did, all CORBA invocations would become function calls and thus the entire server could only run in one thread.

Figure 2.10 shows the average time from when T_s receives a request to when T_m sends a reply. Figure 2.11 shows the average time from when T_s finishes its own computation to when T_m receives the last message from any $T_i, \forall i = 0, \cdots, n - 1$. The error bars in each of these figures show the (small) standard deviation for each data point.

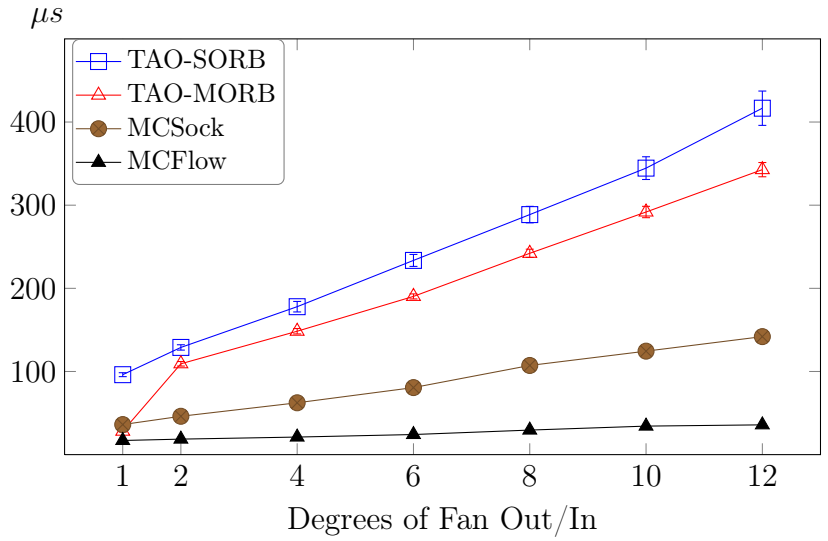


Figure 2.10: Average server response time latency per invocation

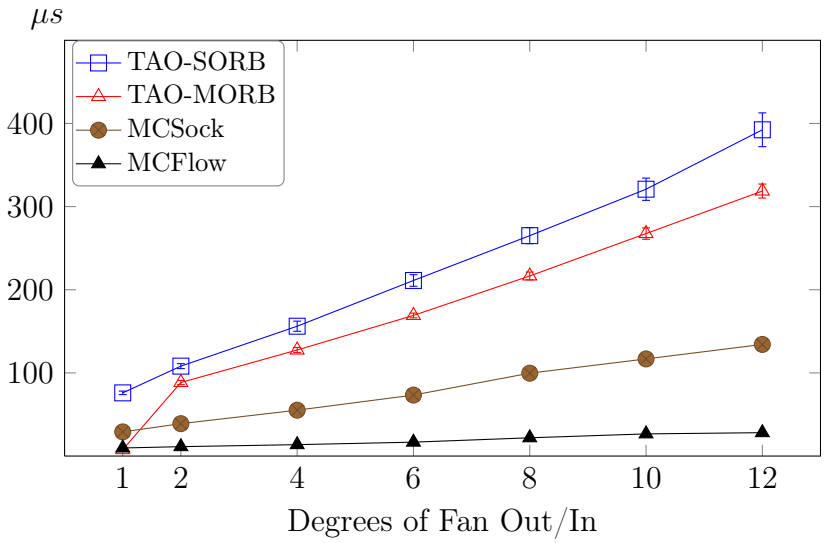


Figure 2.11: Average parallel subtask release to termination latency per invocation

Figures 2.10 and 2.11 show little difference in performance for MCFlow and TAO-MORB when there is only one CPU. However, the latency for each point along the TAO-MORB curve grows far faster than its MCFlow counterpart as the number of cores onto which tasks are split increases. Both MCSock and TAO-MORB use sockets as their only mechanism for inter-core communication, and thus the gap between MCSock and TAO-MORB stems from factors other than inter-core communication. This difference is due at least in part to the smaller memory and CPU footprint of MCFlow compared to TAO since as we note in Section 2.2.2, MCFlow avoids marshaling and demarshaling and other unnecessary features within a single host. In our testing, the sizes of the client and server versions of MCFlow were 700K and 718K bytes; the sizes for TAO were 1793K and 2072K bytes. In addition, the TAO version requires ancillary dynamic link libraries to run while MCFlow requires nothing but the standard libstdc++.

Notice that the single ORB version of TAO always performs worse than the per CPU ORB version: because an ORB maintains resources that need to be synchronized among threads, using only a single ORB may incur significant synchronization overhead. In contrast, the per CPU ORB version duplicates resources to each CPU and thus avoids such resource contention, much in the way MCFlow duplicates resources as we noted in Section 2.2.

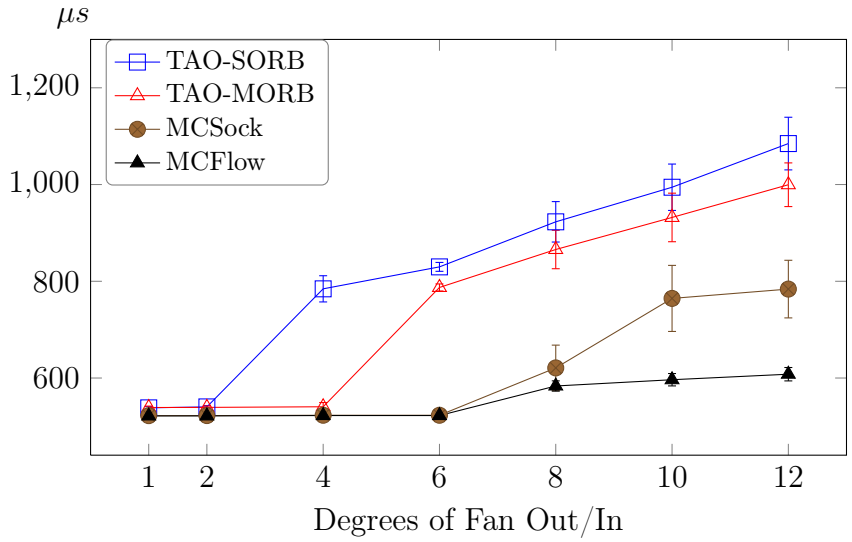


Figure 2.12: Round trip latency comparison

Figure 2.12 shows the average time measured in the client from when T_α sends a request until when T_ω receives a reply. The results shown in Figure 2.12 demonstrate that end-to-end latency differences between MCFlow and TAO are small with between one to four CPUs. This is largely because the round-trip network communication cost dominates the end-to-end latency. However, as the number of parallel subtasks increases MCFlow's ability to parallelize real-time subtasks efficiently across cores becomes the dominant factor.

2.3.2 Speedup Comparison

In the previous comparisons, we showed how the communication cost increases when the same amount of data is being transmitted on each connection, so that the total data size increases linearly with the degree of parallelism. The experiment in this subsection uses a fixed workload which does not vary with the number of cores used, and equally splits the work among cores to evaluate how workload spreading can reduce the end-to-end latency.

The experiment setup is similar to that of the previous experiment; however, T_s and T_m are only used to separate and merge data to and from the cores, and no extra workload is generated in those two subtasks. A workload of roughly 1000 μs is divided equally among n cores and processed by parallel subtasks T_0 to T_{n-1}. The data generated by T_α is 480 bytes long and is equally split among T_0 to $T_n - 1$.

Figure 2.13 shows that with up to 6 CPUS, the response time for MCFlow and for TAO with one ORB per CPU decreases when more cores are used, with MCFlow again providing lower response times. However, there is a slight increase in response times with more than 6 cores. This is due to the fact that our test machine is actually a 6 core machine with hyper-threading enabled. The response time for the TAO Single ORB configuration increases after 4 cores, again due to the higher synchronization overhead mentioned in Section 2.2.

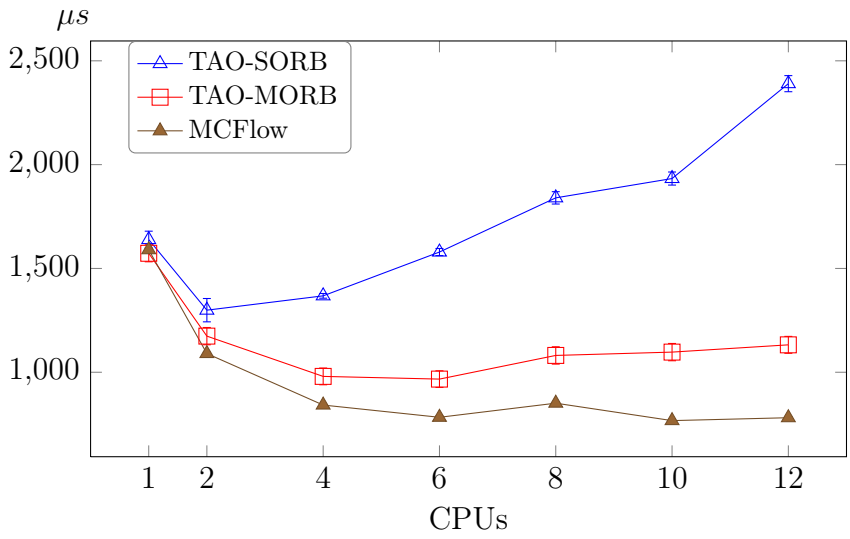


Figure 2.13: Latency comparison server result

2.3.3 Real-Time Performance

We also designed the following experiment to examine how well MCFlow can preserve the priority constraints of an application. In this experiment, we created end-to-end tasks with three different priorities: high, medium and low. All the subtasks of the high priority task have higher priority than those of the other two end-to-end tasks; similarly, all subtasks of the low priority task have lower priority than those of the other two.

Similar to the previous experiment, each end-to-end task spans two hosts and one host is used for a client which only sends periodic requests to server. The server again splits the workload onto multiple CPUs, merges the result and sends it back to the client. In our experiment, all the client subtasks are on the same machine and all server subtasks are on the other. The topology of each end-to-end task is similar to Figure 2.9; however, different CPU assignments and workloads are used.

Table 2.2 shows the CPU assignment and the workload in μs for each subtask of the high, medium, and low priority tasks. The frequencies of the high and medium priority tasks are fixed at 200Hz and 100Hz respectively. We vary the frequency of the low priority task and observe its effect on the rest of the system.

Table 2.2: (CPU, workload in μs) for each task's subtasks

	High	Med	Low
T_s	(0, 900)	(1, 900)	(2, 900)
T_m	(0, 900)	(1, 1800)	(2, 900)
T_0	(0, 1800)	(0, 0)	(0, 1800)
T_1	(1, 1800)	(1, 1800)	(1, 900)
T_2	(2, 1800)	(2, 1800)	(2, 4500)
T_3	(3, 1800)	(3, 1800)	(3, 3600)

Table 2.3: Tasks deadline miss ratios

	High	Med	Low
50 Hz	0	0	0
60 Hz	0	0	0
70 Hz	0	0	0.06
80 Hz	0	0	0.75
90 Hz	0	0	1.0

We assume the deadline of each task is equal to its period. The real-time performance of the tasks in this experiment is summarized in Table 2.3. When the rate of the low priority task is below 70 Hz, there are no deadline misses. With an increase in the low priority task's rate, it begins to miss deadlines but the other two tasks do not. Similarly, the response times of the high and medium priority tasks, shown in Table 2.4, remain stable even as the low priority task's rate increases.

Table 2.4: Average response times in μs

	High	Med	Low
50 Hz	3918	8127	14918
60 Hz	3929	8065	12615
70 Hz	3926	8063	12881
80 Hz	3931	8129	13574
90 Hz	3928	8045	18919

Chapter 3

Mixed-Criticality Scheduling

In this chapter, we discuss the mixed criticality scheduling of periodic or sporadic tasks on uniprocessor and end-to-end systems. First, we introduce a classification of existing mixed-criticality scheduling approaches on uniprocessor systems. Next, we present existing mixed-criticality scheduling approaches in detail and discuss unresolved issues in some of those approaches. We then provide improvements to address those issues. Last, we focus on scheduling mixed-criticality end-to-end tasks in distributed real-time systems where a task may span several processors or hosts with precedence constraints. Most of the mixed-criticality scheduling approaches for uniprocessors need certain adaptations before they can be applied to end-to-end tasks. We discuss those adaptations for the end-to-end task model, and what modifications are needed to the analysis of mixed-criticality scheduling on uniprocessors before it can be applied to end-to-end systems.

3.1 Classification of Mixed-Criticality Scheduling Approaches

In recent years, several algorithms and approaches have been published which focus on the scheduling of mixed-criticality uniprocessor systems. In this chapter, we classify these approaches according to their run-time enforcement mechanisms and means of priority assignment. Run-time enforcement mechanisms determine how long a job can execute and when a job should be aborted or terminate. Priority assignment determines the precedence of execution among jobs which are eligible to run at any

given time. Notice that run-time enforcement mechanisms and priority assignment are not always orthogonal, as some run-time enforcement mechanisms can only be coupled with certain priority assignment strategies.

3.1.1 Run-Time Enforcement Mechanisms

- Zero Slack Scheduling (ZSS): Each task has a fixed zero-slack instant which is offset by a time interval relative to the release time of a job and is not associated with the run-time progression of the job. A job which has not signaled completion after its zero slack instant is said to be in *critical mode*. Whenever a job enters its critical mode, all lower criticality tasks should be suspended. In section 3.2, we discuss zero slack scheduling in detail.

- Period Transformation (PT): The periods of some tasks are equally divided into several segments. The execution time of a task within a period is also equally distributed into the segments to become the execution budgets of the task in the segment.

- Static Mixed Criticality (SMC): all jobs of a task τ_i can execute up to their representative execution time $C_i(\zeta_i)$, but are prevented from executing further.

- Adaptive Mixed Criticality (AMC): There is a system wide criticality level indicator, initialized to 0. Whenever the criticality level indicator is κ, and there exists a task τ_i which executes up to $C_i(\kappa)$ without signaling completion, the criticality level indicator is increased to $\kappa + 1$ and all tasks whose criticality levels are less than or equal to κ would be prevented from execution.

3.1.2 Priority Assignment

The purpose of priority assignment is to determine which job has precedence when more than one job is eligible for execution. According to [21], priority assignment algorithms for a periodic or sporadic mixed-criticality system can be categorized into three different classes: fixed task priority (FTP), fixed job priority (FJP) and dynamic priority.

Fixed Task Priority (FTP): For fixed task priority assignment, all jobs of the same task are statically assigned the same priority. The most basic strategies for FTP are rate monotonic (RM) and criticality monotonic (CM) which assign higher priority to higher rate or higher criticality tasks, respectively. Other mixed-criticality scheduling algorithms are also designed to assume a specific priority assignment strategy. For example, zero slack scheduling is coupled with rate monotonic assignment; period transformation chooses criticality monotonic assignment but transforms the task periods so that it can behave in a rate monotonic fashion. Traditionally, in rate monotonic scheduling (RMS) and criticality monotonic scheduling (CMS), their respective priority assignment strategy is coupled with a run-time enforcement mechanism which allows jobs to executed up until they finish. In this dissertation, we will follow this tradition (i.e., RMS=RM+SMC, CMS=CM+SMC) for the convenience of our discussion.

Another notable strategy is *Audsley's priority assignment method* which is based on two important observations: (1) the response time of a task τ_i is determined if the set of its higher priority tasks (H_i) is known, regardless of the relative priority ordering within H_i; and (2) if a task is schedulable at a given priority level, then it remains schedulable when it is assigned a higher priority. The algorithm operates in increasing priority order, at each step selecting a task and, if it is schedulable, assigning it the current priority and then moving to the next higher priority; otherwise, another task is selected at the current priority level. The algorithm terminates when all tasks are assigned priorities or when no remaining task is schedulable at the current priority.

Audsley's approach can be coupled with SMC or AMC to improve schedulability over RMS or CMS. However, Audsley's approach requires scheduling analysis to check whether a task can be schedulable at a given priority. Given a task τ_i, a scheduling analysis is used to calculate the response time R_i of τ_i. If R_i is smaller than the respective deadline D_i of τ_i, then τ_i is considered schedulable.

For the rest of this dissertation, the acronym FTP will specifically refer to Audsley's priority assignment approach unless otherwise specified.

For CMS, RMS and FTP-SMC, the response time R_i^* of a task τ_i is τ_i's worst case execution time $C_i(\zeta_i)$ plus the ζ_i confidence level time demand for H_i; i.e.,

$$R_i^* = C_i(\zeta_i) + \sum_{\tau_j \in H_i} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j(\zeta_i). \tag{3.1}$$

As for period transformation, the response time of τ_i is the fixed point of

$$R_i = \sum_{\tau_j : \pi_j \geq \pi_i} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j(\zeta_i) + \min \left(\left\lceil \frac{R_i \mod T_j}{T_j / n_j} \right\rceil \frac{C_j(\zeta_j)}{n_j}, C_j(\zeta_i) \right) \right), \tag{3.2}$$

To determine whether a task is schedulable, we can simply check if the computed response time of a task is smaller than its deadline.

On the other hand, the response time analysis for ZSS and FTP-AMC are a lot more complicated. ZSS was proposed by de Niz et al. [39] in the effort to overcome the limitations of DM and PT. However, there still exist some issues with their scheduling method and analysis. In Section 3.2, we discuss those issues and how to address them. Barauh et al. [19] provided two sufficient analysis methods for a two-criticality FTP-AMC system. In Section 3.3, we generalize Barauh's analysis to systems with more than 2 criticality levels; in addition, we refine Barauh's second method to provide a tighter bound than the original.

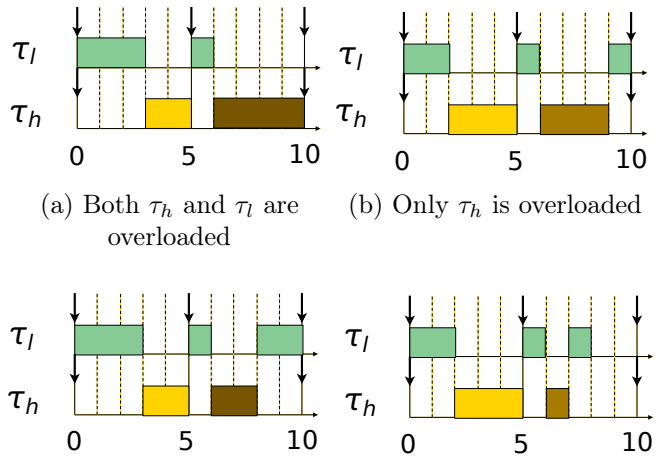
Fixed Job Priority (FJP): A fixed job priority strategy assigns priorities to each job of a system regardless whether two jobs are of the same task. Since jobs of the same task may have different priorities, the priority assignment (at least in part) needs to be deferred to run-time instead of binding it statically as in FTP. On the other hand, fixed job priority will dominate fixed task priority in terms of schedulability in theory because every FTP assignment is also an FJP assignment [21]. However, given a task Γ which is FTP schedulable, whether an FJP specific assignment algorithm can always find a feasible job priority assignment for Γ without resolving FTP analysis is yet another question. We discuss FJP further in Section 3.4.

Table 3.1: Summary of approaches for periodic or sporadic mixed-criticality tasks

		SMC	PT	ZSS	AMC
FTP	CM	Vestal2007 [95]	Vestal2007 [95]		
	RM	Vestal2007 [95]		Niz2009 [39]	
	Audsley	Vestal2007 [95] Baruah2008 [21] Baruah2011 [19]			Baruah2011 [19]
FJP	Audsley	Baruah2008 [21]			Li2010 [66] Guan2011 [50]
DP	EDF	Baruah2008 [21]			

Dynamic Priority: Dynamic priority refers to the strategies in which priorities of jobs can change between their release time and completion time. Strictly speaking, all the previously discussed mixed-criticality scheduling algorithms except RMS, CMS and FTP-SMC can be considered to be dynamic priority according to the above definition, because those algorithms require the scheduler to abort and/or suspend jobs before the jobs' completion under certain conditions. For sake of discussion, however, we exclude job abort/suspend as a form of priority change. Thus, we restrict dynamic priorities to the earliest deadline first (EDF) strategy.

EDF has been proved to be optimal for uniprocessor single-criticality systems. That is, a task set which is un-schedulable under EDF will not be schedulable under any other scheduling policies for uniprocessor systems. However, that property does not hold for mixed criticality systems. In [21], Barauh et. al. proved FTP and DP are incomparable because there exist task sets which are FTP-schedulable (or FJP-schedulable) but not DP-schedulable, and vice versa. In addition, they proved a task is DP-schedulable if and only if it is schedulable under the traditional EDF analysis. That is, the execution time specification of different confidence levels can not help schedulability as it does for FTP and FJP. Therefore, we will not further discuss DP for the rest of the dissertation.



(a) Both τ_h and τ_l are overloaded

(b) Only τ_h is overloaded

(c) Only τ_l is overloaded

(d) None is overlaoded

Figure 3.1: Two tasks example under ZSRM scheduling policy

3.2 Zero-Slack Scheduling and our Improvements

Zero-slack scheduling is a bi-modal scheduling policy, where every task has a *normal mode* and a *critical mode*. When a task is in its normal mode, it is scheduled based on its priority (assigned by a rate-monotonic or deadline-monotonic policy). When a task τ_i is in critical mode, the scheduler will suspend all lower-criticality tasks; in other words, τ_i will steal slack from lower-criticality tasks when it is in critical mode.

Figure 3.1 shows four scenarios using zero-slack rate-monotonic (ZSRM) scheduling for the task set example from Table 1.1. In these scenarios, τ_h switches from normal mode to the critical mode at time 6, and preempts τ_l to continue its execution. The instant to switch mode by a task is called the *zero-slack instant* (ZSI). As we can see, both τ_l and τ_h can finish before their deadlines when there is no overload; moreover, τ_h will never miss its deadline under overload conditions and thus this approach also avoids criticality inversion.

There are also two important issues with the zero-slack scheduling approach which must be addressed in practice to ensure that overloaded lower-criticality tasks cannot impair the schedulability of higher-criticality tasks: (1) since it is difficult to simply halt threads safely atop commonly available operating systems, the implicit assumption that real-time tasks that miss their deadlines are simply dropped rather than

being allowed to continue to run must be removed; (2) a particular form of interference that is not accounted for in the previously published analyses of zero-slack scheduling also must be addressed.

The center piece of zero-slack scheduling [39] is to decide the zero-slack instant (ZSI) of each task. Each ZSI may be computed offline and then provided to the scheduler for run-time enforcement. The objective of the ZSI computation is to find the latest possible instant for a task τ_i to switch mode to reduce its impact on the schedulability of lower-criticality tasks while still maintaining the schedulability of τ_i. In [39], de Niz et al. detailed such an algorithm for calculating ZSIs for independent task sets on uniprocessor systems.

The algorithm starts with the worst case assumption that τ_i is only executed in critical mode. Based on this assumption, it computes the time k_i needed for a job $J_{i,1}$ of τ_i to execute up to its overload budget C_i^o under interference from its higher-criticality tasks. Let the release time of $J_{i,1}$ be time 0 and the deadline of τ_i be D_i. Then $t = D_i - k_i$ is the instant that $J_{i,1}$ can switch from the normal mode to the critical mode, so that $J_{i,1}$ will meet its deadline even when it is overloaded. However, setting the ZSI of τ_i to $D_i - k_i$ so that $J_{i,1}$ switches mode at that time may be too pessimistic because $J_{i,1}$ may have executed for a certain amount of time in normal mode and thus the time budget in critical mode can be over-estimated. To reduce this pessimism, the algorithm finds the minimum amount (θ_i) of slack available for a task in normal mode and then deducts that slack from the overload budget. With the reduced budget in critical mode, the ZSI Z_i of τ_i then can be moved closer to the deadline. The algorithm repeats this recalculation of k_i and Z_i until no more slack is available in normal mode for τ_i.

How much slack is available for a task τ_i to be executed in normal mode is affected by the ZSIs of other tasks which may interfere with τ_i. That is, there are dependency for ZSI calculations among tasks. To make the ZSI of a task as late as possible, the algorithm calculates the ZSIs of all tasks with an assumption of maximum interference (i.e., $\theta_i = 0$ for all τ_i), updates θ_i with each computed ZSI, re-calculates all ZSIs with the updated θ_i, and then continues until the ZSIs of all tasks converge. Since

the algorithm relies on the convergence of ZSIs, [39] also provides a proof that the algorithm will converge as long as the deadline of each task is less than its period[4].

3.2.1 Execution After a Missed Deadline

The original zero-slack scheduler [39] is based on the scheduling guarantee that if a task τ_i is admitted, it will be able to run up to its overloaded budget C_i^o within its deadline as long as no higher-criticality task is overloaded. A task is referred to as *schedulable* if it satisfies this scheduling guarantee. However, this is based on the assumption that no lower-criticality task misses its deadline or that if it does it is simply dropped rather than allowing it to execute beyond its deadline.

Table 3.2: A two task ZSRM example

Task	C^n	C^o	Period	Criticality	Priority	ZSI
τ_1	4	5	9	2	Low	6
τ_2	2	3	5	1	High	0

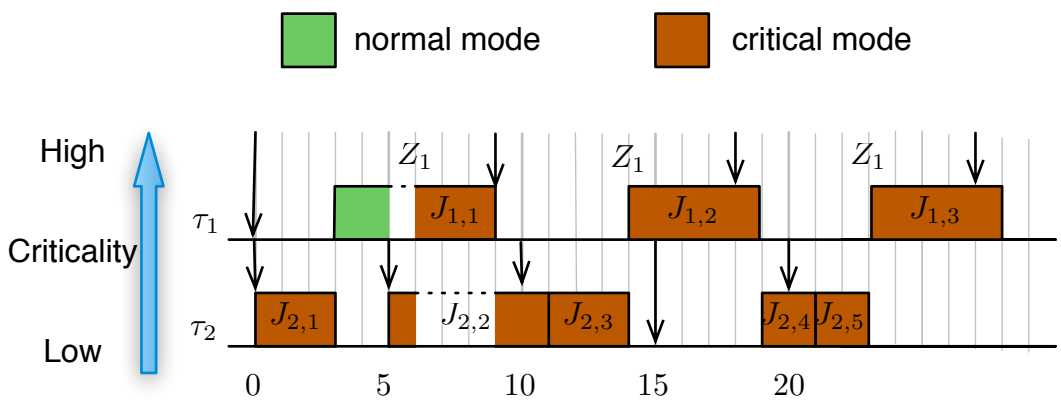


Figure 3.2: Zero-slack rate-monotonic scheduling of the task set in Table 3.2

Figure 3.2 illustrates that, in the example task set shown in Table 3.2, the overloading of a lower-criticality task could trigger the deadline miss of a higher-criticality task under the original zero-slack scheduling approach in [39]. In this example, all jobs of τ_1 are overloaded and run for 5 time units, and $J_{2,1}$, $J_{2,2}$, and $J_{2,3}$ are also overloaded and run for 3 time units. From Figure 3.2, we can see that job $J_{1,2}$ misses its deadline because the lower-criticality task τ_2 misses its deadline. Furthermore, $J_{2,3}$ also misses

[4]The algorithm from [39] is listed in Appendix A

its deadline. In other words, $J_{1,2}$ and $J_{1,3}$ break the scheduling guarantee even though no higher-criticality task is present.

3.2.2 Design Challenges

In theory, this problem can be solved by terminating a job when it misses its deadline. In practice, this may be problematic because the target job could be holding resources such as mutexes, which could lead to deadlocks and other problems. Except in special cases where jobs of the same task cannot share resources or where tasks can be made aware of their deadlines' expirations and can cooperatively release resources and halt execution, that approach is thus impractical on standard platforms.

Tindell [94] offers a way to deal with a similar issue when a job arrives before the previous job of that task has completed in a hard real-time fixed-priority system: the new arrival is deemed to have a lower priority, and is therefore prevented from executing until after the previous invocation terminates.

In a mixed-criticality system, the issue is more complicated because a deadline miss may not only affect the previous invocation, but also lower-priority higher-criticality tasks. One possible way to address this issue is to allow tasks to miss their deadlines and incorporate the deadline miss scenarios into the ZSI calculation. To analyze the impact of a lower-criticality task τ_l on a higher-criticality task τ_h, it is necessary to bound how late τ_l can execute beyond its deadline which in turn requires analysis of the interference from all higher priority and/or higher criticality tasks on τ_l when they are overloaded. Consequently, the analysis would be a response time analysis when every task is overloaded, which may be very pessimistic and leave no slack for τ_l in normal mode.

3.2.3 Solution Approach

A better approach is for the scheduler to demote task τ_i to the lowest priority when it misses its deadline; at the same time, all lower-criticality tasks have to be suspended and can be restored to their original priorities only if the job that missed its deadline

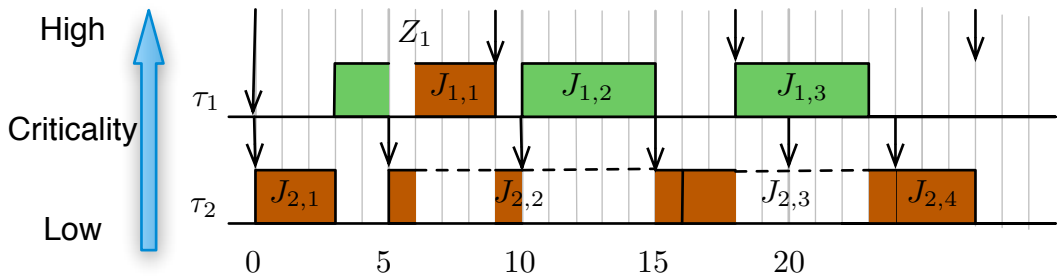


Figure 3.3: Illustration of zero-slack scheduling with demotion-on-deadline rule for the task set in Table 3.2

terminates. τ_i can miss its deadline only if one or more higher-criticality tasks is overloaded, and thus neither τ_i nor its lower-criticality tasks are required to remain schedulable. In the scenario where more than one task misses its deadline, only the ones at the highest criticality level among them will be in the runnable state with the lowest priority level; the others will be suspended. We refer to this new scheduling rule as the *demotion-on-deadline rule*. Figure 3.3 shows the schedule for the task set in Table 3.2 using the demotion-on-deadline rule.

3.2.4 Unaccounted Interference

Let π_i be the priority of task τ_i, with a larger value representing higher priority. Let $H_i = \{\tau_j | \pi_j \geq \pi_i\}$ and $L_i = \{\tau_j | \pi_j < \pi_i\}$. We also use the superscript of H_i and L_i to constraint the criticality levels within the task set. For example, $H_i^{\zeta > \zeta_i} = \{\tau_j | \pi_j \geq \pi_i \text{ and } \zeta_j > \zeta_i\}$. The original analysis of ZSI calculation [39] is based on a particular worst case phasing assumption: given a job $J_{i,1}$ of task τ_i which is released at time 0, $J_{i,1}$ suffers the maximum interference while in normal mode from τ_j when the jobs of higher-or-same-priority tasks are released at time 0. In addition, the instant is also aligned to the ZSIs of the jobs from $L_i^{\zeta > \zeta_i}$. However, that formulation considers only tasks that can interfere directly with τ_i, although some tasks which cannot preempt τ_i directly may also interfere with τ_i through tasks from the set $H_i^{\zeta < \zeta_i}$.

For example, consider the task set in Table 3.3 scheduled with ZSS. Figure 3.4 shows a schedule where the second and third jobs of τ_3 are overloaded and run for 3 and 5 time units respectively; in addition, the first job of τ_2 runs only for 1 time unit

Table 3.3: Example task set for worst case phasing condition

	C^n	C^o	Period	Criticality	Priority	ZSI
τ_1	2	5	10	3	Med	8
τ_2	4	5	15	2	Low	9
τ_3	2	4	7	1	High	0

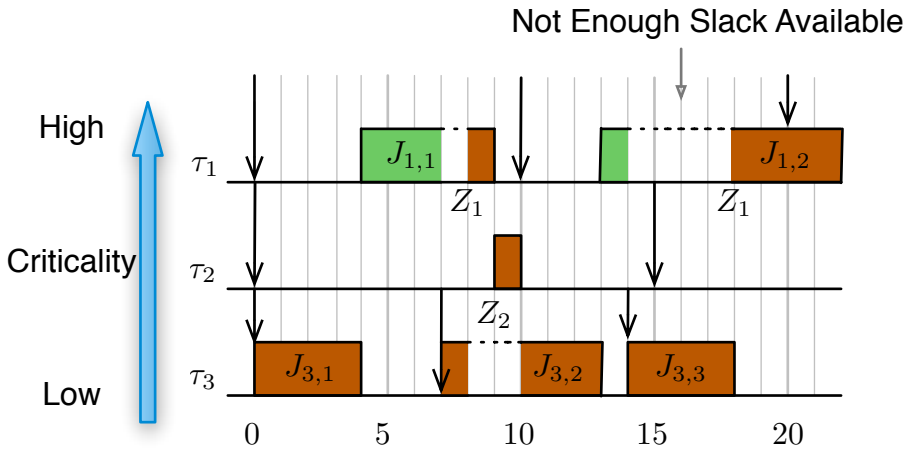


Figure 3.4: Zero-slack scheduling of the task set in Table 3.3

(which is valid because the specification does not require τ_2 to run for at least C_2^n time units). By the original analysis, the normal mode slack vector of τ_1 is $\{(4, 3)\}$, i.e., τ_1 can run for at least three time unit starting from time 4. However, as is shown in Figure 3.4, $J_{1,2}$ has only one unit of slack before its ZSI (at time 18). If $J_{1,2}$ is also overloaded and runs for more than 3 time units, τ_1 would miss its deadline because of the interference from lower-criticality task τ_3, and thus the scheduling guarantee would be violated.

We therefore introduce a revised ZSI calculation algorithm which addresses the previously unaccounted interference. We define $\theta_i(\zeta_m)$ to be the minimum slack that can be used by τ_i before Z_i at criticality level ζ_m. This value is initialized to 0 for all tasks and can be increased during the ZSI calculation.

Let $C_j(\zeta_m)$ be the maximum execution time of τ_j at criticality level ζ_m, as described in Equation 1.1. In addition, let I_j^i be the effective execution interval of τ_j that can interfere with τ_i at criticality level ζ_m, which can be expressed by the following

equation if the priority assignment is rate-monotonic or deadline-monotonic:

$$I_j^i(\zeta_m) \quad = \quad \begin{cases} \max(C_j(\zeta_m) - \theta_j(\zeta_m), 0) & \text{if } \tau_j \in L_i^{\zeta > \zeta_i}, \\ C_j(\zeta_m) & \text{otherwise.} \end{cases} \tag{3.3}$$

The rationale for this equation is based on the observation that when $\tau_j \in L_i^{\zeta > \zeta_i}$, the minimum amount of time τ_j spends in its normal mode is θ_j and τ_j only interferes with τ_i when τ_j is in critical mode; therefore, I_j^i is $C_j(\zeta_m) - \theta_j(\zeta_m)$ (bounded by 0 if $C_j(\zeta_m) < \theta_j(\zeta_m)$).

3.2.5 Worst Case Alignment

To obtain θ_i, we need to know the maximum possible amount of interference τ_i can suffer from other tasks. Let $\Gamma_i^n = L_i^{\zeta > \zeta_i} \cup H_i$ be the set of interfering tasks for task τ_i in normal mode. The ZSI calculation in [39] assumed that all release times of tasks from $\Gamma_i^n - L_i^{\zeta > \zeta_i}$ are aligned to the release time of τ_i, as are the ZSIs of tasks from $L_i^{\zeta > \zeta_i}$. However, as we show in Figure 3.4, this may not be the worst case phasing condition.

The key to the worst case phasing condition for zero-slack scheduling is the alignment between τ_i and the other tasks in Γ_i^n. Let $t_{i,1}^r$ be the time at which job $J_{i,1}$ is released; let $J_{j,0}$ be the last job of $\tau_j \in \Gamma_i^n$ which is released no later than $t_{i,1}^r$. To maximize the interference with τ_i from τ_j, the time when $J_{j,0}$ is able to interfere with τ_i should be no earlier than $t_{i,1}^r$ and as close to $t_{i,1}^r$ as possible. For the example in Figure 3.4, τ_1 suffers the maximum interference from τ_3 when the instant $t_{3,2}^b$ at which $J_{3,2}$ starts execution aligns with the release time of $J_{1,2}$, and then $J_{3,3}$ releases immediately after $J_{3,2}$ terminates. Based on this observation, in Theorem 1 we formally state the conditions for the worst case phasing that maximizes the interference that must be considered for the ZSI calculation.

Theorem 1. *Given two jobs $J_{i,1}$ and $J_{j,0}$ of tasks τ_i and τ_j respectively, let $t_{j,0}^r$ and $t_{j,0}^f$ be the times when $J_{j,0}$ is released and when it finishes its execution, respectively. Let $t_{j,0}^b$ be an instant between $t_{j,0}^r$ and $t_{j,0}^f$ before $J_{j,0}$ starts to execute, and let t be a time interval starting from $t_{j,0}^b$. Further, given that no task τ_k that satisfies the following two conditions is executed within the interval $[t_{j,0}^b, t_{j,0}^f]$: (1) $\tau_k \in L_i^{\zeta \leq \zeta_i}$ and*

$\zeta_k \geq \zeta_j$, and (2) $\tau_k \in L_i^{\zeta>\zeta_i}$ and τ_k is in normal mode, then $J_{i,1}$ suffers the maximum interference from τ_j in the interval t if its release time $t_{i,1}^r$ is aligned with the time $t_{j,0}^b$.

Proof. Illustrated in Figure 3.5, $J_{j,0}$ cannot be executed before $t_{j,0}^b$; therefore, $J_{i,1}$ will always suffer less interference from the subsequent jobs of $J_{j,0}$ within the interval t if $t_{i,1}^r \in [t_{j,0}^r\,,\, t_{j,0}^b)$. By definition, any task τ_k that satisfies the above two conditions cannot be executed within the interval $[t_i^r, t_i^f]$. If τ_k is executed before t_j^f, $J_{i,1}$ must have been finished; i.e., $t_i^f < t_j^f$. In this case, $J_{j,0}$ could not produce maximum interference with $J_{i,1}$. Therefore, the worst case occurs when task execution within $[t_{j,0}^b\,,\, t_{j,0}^f]$ can interfere with $J_{i,1}$, and $J_{i,1}$ will suffer less interference if $t_{i,1}^r \in (t_{j,0}^b\,,\, t_{j,0}^f]$. As a consequence, $J_{i,1}$ suffers the maximum interference from τ_j when $t_{j,0}^b = t_{i,1}^r$. □

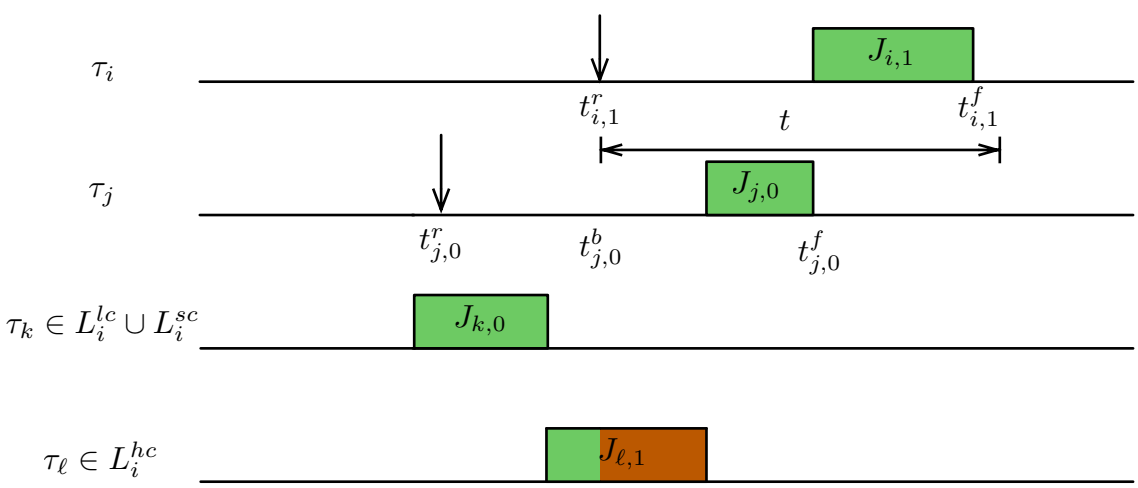


Figure 3.5: Illustration of Theorem 1

Lemma 1. *Given two jobs $J_{i,1}$ and $J_{j,0}$ of tasks τ_i and τ_j respectively where $J_{j,0}$ is released no later than $J_{i,1}$ and finishes after the release of $J_{i,1}$. In addition, $\tau_j \in H_i^{\zeta>\zeta_i} \cup H_i^{sc}$. $J_{i,1}$ suffers the maximum interference from τ_j in interval t after the release of $J_{i,1}$ when the release time of $J_{j,0}$ is aligned to that of $J_{i,1}$.*

Proof. Since $\tau_j \in H_i^{\zeta\geq\zeta_i}$, it is impossible for any task $\tau_k \in L_i^{\zeta\leq\zeta_i}$ or $\tau_k \in L^{\zeta>\zeta_i}$ to be executed before $t_{j,0}^b$ while it is in normal mode; therefore $t_{j,0}^b = t_{j,0}^r$. From Theorem 1, $J_{i,1}$ suffers the maximum interference from τ_j when $t_{j,0}^b = t_{i,1}^r$. As a result, the worst case phasing is when $t_{j,0}^r = t_{i,1}^r$. □

3.2.6 Refining the ZSI Calculation

Since rate-monotonic scheduling is a special case of deadline monotonic scheduling where deadlines of tasks are equal to their periods, we analyze the worst case phasing of zero-slack deadline monotonic scheduling instead of zero-slack rate-monotonic scheduling (while in [39] the shift from ZSRM to ZSDM involves only a single variable, the shift here is more involved, but allows a more general domain to be addressed). For simplicity of discussion, we assume that a task τ_i does not miss its deadline because even if it does the demotion-on-deadline rule discussed in Section 3.2.1 would prevent τ_i from interfering with higher-criticality tasks.

To explore the interference relationships among tasks, we bound the total time demand that can be generated by a task set as a whole. For this purpose we introduce $\delta_i^n(\zeta_m, \tau_j, t)$, the total amount of time demand generated by jobs of τ_j after a job J_i of τ_i is released and before J_i enters critical mode at criticality level ζ_m. Similarly, let $\delta_i^c(\zeta_m, \tau_j, t)$ be the total amount of time demand generated by jobs of τ_j after a job J_i of τ_i enters the critical mode and before J_i finishes execution at criticality level ζ_m.

We then define the interference function $\Delta_i^n(\zeta_m, \Gamma, t)$ ($\Delta_i^c(\zeta_m, \Gamma, t)$), which represents the maximum amount of time demand generated by a task set $\Gamma \subset \Gamma_i^n$ ($\Gamma \subset \Gamma_i^c = L_i^{\zeta > \zeta_i} \cup H_i^{\zeta \geq \zeta_i}$) at criticality level ζ_m during an interval of t time units after the release of a job from τ_i when the job is in the normal (critical) mode. More formally,

$$\Delta_i^n(\zeta_m, \Gamma, t) \equiv \sum_{\tau_j \in \Gamma} \delta_i^n(\zeta_m, \tau_j, t),$$
$$\Delta_i^c(\zeta_m, \Gamma, t) \equiv \sum_{\tau_j \in \Gamma} \delta_i^c(\zeta_m, \tau_j, t).$$

For brevity of presentation, we will omit the parameter ζ_m in the time demand and interference functions if the parameter does not change value within an equation.

With such an interference function, we use the time completion function $\mathcal{K}$ to describe the minimum time duration for a job of τ_i to execute for t time units. Within that duration, only tasks in Γ can interfere with τ_i; in addition, the amount of interference from tasks in Γ is governed by Δ_i. The time completion function can be expressed

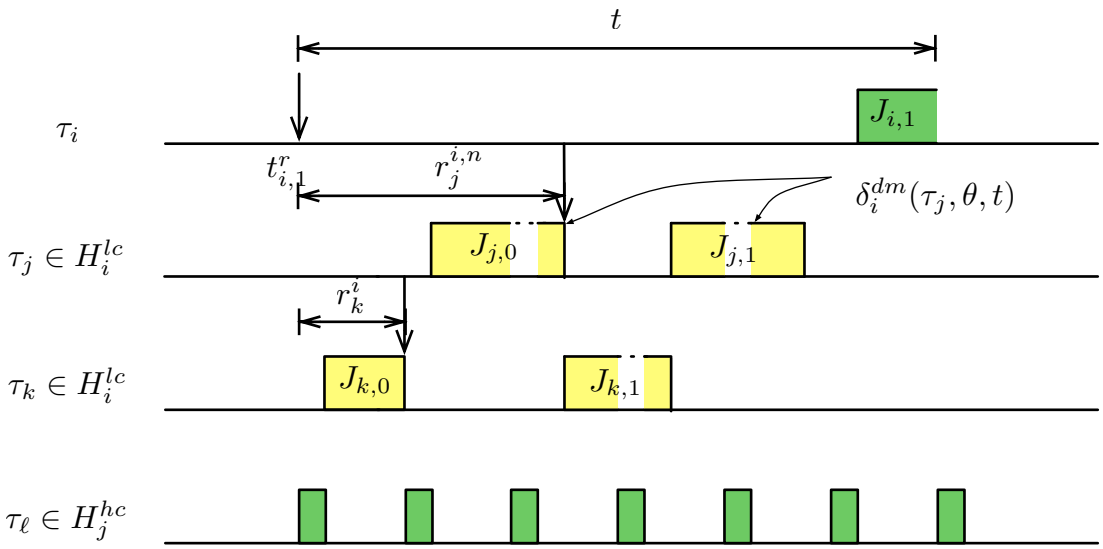


Figure 3.6: Illustration of $r^{i,n}_j$ and $\delta^n_i(\zeta_m, \tau_j, t)$ where both τ_j and τ_k are in $H^{\zeta<\zeta_i}_i$ and $\pi_k > \pi_j$

by

$$\mathcal{K}(t, u, \Gamma, \Delta_i) \equiv \min\left\{\{u\} \cup \{t' \geq t \,|\, t' = t + \Delta_i(\Gamma, t)\}\right\},$$

where u is an upper bound for the returned result.

3.2.7 Normal Mode Time Demand Function

Given a task τ_i in normal mode and another task $\tau_j \in H^{\zeta\geq\zeta_i}_i$, τ_j can always preempt τ_i. The normal mode time demand function is $\lceil t/T_j \rceil C_j(\zeta_m)$, which is the same as that of fixed-priority scheduling. On the other hand, if $\tau_j \in L^{\zeta>\zeta_i}_i$, τ_j can interfere with τ_i only once because $t \leq D_i \leq D_j$. We then can consolidate the time demand function for both cases as

$$\delta^n_i(\zeta_m, \tau_j, t) = \left\lceil \frac{t}{T_j} \right\rceil I^i_j(\zeta_m), \quad \text{if } \tau_j \in \Gamma^n_i - H^{\zeta<\zeta_i}_i. \tag{3.4}$$

.

To compute the normal mode time demand function when $\tau_j \in H^{\zeta<\zeta_i}_i$, if we know the minimum time r^n_j for $J_{j,0}$ to complete executing $C_j(\zeta_m)$ time units after time 0, we can then use r^i_j to derive the relative phase of $J_{j,1}$ with respect to $J_{i,1}$.

To estimate $r_j^{i,n}$, we need to consider the ordering among the tasks in $H_i^{\zeta<\zeta_i}$. Let us suppose $H_i^{\zeta<\zeta_i} \cap (L_j^{\zeta>\zeta_i} \cup H_j^{\zeta<\zeta_i}) = \emptyset$; that is, for any other task $\tau_k \in H_i^{\zeta<\zeta_i} - \{\tau_j\}$, $\pi_j \leq \pi_k$ if and only if $\zeta_j \leq \zeta_k$. Under this assumption, only the tasks from $H_j^{\zeta\geq\zeta_i}$ can preempt τ_j after time 0 and before τ_i starts execution; moreover, the preemptions can occur whether or not τ_j is in critical mode. Therefore $r_j^{i,n}$ can be expressed as

$$r_j^{i,n}(\zeta_m) = \mathcal{K}(C_j(\zeta_m), D_i, H_j^{\zeta\geq\zeta_i}, \Delta_i^n(\zeta_m)). \tag{3.5}$$

If T_j is not equal to D_j, the completion instant of $J_{j,0}$ must be its deadline. Consequently, the release instant $\phi_j^{i,n}$ of $J_{j,1}$ can be expressed as

$$\phi_j^{i,n}(\zeta_m) = r_j^n(\zeta_m) + T_j - D_j. \tag{3.6}$$

With $\phi_j^{i,n}$, the time demand $\delta_i^n(\zeta_m, \tau_j, t)$ for all $\tau_j \in H_i^{\zeta<\zeta_i}$ can then be expressed as

$$\delta_i^n(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi_j^{i,n}(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I_j^i(\zeta_m). \tag{3.7}$$

3.2.8 A Four-Task Example

	C^n	C^o	T	D	criticality
τ_1	1	2	5	5	5
τ_2	2	2	12	10	1
τ_3	3	4	19	19	2
τ_4	4	7	28	28	4

Table 3.4: Example 4-task set for ZSI calculation

However, if $H_i^{\zeta<\zeta_i} \cap (H_j^{\zeta<\zeta_i} \cup L_j^{\zeta>\zeta_i}) \neq \emptyset$, the scenario can be complicated. To see how, let us illustrate it from the perspective of task τ_4 from the four-task example shown in Table 3.4.

Since $\tau_1 \in H_4^{\zeta>\zeta_i}$ can preempt τ_4 at any time, the demand function is then $\delta_4^{dm}(\tau_1, \theta, t) = \lceil t/T_1 \rceil C_1$. For τ_3 and τ_2, which are both in $H_4^{\zeta<\zeta_i}$, we need to find the worst case

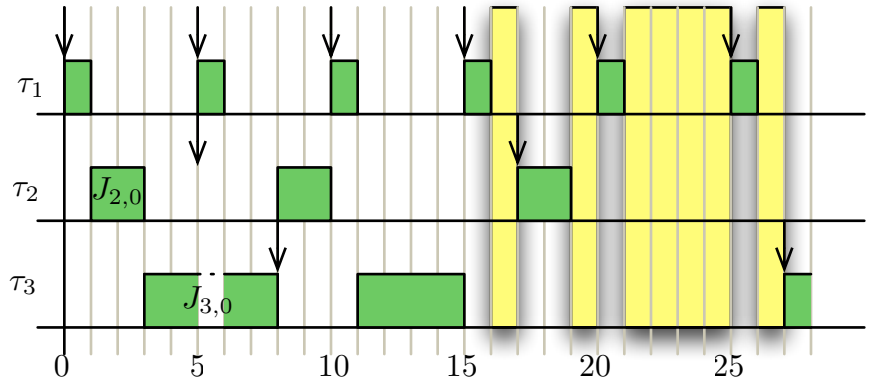
phasing for those two tasks. Because $\pi_2 > \pi_3$ and $\zeta_2 < \zeta_3$, there is no total ordering among the two, and it is difficult to determine which job will finish first. If we assume $J_{3,0}$ won't switch mode before $J_{2,0}$ completes, we can get $r_2^4 = 3$ and $\phi_2^4 = T_2 - D_2 - r_2^4 = 5$. In addition, if $J_{3,0}$ switches mode immediately after $J_{2,0}$ finishes, we can get $r_3^4 = \phi_3^4 = 8$. Figure 3.7a shows the schedule with the above assumptions. On the other hand, Figure 3.7b and 3.7c shows two different scenarios when $J_{3,0}$ starts before $J_{2,0}$.

From Figure 3.7a, 3.7b and 3.7c, we can see that the available slack for τ_4 is different in those scenarios. There is no guarantee in which scenario τ_i can suffer more from the combined interference of τ_2 and τ_3 at a given time. For example, at time 19, the scenario in Figure 3.7a has a smaller amount of empty slack than in Figure 3.7a while the situation reverses at time 25.

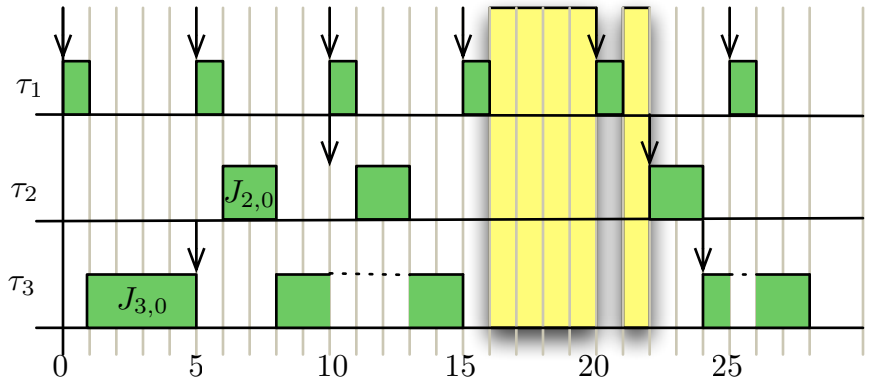
During the calculation of Z_4, the value of Z_3 can't be fixed because Z_3 is also dependent on Z_4. In this case, we can only choose the most conservative approach. Let $\Delta_4^{n,1}(\zeta_4)$ be the interference function when $\phi_2^{4,n} = 5$ and $\phi_3^{4,n} = 8$, $\Delta_4^{n,2}(\zeta_4)$ be the interference function when $\phi_2^{4,n} = 10$ and $\phi_3^{4,n} = 5$. The final interference function for τ_4 would be $\Delta_4^n(\zeta_4) = \max\{\Delta_4^{n,1}, \Delta_4^{n,2}\}$.

In general, the interference function of a task τ_i is a maximum of the functions from all possible orderings of its interference tasks. If the priority/criticality levels for the task set in $H_i^{\zeta<\zeta_i}$ are completely reversed, the computation time would grow exponentially as the set in $H_i^{\zeta<\zeta_i}$ grows.

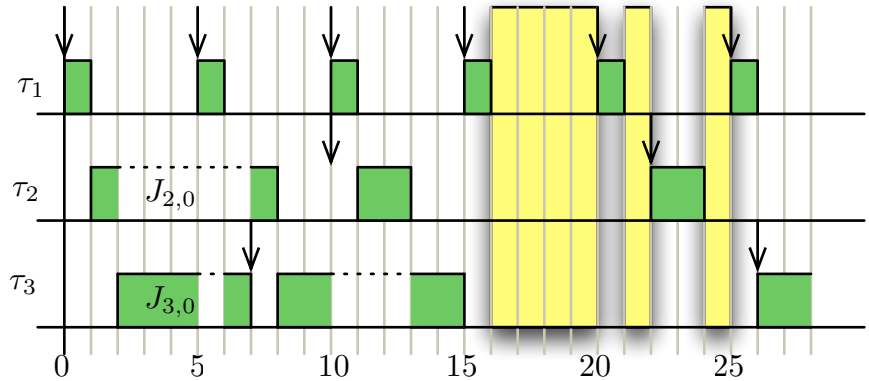
Even though (especially for complex task sets) the exact bound may be intractable to compute, a conservative looser bond is possible. Based on Equation 3.9, the interference grows as the phasing decreases; therefore, we can use the smallest possible phasing interval for all tasks in $H_i^{\zeta<\zeta_i}$ to bound the worst case interference. Looking back to the previous example, $\phi_2^{4,n} = \phi_3^{4,n} = 5$ is used for Δ_4^{θ}. If the interference from $H_j^{\zeta<\zeta_i}$ and $H_j^{\zeta>\zeta_i}$ is not considered, only tasks from $H_j^{\zeta\geq\zeta_i}$ are left in the interference set. Thus we can bound $r_j^{i,n}$ as $r_j^{i,n} = \mathcal{K}(C_j, D_i, H_j^{\zeta\geq\zeta_i}, \Delta_i^{\theta})$, which is exactly the same as Equation 3.5.



(a) $J_{2,0}$ starts/finishes before $J_{3,0}$



(b) $J_{3,0}$ starts/finishes before $J_{2,0}$



(c) $J_{2,0}$ starts before $J_{3,0}$ but finishes after $J_{3,0}$

Figure 3.7: Illustration of worst phasing schedule for the tasks set in Table 3.4

With the availability of Δ_i^n, we can now compute the available slack in normal mode as follows:

$$\theta_i(\zeta_m) = \max(Z_i - \Delta_i(\zeta_m, \Gamma_i^n, Z_i), 0).$$

3.2.9 Critical Mode Time Demand Function

When a task τ_i is in critical mode, only the tasks $\Gamma_i^c \equiv H_i^{\zeta \geq \zeta_i} \cup L_i^{\zeta > \zeta_i}$ can interfere with τ_i. Since the tasks from $H_i^{\zeta < \zeta_i}$ cannot interfere with τ_i, the critical mode time demand function is then

$$\delta_i^c(\zeta_m, \tau_j, t) = \left\lceil \frac{t}{T_j} \right\rceil I_j^i(\zeta_m), \tag{3.8}$$

which was also assumed by de Niz et al. Unfortunately, this is correct only when $\tau_j \notin \{\tau_k \in H_i^{\zeta > \zeta_i} \mid H_k^{\zeta < \zeta_i} \cap H_i^{\zeta < \zeta_i} \neq \emptyset\}$. To see why, let us consider τ_2 from the task set in Table 3.5.

Table 3.5: An example task set where the calculation of Z_2 is based on Equation 3.8

	C^n	C^o	T	ζ	Z
τ_1	2	4	10	3	6
τ_2	3	4	12	2	6
τ_3	2	5	8	1	0

If we use Equation 3.8 to calculate the interference of τ_1 with τ_2 in critical mode, there would be a slack of 8 time units available for τ_2 for every interval of 10 time units.

As shown in Figure 3.8, if we assume that the jobs $J_{1,1}$, $J_{2,1}$ and $J_{3,1}$ are all released at time 0 and both $J_{2,1}$ and $J_{3,1}$ run for their respective overload budget, then the jobs of τ_1 run for only 2 time units. Based on zero-slack scheduling, $J_{1,1}$ and $J_{1,2}$ would be scheduled in the time slots $(5, 7)$ and $(10, 12)$. When $J_{2,1}$ enters critical mode at time 6, only 3 time units are left for τ_2 between the time interval (6,12), which is less than what is given by Equation 3.8 (i.e., 8 time units are available within an interval of 10 time units). The reason for the above phenomenon is that τ_2 suspends the lower criticality task τ_3 in critical mode and thus shortens the inter-job arrival times of τ_1.

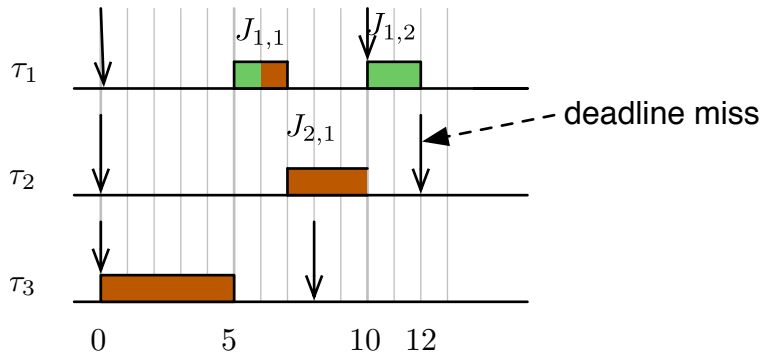


Figure 3.8: A schedule for the task set in Table 3.5 which shows ZSI calculation based on Equation 3.8 can cause τ_2 to miss deadline

Let $H_i^{hc-} \equiv \{\tau_j \in H_i^{\zeta>\zeta_i} \mid H_j^{\zeta<\zeta_i} \cap H_i^{\zeta<\zeta_i} \neq \emptyset\}$ be the task set in $H_i^{\zeta>\zeta_i}$ whose inter-job arrival time could be shortened by suspending the tasks in $H_i^{\zeta<\zeta_i}$. Similar to the calculation of $\phi_i^{j,n}$, to estimate the maximum interference from a task $\tau_j \in H_i^{hc-}$ with τ_i in critical mode, we have to minimize the worst case phasing ϕ_j^c in critical mode. Therefore, we assume τ_j starts execution at the zero slack instant of τ_i. Assuming Z_j is known, we can obtain the maximum response time $r_j^c(\zeta_m)$ of τ_j at criticality level ζ_m based on whether $\theta_j(\zeta_m) \geq C_j(\zeta_m)$. If $\theta_j(\zeta_m) \geq C_j(\zeta_m)$, we need only to count the time for τ_j to run for $C_j(\zeta_m)$ time units in normal mode; i.e.,

$$r_j^c(\zeta_m) = \mathcal{K}(C_j(\zeta_m), Z_j, \Gamma_j^n, \Delta_j^n(\zeta_m))$$

if $\theta_j(\zeta_m) \geq C_j(\zeta_m)$. Otherwise, the response time is Z_j plus the time for τ_j to run for $C_j(\zeta_m) - \theta_j(\zeta_i)$ time units in critical mode; i.e.,

$$r_j^c(\zeta_m) = Z_j + \mathcal{K}(C_j(\zeta_m) - \theta_j(\zeta_i), D_j - Z_j, \Gamma_j^c, \Delta_j^c(\zeta_m))$$

if $\theta_j(\zeta_m) < C_j(\zeta_m)$.

Because we assume τ_j starts execution at its zero slack instant, we can then obtain the phasing of τ_j by

$$\phi_j^c(\zeta_m) = C_j(\zeta_m) + T_j - r_j^c(\zeta_m).$$

Figure 3.9 illustrates the calculation of $\phi_1^c(2)$ for the example in Table 3.5. Since τ_2 does not interfere with τ_1, we can easily see the worst case response time r_1^c of τ_1 is 7. Because Z_2 is not known at the time, we can only assume Z_2 aligns to the

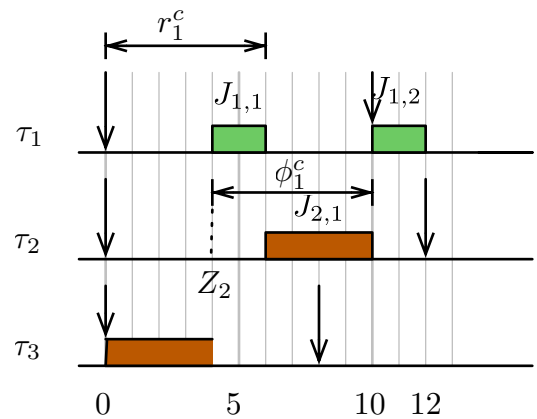


Figure 3.9: The illustration of ϕ_j^c using the example task set in Table 3.5

time when $J_{1,1}$ starts execution so that we can minimize ϕ_1^c. Consequently, $\phi_1^c = C_1(2) + T_1 - r_1^c(2) = 5$. With the phasing estimation, we can obtain the maximum critical mode time demand function for τ_i when $\tau_j \in H_i^{hc-}$ as f ollows:

$$\delta_i^c(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi_j^c(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I_j^i(\zeta_m). \tag{3.9}$$

3.2.10 Available Slack Function

Given that the interference function returns the maximum amount of interference a task τ_i can suffer, we can use it to compute the minimum amount of slack available for a job $J_{i,1}$ of τ_i in a time interval. Intuitively, $t - \Delta_i^n(\zeta_i, \Gamma_i^n, t)$ should be the time available for τ_i in interval t. However, a job released at time t' cannot reclaim the empty slack available before t'. Thus, the amount of empty slack before t should be expressed as

$$\max\{t' - \Delta_i^n(\zeta_i, \Gamma_i^n, t') \mid \forall t' \leq t\}.$$

Furthermore, we are interested in the empty slack which starts no later than t; therefore, we define the available slack function with respect to τ_i as

$$\begin{aligned} S_i^n(t) = \max\{&t' - \Delta_i^n(\zeta_i, \Gamma_i^n, t') \mid (\forall t' < t) \cup \\ &(\forall t' \geq t \text{ where } \Delta_i^n(\zeta_i, \Gamma_i^n, t') = \Delta_i^n(\zeta_i, \Gamma_i^n, t))\}. \end{aligned} \tag{3.10}$$

Figure 3.10 illustrates the relationship between $t - \Delta_4^n(\zeta_4, \Gamma_4^n, t)$ and $S_4^n(t)$, using τ_4 from the task set in Table 3.4.

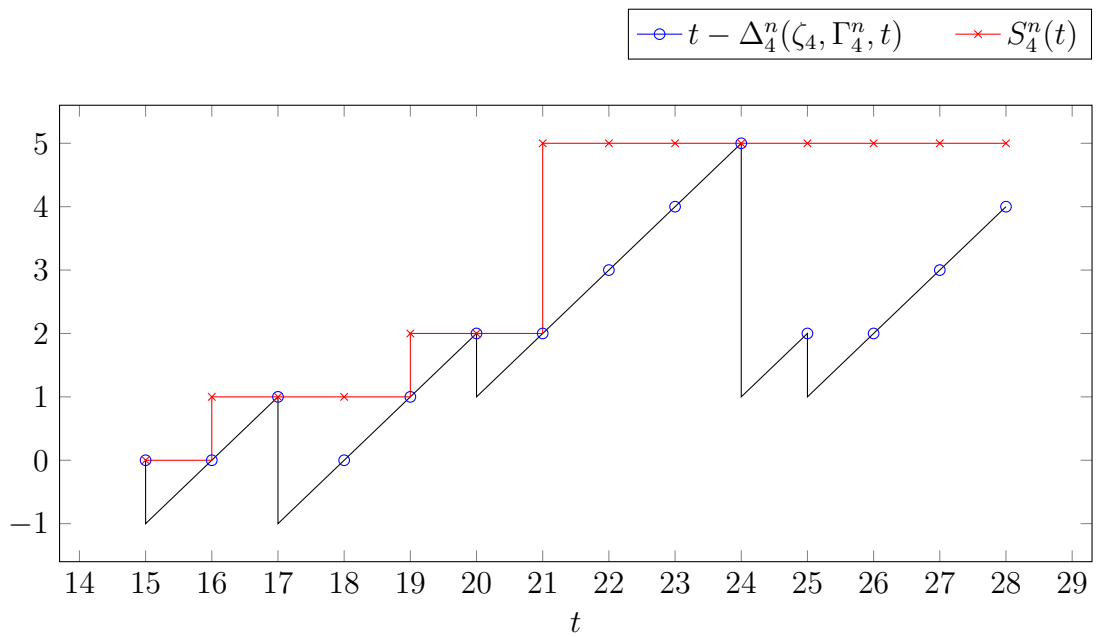


Figure 3.10: The relationship between the slack function $S_4^n(t)$ and $t - \Delta_4^n(\zeta_4, \Gamma_4^n, t)$ for τ_4 in Table 3.4

In [39], de Niz et al. used a slack vector and a procedure `SlackUpToInstant` to calculate the amount of slack available. The slack vector is a sequence of slack regions ordered by time, where each slack region contains a starting instant and duration. Conceptually, the slack vector used by de Niz et al. is an alternative way to express the interference function where $\delta_i^n(\zeta_m, \tau_j, t) = \lceil t/T_j \rceil \, I_j^i(\zeta_m)$ for all $\tau_j \in \Gamma_i^n$. In contrast, our approach requires different time demand functions for the case of $\tau_j \in \Gamma_i^n - H_i^{\zeta<\zeta_i}$ and$\tau_j \in H_i^{\zeta<\zeta_i}$, as shown in Equations 3.7 and 3.9 respectively. In other words, the original algorithm for slack vector calculation does not correctly deal with the interference from tasks in $H_i^{\zeta<\zeta_i}$.

In addition, `SlackUpToInstant` returns the amount of slack available up to the specified instant, whereas S_i^θ returns the amount of empty slack which starts no later than the instant. That tweak allows us to discover more units of slack, as we will demonstrate after we present the improved ZSI calculation algorithm.

3.2.11 Improved ZSI Calculation Algorithm

With the interference and available slack functions, it is straightforward to compute the ZSI of a task as shown in Algorithm 1.

Algorithm 1 GetSlackZeroInstant(τ_i) : Calculate Zero-Slack Instant of τ_i

1: $x \leftarrow 0$
2: **repeat**
3: $\quad x' \leftarrow x$
4: $\quad C_i^c \leftarrow \max(C_i^o - x\ ,\ 0)$
5: $\quad k \leftarrow \mathcal{K}(C_i^c, D_i, \Gamma_i^c, \Delta_i^c(\zeta_i))$
6: $\quad Z_i \leftarrow \max(D_i - k\ ,\ 0)$
7: $\quad x \leftarrow S_i^n(\Gamma_i^n, Z_i)$
8: **until** $x = x'$ or $Z_i = D_i$
9: **return** Z_i

Once again, we use τ_4 from Table 3.4 to illustrate the algorithm and demonstrate how S_i^n improves on `SlackUpToInstant` in [39]. After x is initialized to 0, the algorithm finds $k = 9$ time units is needed for τ_4 to run $C_4^o = 7$ time units in critical mode (Step 5). Then it sets Z_i as $D_4 - k = 19$ which represents the instant when τ_4 should switch mode so that 9 time units is available in its critical mode before deadline. Next, $S_i^n(19) = 2$ in step 7 gives the amount of empty slack available to τ_4. Then τ_4 only needs to reserve $C_4^o - x = 5$ time units in its critical mode (step 4). Following the same process, we can obtain $k = 7$ and $Z_i = 21$ from steps 5 and 6. The next step, $x = S_i^n(21) = 5$. Had `SlackUpToInstant()` been used, θ_i would be 2 and the algorithm would stop at $Z_i = 21$. With $x = 5$, we can push Z_4 further toward its deadline, and then Z_4 becomes 25 when the algorithm stops.

Theorem 2. *Given a task τ_i, The calculation of Z_i using Algorithm 1 is dependent on Z_j of another task τ_j only when $\zeta_j > \zeta_i$.*

Proof. From Algorithm 1, it is clear that the ZSI computation for a task would depend on the result of ZSI computation for other tasks because of the interference functions Δ_i^n and Δ_i^c which in turn depend on the time demand functions δ_i^n and δ_i^c. From Equations 3.3 and 3.4, δ_i^n is dependent on θ_j only if $\tau_j \in L_i^{\zeta>\zeta_i}$. Similarly, δ_i^c is dependent on θ_j only if $\tau_j \in L_i^{\zeta>\zeta_i} \cup H_i^{hc-}$. Because the calculation of θ_j requires Z_j, we can conclude Z_i is dependent on Z_j only when $\zeta_j > \zeta_i$. □

Based on Theorem 2, we can safely calculate the ZSIs of a task set in decreasing criticality order. In [39], ZSIs are computed in an unspecified order and then the algorithm keeps looping until all ZSIs of a task set are stabilized. Our algorithm improves on [39] by computing ZSIs in a deterministic order to avoid unnecessary computation and is unconditionally guaranteed to terminate on any finite task set.

3.3 Fixed Task Priority AMC Scheduling (FTP-AMC)

In zero slack scheduling, the priority assignment is based on task rate and the time a task switches mode (i.e., the zero slack instant) is relative to the release time of a job. In adaptive mixed criticality (AMC), on the other hand, priority assignment is based on Audsley's algorithm and the mode change is triggered by the execution progress of tasks. Baruah et al. [19] discussed two versions of response time analysis of FTP-AMC with only two criticality levels. In this section, we further extend those analyses to systems with more than two criticality levels.

3.3.1 Basic Analysis

Given a task τ_i in a task set Γ,

$$H_i^{\zeta=m} \equiv \{\tau_j \in \Gamma \mid \pi_j \geq \pi_i \text{ and } \zeta_j = m\}$$

$$H_i^{\zeta\geq m} \equiv \{\tau_j \in \Gamma \mid \pi_j \geq \pi_i \text{ and } \zeta_j \geq m\}$$

Assume the deadline of a task does not exceed its period and each task τ_i only executes for $C_i(0)$. The response time of τ_i would be the fixed point of

$$R_i(0) = C_i(0) + \sum_{\tau_j \in H_i} \left\lceil \frac{R_i(0)}{T_j} \right\rceil C_j(0).$$

Given a task $\tau_j \in H_i^{\zeta=0}$, the maximum number of job releases in a busy period would be bounded by $\left\lceil \frac{R_i(0)}{T_j} \right\rceil$. Therefore, we can bound the response time of $R_i(1)$ for which

the system wide criticality level indicator never exceeds 1 by the equation

$$R_i(1) = C_i(1) + \sum_{\tau_j \in H_i^{\zeta=0}} \left\lceil \frac{R_i(0)}{T_j} \right\rceil C_j(0) + \sum_{\tau_j \in H_i^{\zeta=1}} \left\lceil \frac{R_i(1)}{T_j} \right\rceil C_j(1).$$

By induction, we can easily bound the response time $R_i(m)$ of τ_i when the criticality level indicator never exceeds m by

$$R_i(m) = C_i(m) + \sum_{\ell=0}^{m} \left(\sum_{\tau_j \in H_i^{\zeta=\ell}} \left\lceil \frac{R_i(\ell)}{T_j} \right\rceil C_j(\ell) \right).$$

Thus the response time R_i^* of τ_i can be expressed by

$$R_i^* = R_i(\zeta_i).$$

However, this bound is pretty loose because it assumes every job $\tau_{i,j}$ executes for $C_i(\zeta_i)$ which is different from the scheduling algorithm we described.

3.3.2 Improved Analysis

To overcome the overestimation described in basic analysis, Baruah et al. [19] developed another analysis which they described as AMC-max. However, that approach is hard to extend for more than 2 criticality levels. Therefore, we develop another analysis which is based on AMC-max with certain modification.

To simply our discussion, we begin with a system with only two criticality levels (0 and 1). Let s to be the last job deadline before criticality change[5]; i.e., all jobs $J_{i,j}$ whose deadlines are before s can only executes $C_i(0)$. The number of job releases for

[5]In [19], s is initially defined as the time point when the system criticality level indicator changes from 0 to 1. However, in the latter part of the paper, it assumes all jobs released before and at s only execute in low criticality mode.

a low criticality task τ_j is thus bound by

$$n_j(s) = \left\lceil \frac{s}{T_j} \right\rceil .$$

Since any job of a higher criticality task τ_k whose deadline is before s can executed only for $C_k(0)$, we can bounded the number $n_k(s)$ of job releases for τ_k that executes in low criticality mode by

$$n_k(s) = \max\left(\left\lfloor \frac{s - D_k}{T_k} \right\rfloor + 1, 0 \right).$$

We use $floor + 1$ instead of ceiling because the job of τ_k which releases at exactly $s - D_k$ can only be executed for $C_k(0)$ based on our assumption of s, and thus it should be counted in $n_k(s)$. Given a task $\tau_k \in H_i^{\zeta \geq \zeta_i}$, the maximum number of job releases in high criticality mode within a busy period of R is thus bounded by $\left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - n_k(s) \right)$. Therefore, the response time R_i^s of τ_i is the fixed point of

$$R_i^s = C_i(0) + \sum_{\tau_j \in H_i} n_j(s) C_j(0) + \sum_{\tau_k \in H_i^{\zeta \geq \zeta_i}} \left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - n_k(s) \right) C_k(1).$$

Since s represents the deadline of a job; the release time of the job is bounded by $R_i(0)$, we can then enumerate all possible values of s to obtain the maximum response time of τ_i by

$$R_i^* = \max\{R_i^s \mid \forall \tau_j, \quad s \in [D_j, \ldots, \left\lceil \frac{R_i(0)}{T_j} \right\rceil T_j + D_j]\}.$$

Consider an example task set comprised of 3 tasks as follows

	ζ_i	$C_i(0)$	$C_i(1)$	D_i	T_i
τ_1	0	1	1	2	2
τ_2	1	1	5	10	10
τ_3	1	20	20	100	100

Suppose there is no criticality change: the response time of τ_3 could be computed by

$$R_3(0) = 20 + \left\lceil \frac{R_3(0)}{2} \right\rceil + \left\lceil \frac{R_3(0)}{10} \right\rceil,$$

which has solution 50. The deadlines of jobs between (0, 50] are the even numbers between 2 and 50. Each of these values for s needs to be checked. The worst case occurs when $s = 48$.

$$R_3^{48}(1) = 20 + \left\lceil \frac{48}{2} \right\rceil + \left(\left\lfloor \frac{48-10}{10} \right\rfloor + 1 \right) + \left(\left\lceil \frac{R_3^{48}(1)}{10} \right\rceil - \left(\left\lfloor \frac{48-10}{10} \right\rfloor + 1 \right) \right) 5,$$

which has solution 58.

Now, consider the case where the number of criticality levels of the system is greater than 2. Let s_ℓ, referred as the *criticality changing point*, be the last job deadline before the system criticality level indicator changes from $\ell - 1$ to ℓ where s_0 is defined as 0. Let $\mathbb{S} = \{s_0, s_1, \dots\}$ be a sequence of criticality changing points for a busy period of a mixed-criticality task schedule. Let $n_j^\ell(\mathbb{S})$ be the number of τ_j job releases which the jobs execute at criticality level no larger than ℓ. The number of job releases within the criticality level ℓ is thus

$$n_j^\ell(\mathbb{S}) = \begin{cases} 0 & \text{if } \ell < 0, \\ n_j^{\ell-1} & \text{else if } \zeta_j < \ell, \\ \left\lceil \frac{S_\ell}{T_j} \right\rceil & \text{else if } \zeta_j = \ell, \\ \max\left(\left\lfloor \frac{s_\ell - D_j}{T_j} \right\rfloor + 1, 0 \right) & \text{otherwise.} \end{cases}$$

Let $I^{\mathbb{S}}(t, m, \Gamma)$ be the time demand from a task set Γ within a busy interval of t, given that the system criticality change is governed by $\mathbb{S}$ and the system criticality level indicator does not exceed m in (or before) that interval. $I^{\mathbb{S}}(t, m, \Gamma)$ can be represented

by

$$I^{\mathbb{S}}(t,m,\Gamma) = \sum_{\ell=0}^{m-1} \left\{ \sum_{\tau_j \in \Gamma} (n_j^{\ell}(\mathbb{S}) - n_j^{\ell-1}(\mathbb{S}))C_j(\ell) \right\} + \sum_{\tau_k \in \Gamma^{\zeta \geq m}} \left(\left\lceil \frac{R_i^{\mathbb{S}}(m)}{T_k} \right\rceil - n_k^{m-1}(\mathbb{S}) \right) C_k(m), \tag{3.11}$$

where $\Gamma^{\zeta \geq m} = \{\tau_j \in \Gamma \mid \zeta_j \geq m\}$.

Assume that $D_i \leq T_i$, the response time $R_i^S(m)$ of τ_i if the system criticality level indicator is no larger than m is thus

$$R_i^{\mathbb{S}}(m) = C_i(m) + I^{\mathbb{S}}(R_i^{\mathbb{S}}(m), m, H_i).$$

The relationships of the elements in $\mathbb{S}$ can be represented by

$$s_\ell \in (s_{\ell-1}, R_i^{\mathbb{S}}(\ell-1) + D_{max}),$$

where D_{max} represents the maximum relative deadline for the tasks in H_i. Then the maximum response time of τ_i is

$$R_i^* = \max\{R_i^{\mathbb{S}}(\zeta_i) \mid \forall\, \mathbb{S}\}.$$

3.4 Fixed Job Priority AMC Scheduling (FJP-AMC)

In [20], Baruah et al. developed an algorithm, called own criticality based priority (OCBP), for mixed-criticality systems comprised of a finite number of (non-recurring) jobs with an AMC scheduling policy. The job priority assignment algorithm is also based on Audsley's approach. To be more precise, within each step, the algorithm selects a job J_i whose deadline is longer than the total time demand for the remaining jobs $\mathbb{J}$ and assigns J_i the lowest priority among $\mathbb{J}$. Next, J_i is removed from $\mathbb{J}$ and the process continues until no job is available for priority assignment.

Li [66] further developed an on-line job priority assignment algorithm for scheduling sporadic tasks based on OCBP. The algorithm starts by assuming all tasks are released at the same time and assigns job priorities according to OCBP. Whenever a new job $J_{i,j}$ arrives, the on-line scheduler compares the previously assigned priority $\zeta_{i,j}$ of $J_{i,j}$ and the priority $\zeta_{k,\ell}$ of the current running job $\tau_{k,\ell}$. If $\zeta_{i,j} > \zeta_{k,\ell}$, the priorities of all jobs in $\{J_{i,j} \mid \zeta_{i,j} > \zeta_{k,\ell}\}$ are re-computed using OCBP based on the worst possible job arrival pattern starting from the $\tau_{k,\ell}$ arrival time. Li also proved that no priority re-computation is needed if the newly arrived job $\tau_{k,\ell}$ has lower priority than the current running job. However, the complexity of the on-line priority re-computation is pseudo-polynomial in the worst case. This makes the algorithm less acceptable for systems which require stringent worst-case bounds.

To reduce the run-time complexity of Li's algorithm, Guan [50] presented an algorithm called PLRS (Priority List Reuse Scheduling). Instead of solely relying on the high complexity on-line priority re-computation for job scheduling, PLRS separates the algorithm into two stages: an off-line priority computation and an on-line priority assignment stage. The PLRS off-line priority computation is essentially the same as the on-line priority re-computation of Li's method, but PLRS only computes jobs of the worst case busy period when all tasks are released at the same time. The resulting information is then used by a lighter weight on-line algorithm for priority assignment. The complexity of the off-line stage is still pseudo-polynomial but that of the on-line stage is reduced to be quadratic in the number of tasks.

However, both Li's and Guan's algorithm require a worst case job arrival pattern for job priority calculation; i.e., how many jobs (along with execution criticalities) of each task can be executed in a busy period. To obtain the job arrival pattern, the longest busy period and the distribution with each criticality level needs to be bounded. Li [66] offered a busy period bound based on the *load*[6] of each criticality level. However, in our experiments, we found the bound is too loose to be useful, and so it is essential to have a tighter busy period bound for FJP-AMC based algorithms.

We therefore develop a new analysis to find a job arrival pattern for FJP-AMC job priority calculation. Like the FTP-AMC analysis, given a task set Γ, the longest busy

[6]The load of a collection of jobs denotes the maximum execution requirements over all time intervals, normalized by the interval length.

period without any criticality change would be the fixed point of

$$B(0) = \sum_{\tau_i \in \Gamma} \left\lceil \frac{B(0)}{T_i} \right\rceil C_i(0).$$

We can also express it in terms of $I^{\mathbb{S}}$ from Equation 3.11 by

$$B(0) = I^{\{0\}}\left(B(0), 0, \Gamma\right),$$

where $n_k^{-1} = 0$ for all k.

For a task τ_i of criticality 0, the number of jobs n_i^0 is bounded by $\lceil B(0)/T_i \rceil$. To extend the busy period, the system criticality level indicator cannot be changed before the last job $J_{i,j}^{\zeta=0}$ of criticality 0 finishes. The earliest time for $J_{i,j}^{\zeta=0}$ to finish is

$$s^1 = \max_{\tau_i \in \Gamma^{\zeta=0}} \left\{ \left\lfloor \frac{B(0)}{T_i} \right\rfloor T_i + C_i(0) \right\}.$$

Assuming any job $J_{i,j}$ released before s_1 can only execute for $C_i(0)$, then the maximum number n_i^0 of jobs for τ_i to be executed for $C_i(0)$ in a busy period is

$$n_i^0 = \left\lceil \frac{s^1}{T_i} \right\rceil.$$

To generalize to a task set with m criticality levels, we get

$$\begin{aligned}
B(m) &= I^{\mathbb{S}}\left(B(m), s^m, \Gamma\right), \\
s^m &= \max_{\tau_i \in \Gamma^{\zeta=m-1}} \left\{ \left\lfloor \frac{B(m-1)}{T_i} \right\rfloor T_i + C_i(m-1) \right\}, \\
n_i^m &= \begin{cases} 0 & \text{if } m < 0, \\ n_i^{m-1} & \text{if } \zeta_i < m, \\ \left\lceil \frac{s^m}{T_i} \right\rceil & \text{if } \zeta_i \geq m. \end{cases}
\end{aligned}$$

Consider an example comprised of 4 tasks,

	ζ_i	$C_i(0)$	$C_i(1)$	D_i	T_i
τ_1	1	4	6	10	10
τ_2	1	3	5	20	20
τ_3	0	6	6	30	30
τ_4	0	2	2	15	15

Assuming that there is no criticality change, the criticality 0 busy period would be

$$B(0) = \left\lceil \frac{B(0)}{10} \right\rceil 4 + \left\lceil \frac{B(0)}{20} \right\rceil 3 + \left\lceil \frac{B(0)}{30} \right\rceil 6 + \left\lceil \frac{B(0)}{15} \right\rceil 2,$$

which has solution 28. That is, for any busy period of the task set, τ_3 and τ_4 can release at most 1 and 2 jobs, respectively; i.e., $n_3^0 = 1$ and n_4^0=2. For a busy period in which τ_3 and τ_4 release exactly 1 and 2 jobs, the system criticality level indicator cannot change from 0 to 1 before 17 time units after the beginning of the busy period because 17 is the earliest possible time for the second job of τ_4 to finish.

For the high criticality task τ_1, its first job $J_{1,1}$ should execute only for $C_1(0)$ in order to maximize the busy period because its deadline is before 17. The question is whether $J_{1,2}$ and $J_{2,1}$ should execute for their worst case execution times $C_1(1)$ and $C_2(1)$.

Assume those two jobs run only for $C_1(0)$ and $C_2(0)$. Then, the number of jobs for τ_1 and τ_2 to run in confidence level 0 are 2 and 1, respectively; i.e., $n_1^0 = 2$ and $n_2^0 = 1$. Thus we can calculate the busy period when τ_1 and τ_2 execute for their worst case execution time after their second and first job, respectively.

$$B(1) = 2 \times 4 + 3 + 6 + 2 \times 2 + \left(\left\lceil \frac{B(1)}{10} \right\rceil - 2 \right) 6 + \left(\left\lceil \frac{B(1)}{20} \right\rceil - 1 \right) 5,$$

where the solution of $B(1)$ is 38. Consequently, $n_1^1 = \lceil 38/10 \rceil = 4$ and $n_2^1 = \lceil 38/20 \rceil = 2$.

With the job arrival pattern, we use OCBP to assign job priorities as follows,

τ_1	7,5,2,0
τ_2	6,1
τ_3	3
τ_4	8,4

Now let us consider the case where $J_{1,2}$ and/or $J_{2,1}$ execute for their worst case execution time under the above job priority assignment. Since both $J_{1,2}$ and $J_{2,1}$ have higher priority (5 and 6) than $J_{3,1}$ and $J_{4,2}$ (3 and 4), both $J_{3,1}$ and $J_{4,2}$ would be aborted or suspended at the moment $J_{1,2}$ and $J_{2,1}$ executes up to their low criticality execution time and the system is still considered schedulable.

On the other hand, if we assume $J_{1,2}$ and/or $J_{2,1}$ can execute for their worst case execution time; i.e., $n_1^1 = \lfloor s^1/T_1 \rfloor = 1$ and $n_2^1 = \lfloor s^1/T_2 \rfloor = 0$, $B(1)$ would become 59. Even though this busy period is longer, the job arrival pattern obtained from it is not FJP-AMC schedulable.

In general, using *floor* instead of *ceiling* for n_i^m when $\zeta_i \geq m$ is not necessary to produce an FJP-AMC un-schedulable job arrival pattern. However, if a job arrival pattern derived from $n_i^m = \lfloor s^m/T_i \rfloor$ is FJP-AMC schedulable, the job arrival pattern derived from $n_i^m = \lceil s^m/T_i \rceil$ for the same task set is always schedulable because of the lesser time demand of the latter. Therefore, using $n_i^m = \lceil s^m/T_i \rceil$ when $\zeta_i \geq m$ is sufficient for schedulability test. However, using *floor* may allow more jobs to be admitted at run-time. For PLRS, where job priorities are computed off-line, it is better to use the floor version unless it produces an un-schedulable job arrival pattern.

3.5 Mixed-Criticality End-to-End Task Sets

In the previous sections, we discussed the mixed criticality models and the various techniques to schedule mixed-criticality real-time task sets for uniprocessor systems. In this section, we focus on scheduling mixed-criticality task sets in distributed real-time systems and investigate whether those approaches can perform similarly efficiently when applied to a distributed system where a task may span several processors or hosts with precedence constraints, or whether there are other better alternatives.

Before considering end-to-end scheduling of mixed-criticality systems, we first discuss background on end-to-end scheduling of traditional single-criticality end-to-end tasks. For real-time distributed applications with multiple periodic tasks, it is necessary to map each application to the end-to-end system model. There are four major problems involved in this mapping: task assignment, priority assignment, synchronization protocol and scheduling analysis. In this section, we introduce these problems and briefly describe the strategies to deal with them.

Task Assignment: When the system is static, the application is partitioned into modules which are bound to particular processors [69]. This step is called task assignment. For traditional fixed priority systems, task assignment can be based on one or more (of three) factors: execution time requirements, cost of communication, and placement of resources. Finding a optimal solution is NP hard [69] even when only the execution time requirement is considered. Liu [69] describes several heuristic algorithms for task assignment based on those three factors without considering mixed criticality issues. Lakshmanan [63] provides a heuristic for task allocation of independent mixed criticality task sets that considers only the execution time requirement. In this dissertation, we will assume the task assignment has been fixed (e.g., based on certain heuristic algorithms).

Priority Assignment: When subtasks are to be scheduled on a fixed priority basis, the first issue is assigning priorities to the subtasks. In general, assigning priorities to subtasks so that the system is schedulable is called the *priority assignment problem*. Like the task assignment problem, it is often intractable to find an optimal solution to priority assignment in a large system. Sun [91] describes several heuristic methods for traditional fixed priority end-to-end systems. In Section 3.5.1, we discuss methods that are specifically tailored to mixed-criticality systems.

Synchronization Protocol: The third problem in end-to-end scheduling is concerned with the timing of releases of non-root subtasks of an end-to-end task. This problem arises from the fact that a non-root subtask can only start after the completion of its immediately previous subtask and the completion times of a subtask may not be periodic. Existing fixed priority schedulability analysis is based on the periodic

assumption; however, this assumption is no longer true for the non-root subtasks if a non-root subtask is released immediately after the completion of its previous subtask.

Several synchronization protocols [91, 69] have been developed to deal with this issue. Most of them are based on the idea that a non-root subtask $\tau_{i,j}$ should not be released immediately after its previous subtasks complete; instead, the release time of $\tau_{i,j}$ should be deliberately delayed for some time so that all instances of $\tau_{i,j}$ can be released periodically.

Among those synchronization protocols, the Release Guard Protocol [91] is considered most suitable for real-time systems because of its low complexity and low overhead. With the release guard protocol, every non-root subtask has to keep a periodic release timer. A subtask sends a synchronization signal immediately after its completion to its downstream subtasks and a task can only be released when both the upstream synchronization signal is released and its periodic release timer has expired. Due to the suitability of the release guard protocol in real-time systems, we will adopt it as the basis of our mixed-criticality end-to-end tasks models.

Scheduling Analysis: Once the task/priority assignments have been determined and the synchronization protocol is chosen, the next question is whether all tasks in the system can meet their deadlines. For fixed priority systems, the response time of each subtask can be computed using the Lehoczky approach [64] and the worst case response time of an end-to-end task can be obtained by the summation of response times of the longest path along the task graph. The schedulability of an end-to-end task can then be determined by whether the response time of the task is less than its deadline.

3.5.1 Priority Assignment for Mixed-Criticality End-to-End Tasks

In this subsection, we focus on the problem on how to assign priorities to subtasks so that all end-to-end tasks in a mixed-criticality multiprocessor system can be feasibly scheduled. In general, this is an NP-hard problem since priority assignment for even

a single criticality system is NP-hard [69]. Like single criticality systems, one could use computationally intensive techniques such as simulated annealing [93] for assigning priorities off-line, or we can use heuristic methods [91, 69] for subtask priority assignment on-line, which is our focus here.

Similar to the uniprocessor mixed-criticality systems, we can assign subtask priorities based on the rate or criticality of a task. However, these approaches are generally inefficient as we have shown in the uniprocessor systems. The other approach for the priority assignment problem is to solve it in two steps: deadline-assignment and then priority-assignment. In the deadline-assignment step, a local relative deadline is assigned to each subtask in accordance to the end-to-end deadline. There are many deadline-assignment algorithms [60, 61, 91]. Let $d_{i,j}$ be the local relative deadline of subtask $\tau_{i,j}$. We now characterize four deadline-assignment algorithms with mixed-criticality extensions.

- Ultimate Deadline Algorithm (UD)

$$d_{i,j} = D_i;$$

- Effective Deadline Algorithm (ED)

$$d_{i,j} = D_i - \sum_{k=j+i}^{N_i} C_{i,k}(\zeta_i);$$

- Proportional Deadline Algorithm (PD)

$$d_{i,j} = D_i C_{i,j}(\zeta_i) / \sum_{\forall k} C_{i,k}(\zeta_i);$$

- Normalized Proportional Deadline Algorithm (NPD)

$$d_{i,j} = D_i \frac{C_{i,j}(\zeta_i) U(\tau_{i,j})}{\sum_{\forall k} C_{i,k}(\zeta_i) U(\tau_{i,k})};$$

where N_i is the number of subtasks in task τ_i and $U(\tau_{i,j})$ is the total CPU utilization of all subtasks in the same processor to which subtask $\tau_{i,j}$ is assigned.

Once each subtask has a local relative deadline, the scheduler can then use this intermediate relative deadline as the virtual deadline of the subtask and assign subtask priority in the basis of this virtual deadline. Thus the problem of assigning priorities of end-to-end tasks is reduced to that of assigning priorities to subtasks on each processor. In other words, we can use the uniprocessor schedulability analysis to calculate the response time of each subtask on each processor and the end-to-end response time can be obtained by the summing up the response times of subtasks in each end-to-end task.

Schedulability Analysis for Fixed Task Priority: For traditional (single criticality) end-to-end systems, Deadline Monotonic (DM) scheduling, where higher priorities are assigned to subtasks with shorter deadlines, has been extensively studied[91, 69] in conjunction with the mentioned deadline-assignment algorithms. The major complication about applying the previously mentioned uniprocessor schedulability analysis is that simple fixed task priority response time analysis such as Equation 3.1 is based on the assumption that the deadline is smaller than the period of the task. However, the deadline assignment algorithms we discussed cannot guarantee to maintain this constraint for all subtasks. To overcome this limitation of the response time analysis, a technique called *busy period analysis* [64, 91] is required.

Given a task τ_i in a fixed task priority periodic or sporadic task set, let H_i be the set of tasks with equal or higher priorities than τ_i. The *busy period* of τ_i refers to the time interval (t_b, t_e) in which the processor is idle or none of the tasks in H_i is executing immediately before t_b and after t_e; in addition, only the tasks in H_i can be executed between t_b and t_e and the processor is never idle within the interval. For fixed task priority scheduling of sporadic tasks, it is proved that the longest busy period of a task H_i occurs when the tasks in $H_i \cup \{\tau_i\}$ are released at the same time. Thus the response time of a task τ_i with unconstrained deadline can be obtained by analyzing the finish times of all jobs of τ_i in the longest busy period and then subtract the release times of their respective jobs.

For the analysis of fixed task priority mixed-criticality tasks (i.e., FTP-SMC and FTP-AMC schemes) with unconstrained deadlines, busy period analysis is still applicable because no assumption is violated. In the following, we use the FTP-AMC scheme

to illustrate how busy period analysis works. Consider the case where $D_i > T_i$, the longest busy period $W_i^s(m)$ for τ_i in a given criticality level m is the fixed point of

$$W_i^{\mathbb{S}}(m) = I^{\mathbb{S}}\left(W_i^{\mathbb{S}}(m), m, H_i \cup \{\tau_i\}\right).$$

Let $Z_i^{\mathbb{S}}(k)$ be the criticality level of the k-th job release in the busy period governed by $\mathbb{S}$; i.e., $Z_i^{\mathbb{S}}(k) \equiv \ell$ if $n_i^{\ell-1}(\mathbb{S}) \leq k < n_i^{\ell}(\mathbb{S})$. Let $L_i^{\mathbb{S}}(k)$ be the maximum workload generated by τ_i in a busy period governed by $\mathbb{S}$,

$$L_i^{\mathbb{S}}(k) = \sum_{\ell=0}^{\zeta_i} \left(\min(n_i^{\ell}(\mathbb{S}), k) - \min(n_i^{\ell-1}(\mathbb{S}), k)\right) C_i(\ell).$$

The k-th response time of τ_i is then the fixed point of

$$\rho_i^{\mathbb{S}}(k) = L_i^{\mathbb{S}}(k) + I^{\mathbb{S}}(\rho_i^{\mathbb{S}}(k), Z_i^{\mathbb{S}}(k), H_i).$$

The response time of τ_i is thus

$$\max\left\{\rho_i^{\mathbb{S}}(k) - kT_i \mid \forall \mathbb{S}, \forall k, \text{ where } 1 \leq k \leq \left\lceil \frac{W_i^{\mathbb{S}}(\zeta_i)}{T_i} \right\rceil\right\}. \tag{3.12}$$

Schedulability Analysis for Period Transformation: For mixed-criticality end-to-end tasks, we can still apply the period transformation technique by assigning local relative deadlines using PD or NPD, and then transforming the period T_i of a subtask $\tau_{i,j}$ to $T'_{i,j} = T_i/n_{i,j}$ such that $T'_{i,j} < T'_{m,n}$ for all $\tau_{m,n}$ where $\tau_{m,n}$ is allocated to the same processor as $\tau_{i,j}$ and $\zeta_i > \zeta_m$. In the case of end-to-end tasks, the local relative deadline $d_{i,j}$ of a subtask $\tau_{i,j}$ may be far smaller than its period T_i and thus Equation 3.2 cannot be directly used because it assumes period equals to deadline for every task. To overcome this limitation, the execution time of a subtask cannot be equally partitioned into each transformed period; i.e., let ζ_{max} be the maximum criticality level in the system, the transformation should guarantee $C_{i,j}(\zeta_{max}) / \lceil d_{i,j} n_{i,j} / T_i \rceil$ units of execution time within each transformed period $T'_{i,j} = T_i/n_{i,j}$ so that $\tau_{i,j}$ can finishes execution before its local relative deadline. The transformed deadline $d'_{i,j}$ thus becomes $(d_{i,j} \mod T'_{i,j}) + T'_{i,j}$.

Schedulability Analysis for Fixed Job Priority: The analysis presented in Section 3.3 does not make any assumption about the deadlines of tasks; therefore, it can be directly applied to end-to-end tasks on a processor without any modification.

Schedulability Analysis for Zero-Slack Scheduling: Although zero-slack scheduling is introduced as a viable solution for mixed-criticality scheduling in uniprocessor system, our schedulability evaluation (discussed in Chapter 4) shows it offers little improvement over rate monotonic and performs much worse than other approaches specific to mixed-criticality systems. Therefore, we don't think it worthwhile to extend zero-slack scheduling to end-to-end systems.

Chapter 4

Mixed Criticality Evaluations

In Chapter 3, we have discussed various mixed-criticality scheduling approaches and their schedulability analysis. In this chapter, we start with presenting an evaluation of different approaches to mixed-criticality scheduling in terms of task schedulability on uniprocessor systems. Next, we evaluate those approaches with the modifications tailed for distributed end-to-end tasks. Last, we describe mixed criticality support in MCFlow and an evaluation of enforcement mechanisms for the various mixed-criticality scheduling approaches.

4.1 Schedulability Evaluation for Mixed-Criticality Tasks on Uniprocessor Systems

In this section, we present our investigation of the schedulability of different mixed-criticality scheduling approaches and demonstrate the effectiveness of our improved analyses over their original counterparts.

4.1.1 Task set generation parameters

We studied the schedulability of randomly generated task sets, where each task set Γ_k was generated based on the following parameters:

- The CPU utilization U_k of Γ_k is a number between 0.05 and 0.95. Given a U_k, the utilization u_i of a task τ_i in Γ_k was generated by the UUnifast algorithm [25].
- The period T_i of a task τ_i was $100x$ where x was randomly sampled from a uniform distribution between 1 to 100, and the deadline D_i was the same as the period T_i.
- The criticality levels of tasks were assigned sequentially as tasks were generated; i.e., if the total number of criticality levels in a task set was N, then the criticality level ζ_i of the i-th generated task τ_i was $i \mod N$.
- The execution time $C_i(0)$ of τ_i was obtained from the equation $C_i(0) = \max(\lfloor T_i u_i \rfloor, 1)$.
- The default ratio (CF) between the worst case execution time $C_i(\zeta_i)$ and $C_i(0)$ was 1.5. Given a CF, the execution time specification of τ_i was

$$C_i(\zeta) = \begin{cases} C_i(0) & \text{if } \zeta < \zeta_i, \\ CF \times C_i(0) & \text{otherwise.} \end{cases}$$

- For each set of parameters, 1000 tasks were generated.

4.1.2 Scheduling approaches and analysis investigated

The scheduling approaches and analysis we used are as follows:

- Rate monotonic scheduling (RMS) with response time analysis based on Formula 3.1.
- Criticality monotonic scheduling (CMS) with response time analysis based on Formula 3.1.
- Period transformation (PT) with response time analysis based on Formula 3.2.
- Zero slack scheduling (ZSS) with our improved analysis described in Section 3.2.
- Static Mixed Criticality (FTP-SMC), priority assigned with Audsley's approach with response time analysis based on Formula 3.1.

- FTP-AMC with the first analysis (AMC-RT) developed by Baruah et al. [19] which was also described in Section 3.3.1.
- FTP-AMC with the second analysis (AMC-MAX) developed by Baruah et al. [19].
- FTP-AMC with our improved analysis (AMC-HGL) as described in Section 3.3.2.
- FJP-AMC with job arrival pattern developed by Li [67] (FJP-LB).
- FJP-AMC with our improved job arrival pattern as described in Section 3.4 (FJP-HGL).

4.1.3 Simulation

For convenience of presentation, we will separate our evaluation into three different parts. The first part compares the effectiveness of different FTP-AMC analyses as well as FTP-SMC. The second part evaluates FJP-AMC with two different methods of job arrival pattern calculation. Last, we compare how different scheduling approaches affect the schedulability of mixed-criticality tasks.

FTP-SMC/FTP-AMC Comparison Figure 4.1 plots the result with 20 tasks in each task set (NT=20), 2 levels of criticality (NC=2) and the ratio between $C_i(\zeta_i)$ and $C_i(0)$ is 1.5 (i.e., $CF = C_i(\zeta_i)/C_i(0) = 1.5$). The conducted our experiment with the total utilization of a task set varying from 0.05 to 0.95. However, we only show the utilization ranges from 0.55 to 0.88 because the percentage of schedulable tasks are either 1 or 0 outside of the range, and thus the distinction between these analyses can be more clear.

As shown in Figure 4.1, AMC based analyses performed better in schedulability tests because of the mechanism that lower criticality tasks have to be suspended/aborted when the execution of a higher criticality task exceeds certain execution thresholds. The differences between all 3 AMC analyses are very small; however, AMC-HGL consistently performs better than AMC-MAX and AMC-RT. Beside the marginal schedulability improvement of AMC-HGL over AMC-MAX, AMC-HGL analysis can

be easily extended beyond two levels of criticalities while it is not the case the AMC-MAX.

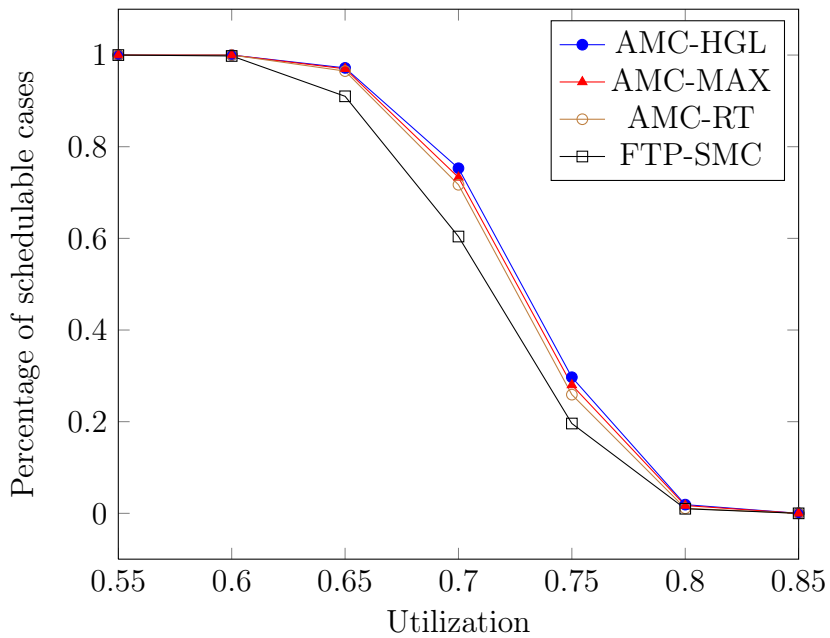


Figure 4.1: Schedulability evaluation for fixed task priority SMC/AMC analyses with 20 tasks (NT=20, CF=1.5, NC=2)

Figures 4.2 and 4.3 further demonstrate the slightly better performance for AMC-HGL over AMC-MAX and AMC-RT in schedulability tests. In both Figures 4.2 and 4.3, we show the weighted schedulability measure [22, 19] over varying number of tasks (NC) and CF ratio, respectively. For each parameter p (NC or CF), the weighted schedulability measure combines results for all the task sets Γ generated for all of a set of equally spaced utilization levels (0.05 to 0.95 in steps of 0.05).

Given a total CPU utilization u_i, let $\pi_i(p)$ be the percentage of schedulable cases (out of 1000) with parameter p. The weighted schedulability measure $W(p)$ is defined as

$$W(p) = (\sum_{\forall i} u_i \pi_i(p)) / \sum_{\forall i} u_i.$$

The weighted schedulability measure reduces what would otherwise be a 3-dimensional representation to 2 dimensions. Weighting the individual schedulability results by task set utilization reflects the higher value placed on being able to schedule higher utilization task sets [22, 19].

Besides the slight performance gain in AMC-HGL, Figure 4.2 also shows all the weighted schedulability curves over the number of tasks are relatively flat which indicates neither the schedulability of AMC nor that of SMC is sensitive to the number of tasks in the system. On the other hand, because the CPU utilization of a generated task τ_i is based on the execution time at the lowest confidence level $C_i(0)$, it is expected that the schedulability decreases as the CF ratio increases as shown in Figure 4.3.

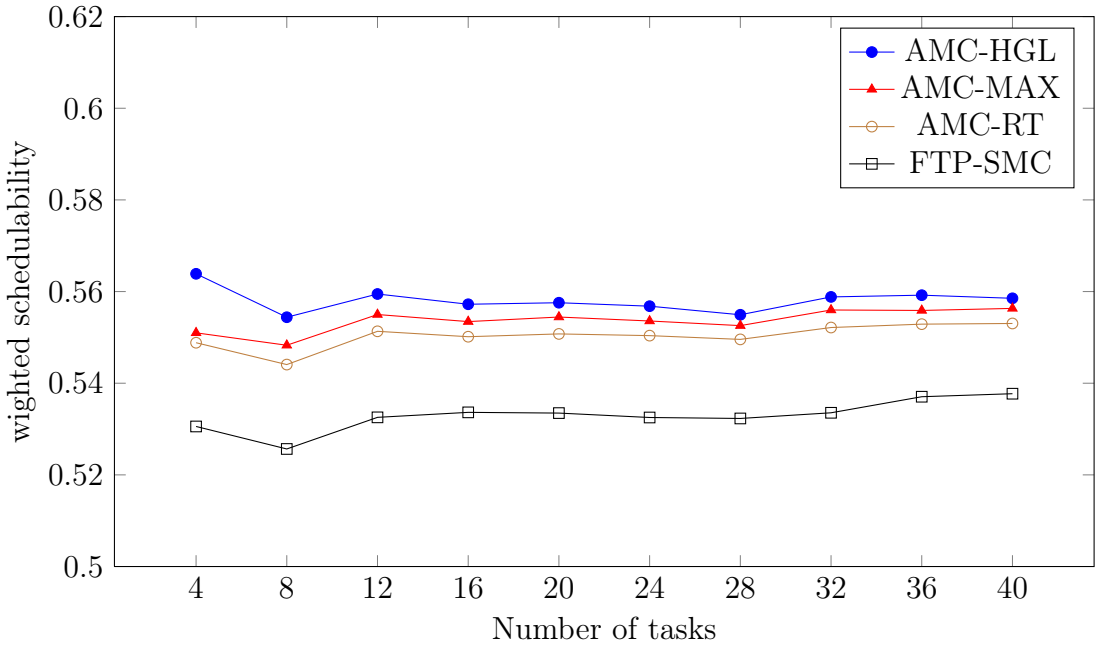


Figure 4.2: Schedulability evaluation for fixed task priority SMC/AMC analyses with varying numbers of tasks (CF=1.5, NC=2)

Comparison of FJP Analyses: As we have described in Section 3.4, assigning job priorities to mixed-criticality sporadic tasks was proposed by Li et al [66, 67]. Before priorities can be assigned to jobs, it is necessary to estimate job arrival patterns (i.e., the number of jobs for each task that can be executed in each criticality level) in a worst case busy period. Li provided an approach for job arrival pattern estimation based on the *load* of sporadic tasks. However, that approach is virtually useless in practice because of two reasons. First, the *load* calculation is computationally expensive [44]. In our preliminary test, we were unable to finish testing schedulability of even 10 randomly generated task sets (with the parameters NT=8, U=0.05, CF=1.5)

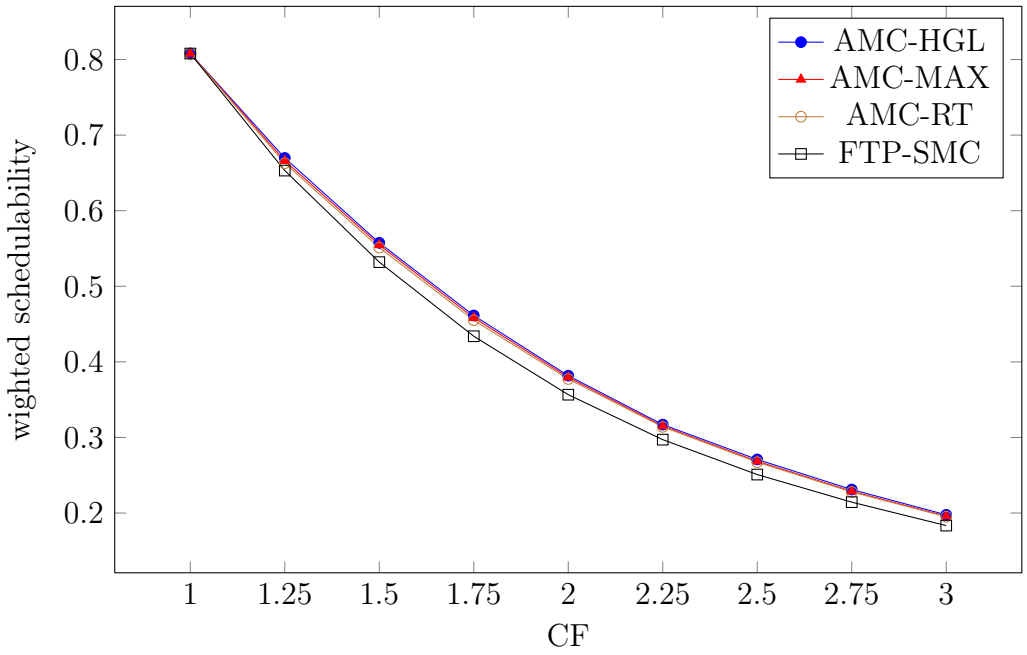


Figure 4.3: Schedulability evaluation for fixed task priority SMC/AMC analyses with varying CF (NT=20, NC=2)

in a day. Second, the successful rate (if any) that can pass the schedulability test based on this approach is often very poor. In fact, none of the generated task sets passed the schedulability test with the following parameters: NT=4, NC=4, CF=1.5. Figure 4.4 shows the result for our revised job arrival pattern (FJP-HGL) calculation described in Section 3.4 and Li's method[67].

Comparison of Different MC Scheduling Approaches: Because AMC-HGL and FJP-HGL perform best for their respective scheduling approaches, we will use them for the comparison of their representative analyses against other approaches. Figure 4.5 shows the percentage of schedulable task sets where each task set consists 20 tasks, 4 criticality levels and a CF ratio of 1.5 under different scheduling approaches. Figure 4.6 compares the approaches by varying the number of tasks in each task set. Figure 4.7 compares the approaches by varying CF. Figure 4.8 compares the approaches by varying the number of criticalities in a task set.

The following observations are illustrated by these figures:

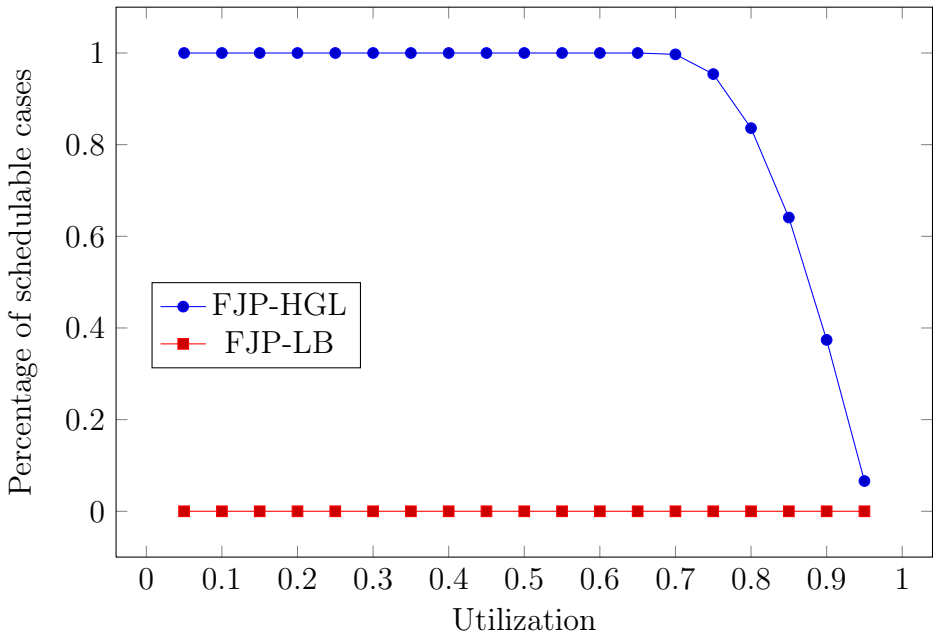


Figure 4.4: Schedulability evaluation for fixed job priority analyses (NT=4, NC=4, CF=1.5)

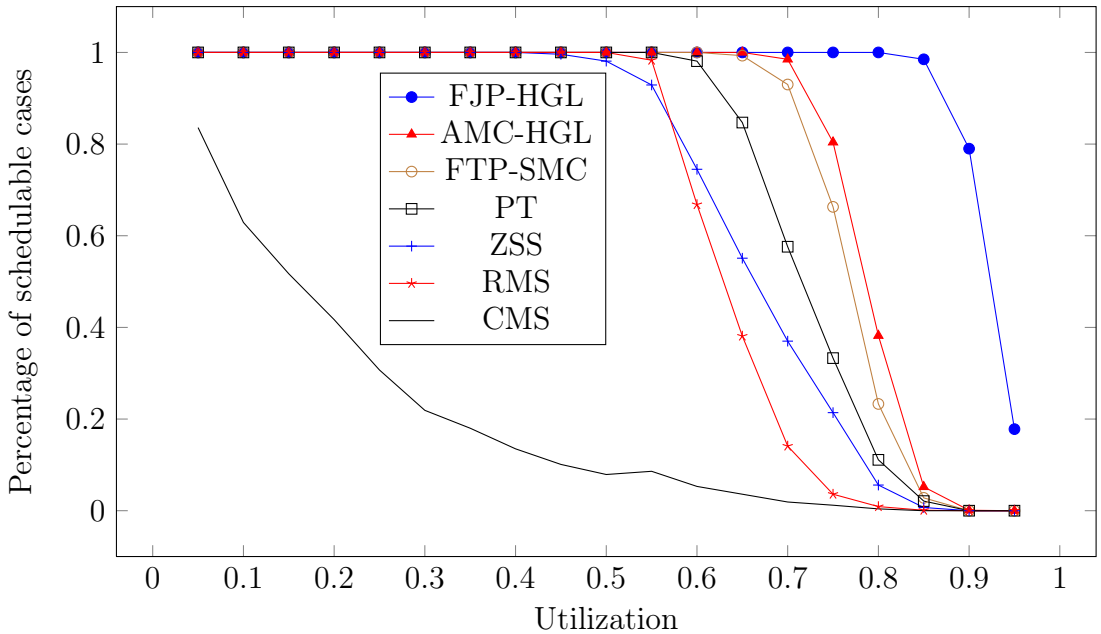


Figure 4.5: Schedulability evaluation for MC scheduling approaches with 20 tasks (NT=20, CF=1.5, NC=4)

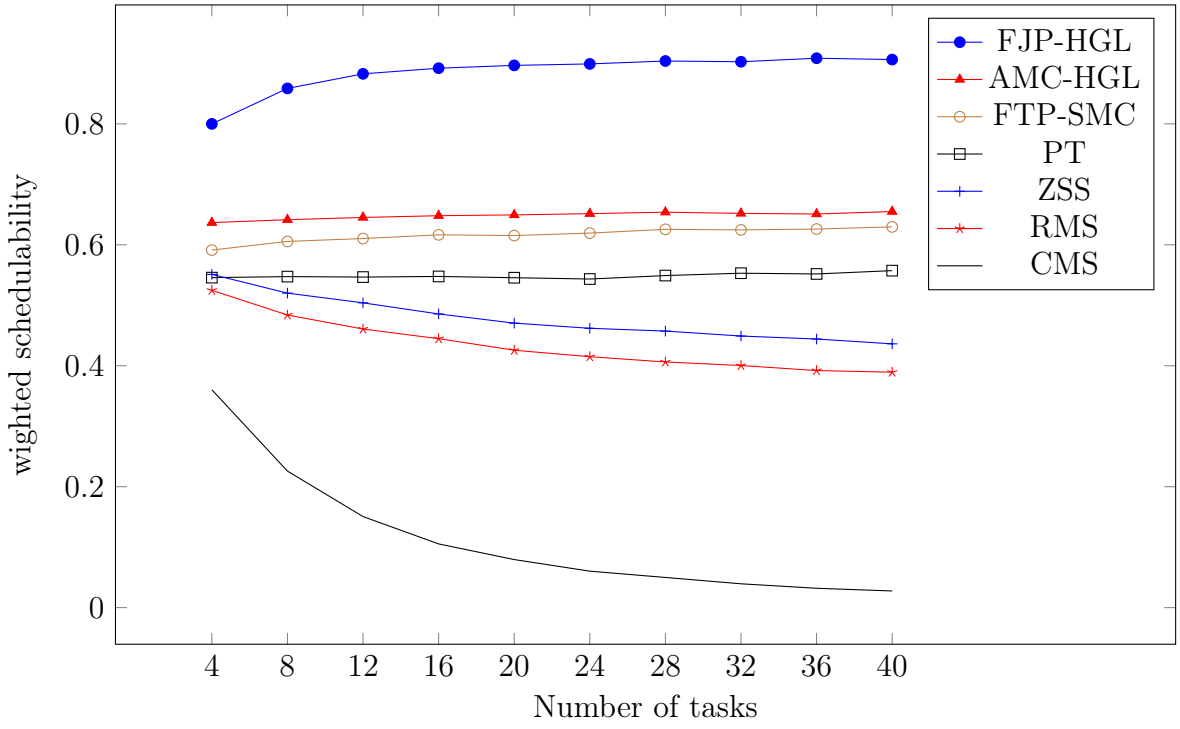


Figure 4.6: Schedulability evaluation for MC scheduling approaches with varying number of tasks (CF=1.5, NC=4)

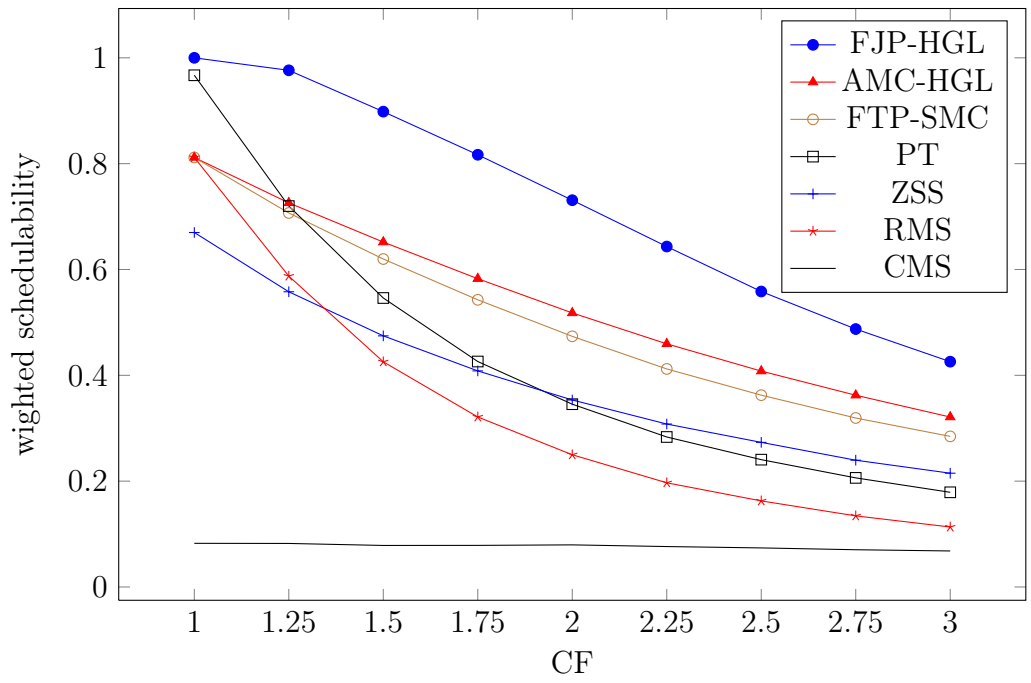


Figure 4.7: Schedulability evaluation for MC scheduling approaches with varying CF (NT=20, NC=4)

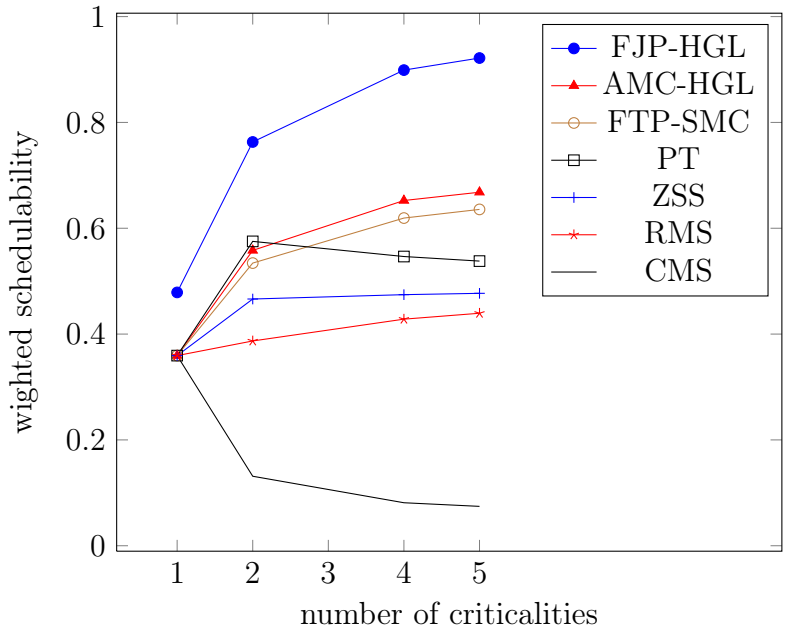


Figure 4.8: Schedulability evaluation for MC scheduling approaches with varying number of criticalities (NT=20, CF=1.5)

- CM performs very badly in terms of schedulability.
- AMC performs better than ZS because the time to suspend/abort lower criticality tasks is determined at run-time rather being statically pre-computed. This allows ACM to be more adaptive than ZS based on the workload.
- For the approaches based on Audsley's priority assignment algorithm (AMC or SMC), the rate of successful schedulable tasks increases as the number of tasks increases. This is largely due to the finer granularity for schedulable job/task selection given a fixed CPU utilization. This factor is more prominent when the number of tasks is small.
- Except for CM and PT, the number of successfully schedulable tasks increases as the number of criticality levels increases. This is because the finer the granularity of the criticality levels, the more tasks a high criticality task can suspend or abort, which thus increases its schedulability. However, we also observed an increase in the number of criticality levels also significantly increases the computation time for the schedulability tests, especially for FTP-AMC and FJP-AMC. This is because the computational complexity of the FTP-AMC and FJP-AMC schedulability tests depend on the length of longest busy period, and the busy period increases as the the number of criticality level increases.

Overall the key observation is that FJP-AMC performs best in all scenarios. Given the fact that FJP-AMC selects schedulable jobs instead of tasks in each step of Audsley's priority assignment algorithm, it is not surprising that FJP-AMC would outperform FTP-ACM because of the finer granularity. In theory, FJP dominates FTP because any FTP priority assignment is also a valid FJP assignment [21]. However, this does not translate to the necessity that FJP-HGL dominates FTP-HGL. This discrepancy comes from the fact that the job arrival patterns used by FJP-HGL are different from those of FTP-HGL. If an FJP algorithm chooses to test all possible job arrival patterns, it would dominate all possible FTP algorithms; however, the complexity of doing that would be too high in practice.

4.2 Schedulability Evaluation for Mixed-Criticality End-to-End Tasks

From Section 3.5, we see that mixed-criticality end-to-end scheduling involves deadline assignment algorithms and different mixed-criticality scheduling approaches. In order to understand the strength and weakness of these approaches, we now present a an quantitative comparison of these algorithms that we have conducted to evaluate those issues.

4.2.1 Workload generation

We studied the schedulability of randomly generated end-to-end task sets, each generated based on the following parameters:

- Each generated task set consisted of 8 independent tasks, where the number of subtasks was an integer randomly sampled from a uniform distribution between 2 to 8.
- The period of each task was an integer randomly sampled from a distribution between 100 to 10000.
- The total CPU utilization $U(PR_k)$ of a processor PR_k was a floating point value randomly sampled from a uniform distribution between 0.5 and 0.8.
- The criticality levels of tasks were assigned sequentially modulo N as tasks were generated; i.e., if the total number of criticality levels in a task set was N, then the criticality level ζ_i of the i-th generated task τ_i was $i \mod N$.
- The execution time at the lowest confidence level $C_{i,j}(0)$ for each subtask $\tau_{i,j}$ was determined by a uniformly distributed random number $u_{i,j}$ (called the utilization factor) that was randomly sampled from a uniform distribution between 0.1 and 1. With the availability of the task period T_i and the CPU utilization, $C_{i,j}(0)$ is derived from the following equation

$$C_{i,j}(0) = \max\{1, T_i \times \frac{u_{i,j}}{\sum_{\forall \tau_{m,n} \in Pr(\tau_{i,j})} u_{m,n}} U(Pr(\tau_{i,j}))\}.$$

- The ratio (CF) between the worst case execution time $C_i(\zeta_i)$ and $C_i(0)$ was 1.2. Given a CF, the execution time specification of $\tau_{i,j}$ was

$$C_{i,j}(\zeta) = \begin{cases} C_{i,j}(0) & \text{if } \zeta < \zeta_i, \\ CF \times C_{i,j}(0) & \text{otherwise.} \end{cases}$$

4.2.2 Scheduling approaches investigated

The following list summarizes the priority assignment approaches we investigated for end-to-end schedulability.

- Deadline monotonic scheduling (DMS).
- Period transformation (PT) with response time analysis.
- Static mixed criticality (FTP-SMC), priority assigned with Audsley's approach with response time analysis.
- Fixed task priority assigned with Audsley's approach coupled with adaptive mixed criticality run-time mechanisms using our improved analysis (AMC-HGL).
- FJP-AMC with our improved job arrival pattern as described in Section 3.4 (FJP-HGL).

4.2.3 Simulation

We randomly generated 1000 end-to-end task sets. For each task set Γ, we began by setting the end-to-end deadlines of each task τ_i in Γ the same as the period of τ_i (i.e., $D_i/P_i = 1$ or DP=1), assigning local deadline of each subtask in τ_i using proportional deadline assignment. Then we tested the schedulability of each end-to-end task in the task set Γ. Next, we increased the end-to-end deadline of all tasks in Γ to be 1.1 times of their respective periods (DP=1.1) and repeated the schedulability test process until the DP ratio of each task reached 1.8.

Figure 4.9 plots the number of schedulable end-to-end tasks under different scheduling approaches. Not surprisingly, FJP-HGL performs best among the approaches investigated, as it does in the uniprocessor environment. Although period transformation (PT) is generally better than rate monotonic (RM) scheduling in the uniprocessor environment, that is not the case for end-to-end task sets. In fact, PT is consistently worse than deadline-monotonic (DM) priority assignment. Compared to DM, PT would make the response times of all subtasks closer to their artificial relative deadlines, and thus increase the end-to-end response time in general.

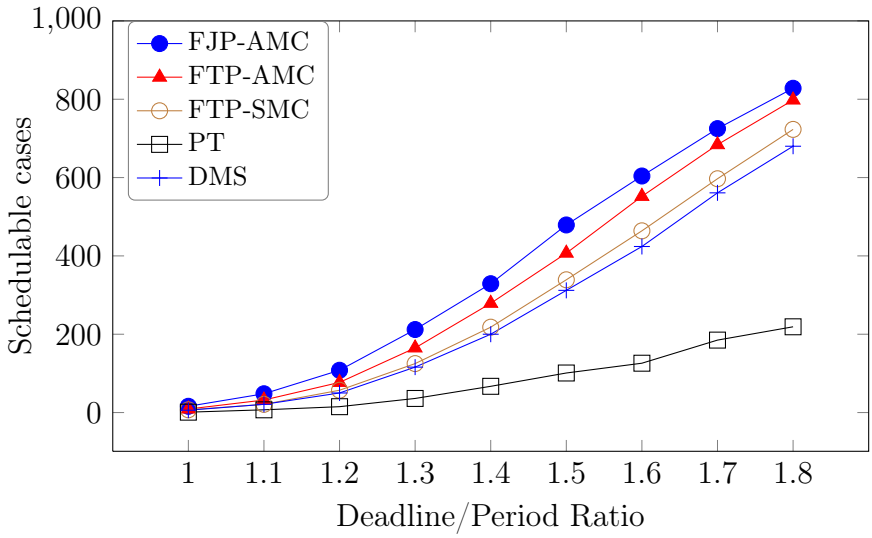


Figure 4.9: Comparison between different priority assignments using proportional deadline

Let $S_{i,j}$ be the difference between the relative deadline and response time of a subtask $\tau_{i,j}$. If all subtasks in a processor P_k meet their respective relative deadlines, i.e., $S_{i,j} > 0$ for each subtask $\tau_{i,j}$ in P_k, we call the subtasks on P_k *frozen*. If there exists any frozen subtask $\tau_{i,j}$, the slack $S_{i,j}$ can be redistributed to other unfrozen subtasks of task τ_i which thus can increase the assigned deadlines of unfrozen subtasks. Next, we can re-assign the priorities of unfrozen subtasks with the newly assigned deadlines, and calculate the end-to-end response times of all tasks. The process continues until all tasks are schedulable or no more frozen subtasks are available. We will call this technique *slack reallocation*.

Figure 4.10 shows the number of task sets (out of 1000) which are made schedulable because of slack reallocation. Figure 4.11 shows the number of schedulable task sets

with different priority assignment approaches with and without slack reallocation. We can see that slack reallocation is most helpful to FJP-HGL because of its finer granularity priority assignment approach. On the other hand, it does not help PT much.

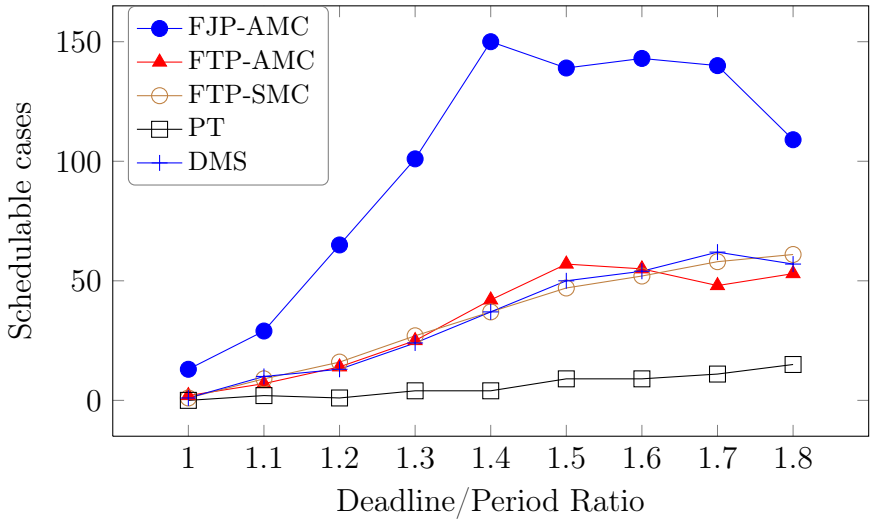


Figure 4.10: The improvement of slack reallocation

In order to understand effect of different deadline assignments, we also use normalized deadlines to evaluate end-to-end schedulability. Figure 4.12 compares the schedulable end-to-end tasks of FJP-HGL (with and without slack reallocation) using proportional deadline (PD) and normalized proportional deadline (NPD) approaches. It shows these two deadline assignment approaches are very close and neither approach is a clear winner.

Table 4.1 shows the Number of task sets that are schedulable only under each scheduling approach with or without slack reallocation when proportional deadline assignment is used. Although a task set is mostly likely to be schedulable using FJP-HGL, the table shows that FJP-HGL does not dominate other scheduling approaches.

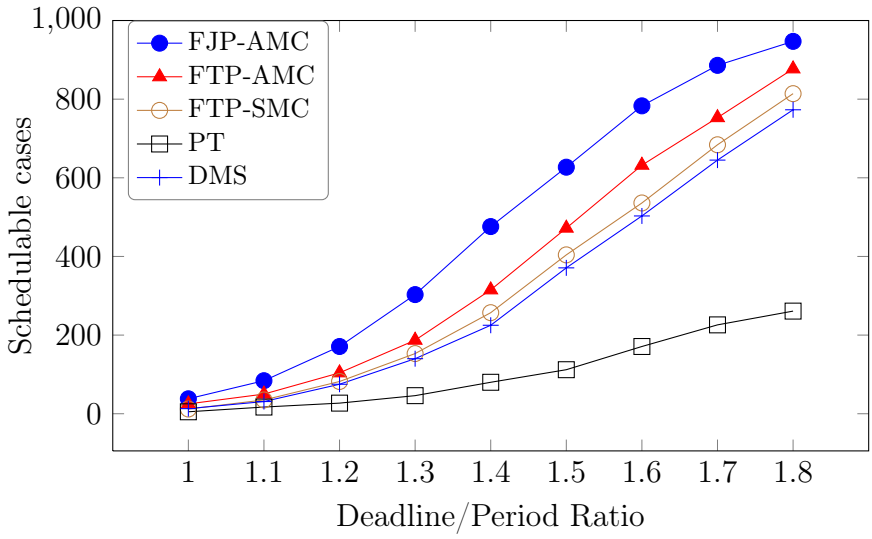


Figure 4.11: Comparison between different priority assignments with and without slack reallocation using proportional deadline

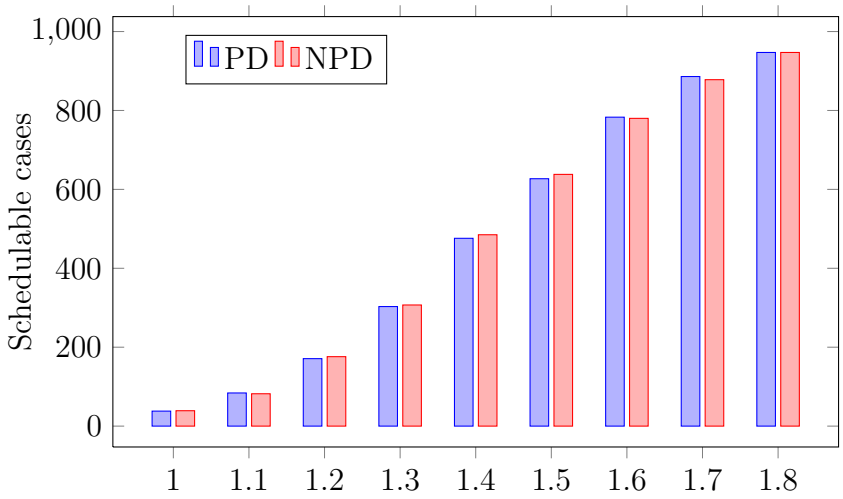


Figure 4.12: Comparison between PD and NPD for FJP-HGL

Table 4.1: Number of task sets that are uniquely schedulable by each scheduling approach with proportional deadline assignment (DP=1.6)

	FJP-HGL	FTP-HGL	FTP-SMC	PT	DMS
without slack reallocation	72	4	1	9	1
with slack reallocation	128	9	1	9	1

4.3 Mixed-Criticality Runtime Enforcement Mechanism Implementation and Evaluation

Among the scheduling approaches we have considered for mixed-criticality real-time systems, FTP-SMC, rate monotonic and criticality monotonic scheduling can be easily implemented atop thread priority mechanisms commonly provided by modern operating systems. To support period transformation, additional bandwidth-preserving server mechanisms[69] are needed to ensure a task will not execute beyond its budget within a transformed period. Enforcement of zero-slack and AMC scheduling requires the use of additional timers to trigger mode changes and deadline miss handling. The additional overheads of period transformation and zero-slack scheduling mechanisms, compared to the FTP-SMC approaches are thus of practical interest. In this section, we present a user space implementation of deferrable server, zero-slack scheduling and AMC mechanisms atop Linux, and evaluate the run-time overhead of the different scheduling policies for mixed-criticality systems.

4.3.1 Zero-Slack Scheduling Implementation

A task is implemented as a thread with a priority assigned in accordance with the application and platform. For Linux, valid priority levels range from 0 to 99, where 99 is the highest priority. Our zero-slack scheduling mechanisms reserve priority levels 1 and 99 for criticality enforcement purposes. Task suspension is emulated by lowering the priority of a task to 0. As a result, application tasks can use only priority levels 2 through 98. To simplify discussion, we assume that the criticality levels assigned to a task set are contiguous positive integers.

Each task is associated with three periodic timers, for the job release, ZSI, and deadline. Expiration of the job release timer is received and handled in the task's associated thread. An additional enforcement thread with priority 99 is created to wait for all other timer expiration events as well as for job termination events, and to make scheduling decisions based on the events it receives. To handle task suspension and resumption correctly, the enforcement thread maintains a binary heap of criticality levels. The top element of the heap has the highest criticality among the tasks that

have been suspended. For convenience, we use ζ^{s0} and ζ^{s1} to denote the criticality of the top two elements in the binary heap.

A suspension event with a criticality level is used to trigger the enforcement thread to suspend a subset of tasks. When the enforcement thread receives a suspension event with criticality x, it suspends the tasks whose criticality is less than or equal to x and higher than ζ^{s0}. In addition, criticality x is inserted into the binary heap. Notice that x could be smaller than ζ^{s0} and hence no tasks would be suspended. However, the new value of x should still be inserted into the binary heap so that the enforcement thread can keep track of which tasks are in critical mode.

When the ZSI timer of a job $J_{i,k}$ expires and the job has not finished its execution, a suspension event with criticality $\zeta_i - 1$ is sent to the enforcement thread. Deadline timer expiration of a task τ_i is handled in the same way as ZSI timer expiration, except that if $J_{i,k}$ misses its deadline, a suspension event with criticality ζ_i is sent instead. When a job $J_{i,k}$ finishes while in critical mode or after missing its deadline, an event is sent to the enforcement thread to wake up the tasks that were suspended. When the event is received, the enforcement thread restores the priority of each task τ_j where $\zeta^{s1} < \zeta_j \leq \zeta^{s0}$, and then the top element of the binary heap is removed. Priority restoration is done in non-increasing criticality order. When an awakened job $J_{i,k}$ has already missed its deadline, the priority of τ_j is changed to 1 instead of π_j. In addition, ζ_j is inserted into the binary heap so that tasks with criticality levels less than or equal to ζ_j will remain suspended.

4.3.2 Period Transformation Implementation

To support period transformation, we also implemented a *deferrable server* enforcement mechanism in MCFlow. In the deferrable server approach, a task with a transformed period is executed within a server thread. Each server has a period, a budget, and a priority, all of which are assigned according to the transformation mechanism described in Section 4.1. The server budget is replenished at the beginning of each period. The budget decreases while the server is executing a task and is preserved (until the end of the current period) while the server is idle. A server can execute its respective task as long as its budget has not been exhausted.

Similar to our zero-slack scheduling implementation, a manager thread at highest priority is allocated to manipulate the consumption and replenishment of servers' budgets. This thread sits in an `epoll_wait` system call and waits for the budget replenishment and exhaustion events which are generated by the POSIX real-time timer APIs. For the budget replenishment events, we use timers with the `CLOCK_MONOTONIC` clock id (wall clock timer) to generate asynchronous timeout signals. To monitor the budget consumption of a server, timers with the `CLOCK_THREAD_CPUTIME_ID` clock id (thread CPU timer) are used. However, in the implementation of our test platforms, relying on the thread CPU timer to trigger budget exhaustion events is imprecise because the timer expiration can be triggered only right after scheduling quantum expiration. In our platform, the quantum duration is 1 ms. That is, if a thread CPU timer expires 100 μs after the quantum expiration time, the expiration event of the thread CPU timer timer would have to to wait another 900 μs to be triggered by the kernel. On the other hand, wall clock timers can always be triggered with microsecond level precision, regardless of the quantum expiration period.

For a system with only a few servers, imprecise triggering might not be a significant problem. However, such jitter can aggregate as the number of servers grows. To overcome this limitation, we utilize both thread CPU timers and wall clock timers to generate budget exhaustion events of a server. Whenever a budget replenish event arrives, we set the priority of the server thread to its respective real-time priority and reset the corresponding thread CPU timer. At the same time, a wall clock timer is set to generate asynchronous signals based on the remaining time on the thread CPU timer. Upon expiration of the wall clock timer, the corresponding thread CPU timer is checked to see if the budget has been exhausted. If the budget is not exhausted, the wall clock timer is armed again with the remaining time read from the thread CPU timer. If the budget is exhausted, the priority of the server thread is set to the lowest priority, 0.

4.3.3 AMC Implementation

Our implementation of AMC is basically a mixture of zero-slack scheduling and period transformation. Common to the implementation of all these scheduling approaches, they all need a manager thread of highest priority for certain scheduling control

actions. Like period transformation, AMC requires both wall clock and thread CPU timers to monitor the progress of a task. Similar to the expiration of a ZSI timer, the AMC implementation must suspend lower criticality tasks once the execution of a task exceeds the execution specification for a particular certain confidence level.

However, the availability of only 99 priority levels in Linux limits the usefulness of this implementation for use use with FJP-AMC scheduling without any modification. FJP-AMC assigns a unique priority level to every single job in the job arrival pattern. The highest priority is reserved for manager thread and the lowest priority is reserved for suspended task, only 97 priority levels are left for job assignments. In the 1000 task sets we generated based on the method in Section 4.1 (with parameters CF=1.5, NC=4, NT=20, U=0.8), on average FJP-HGL required 115.408 distinct jobs in their job arrival patterns. Therefore, the 97 levels of priorities may be inadequate.

To overcome this limitation, we implemented an alternate way to enforce job priorities when the priority levels provided by the operating systems are not sufficient. Instead of mapping job priorities to native OS priorities, our alternative implementation only uses 3 levels of native OS priorities (namely, `MGR_PRIO`, `RUN_PRIO` and `SUSPENDED_PRIO` in the decreasing priority order). The manager thread (running in `MGR_PRIO` priority) works as a job dispatcher to control when jobs are released, preempted and suspended. Only the thread of the current running job with highest FJP priority is assigned with `RUN_PRIO` priority; other threads are set to the `SUSPENDED_PRIO` priority.

Besides job priority enforcement, FJP-AMC also requires a run-time mechanism to compute the priorities of jobs. To avoid run-time job priority computation of pseudo-polynomial complexity described in [66], we implemented PLRS [50] which off-loads the pseudo-polynomial complexity to an off-line stage and employs an on-line algorithm of quadratic complexity.

4.3.4 Empirical Evaluation

To measure the overhead imposed by these scheduling mechanisms, we conducted experiments on a testbed consisting of a 6-core Intel core7 980 3.3GHZ CPU with

hyper-threading enabled, running Ubuntu Linux 10.04 with the 2.6.33-29-realtime kernel which incorporates the Linux RT-Preempt patch [82]. To avoid task migration among cores, CPU affinity was assigned so that our test program was executed in one particular core. All hardware IRQs except those associated with timers were assigned to cores other than the one for application execution. Each task was implemented with a `for` loop with a fixed number of iterations, where every 31 iterations yielded a 1 μs workload. In each iteration, the CPU timestamp counter was read and then compared with the counter read from the previous iteration. If the difference was greater than a specified number of ticks (700), we considered the thread to have been preempted and the new timestamp counter was stored. After a specified amount of time, all stored timestamp counters were written to a file, and then the test program terminated.

ZSRM Measurements: Our first experiment used the task set in Table 1.1, where τ_h and τ_ℓ ran workloads of 4 and 2 ms within their periods, respectively. Table 4.2 shows the time stamp log for the experiment in microseconds and Figure 4.13 is a graphical illustration of the log. To enhance readability, all the time stamps in Table 4.2 have been normalized so that the first time stamp of the table is 0. Each cell in the table represents a starting and end time of a task running continuously. The number in parenthesis is the time difference between the end of the current entry to the start of next entry. For instance, the first job $J_{1,1}$ of τ_1 starts at 1938 μs and runs until 2840 μs before it is preempted. After another 3 μs, $J_{1,1}$ continues to run from 2843 μs to 3840 μs. The time stamps in the same shaded box belong to the same job. For example, $J_{2,1}$ runs from 0 to 1936 μs; similarly, $J_{1,1}$ runs from 1938 μs to 6816 μs.

We found that approximately every 1000 μs, there was a 3 μs interval that was not used by the task set or by the enforcement thread, which we attribute to the invocation of the Linux scheduler scheduling every quantum, which was set to 1 ms on our test machine.

As was mentioned earlier, the enforcement thread can be invoked by job termination as well as by ZSI timers and deadline timers. In this experiment, τ_h did not have a deadline timer because there was no higher-criticality task with which it could

Table 4.2: Time stamp log (in μs) for the example in Table 1.1 where only the first jobs of the tasks are overloaded

Enforcement	τ_h	τ_l
		0-840 +(4)
		844-1840 +(3)
		1843-1936 +(3)
	1938-2840 +(3)	
	2843-3840 +(3)	
	3843-4840 +(3)	
	4843-4990 +(6)	
4996-4996 +(2)		
		4998-5840 +(3)
		5843-5992 +(2)
5994-6000 +(1)		
	6001-6816 +(9)	
6824-6828 +(1)		
		6829-6840 +(3)
		6843-7769 +(2226)

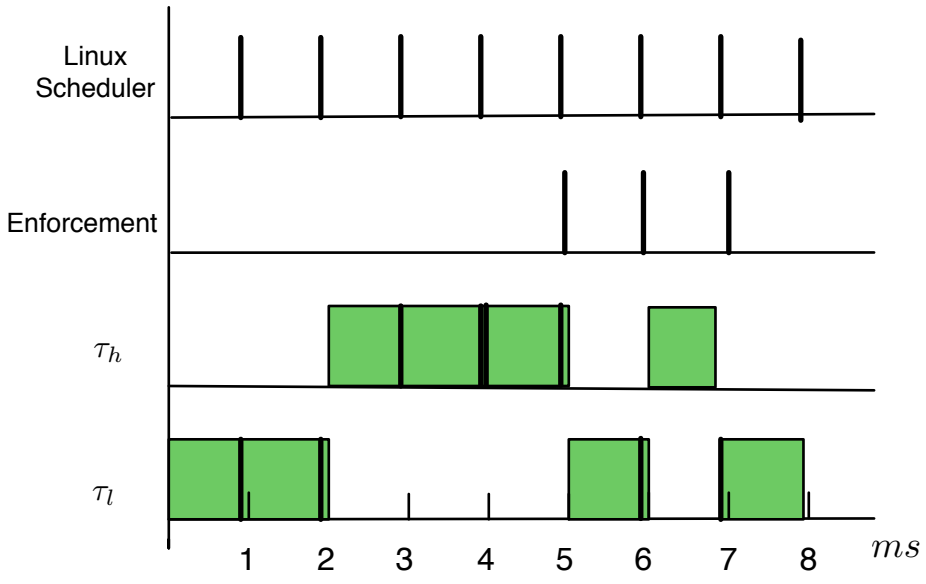


Figure 4.13: Illustration of the time stamp log in Table 4.2

interfere if it missed its deadline. Similarly, τ_l did not have a ZSI timer because there was no lower-criticality task that it needed to suspend when it entered critical mode.

When a timer expires or a job terminates, there is an overhead to switch from the current task thread to the enforcement thread. Depending on the scheduling context, the enforcement thread may demote or promote the priorities of some tasks and then return to the task with the highest priority. Thus the overhead of every enforcement thread invocation is the sum of the overheads for preemption invocation, thread priority adjustment, and preemption return.

Comparison between Scheduling Policies: To evaluate the cost of criticality enforcement in the implementations of different scheduling policy enforcement mechanisms, we benchmarked the task set in Table 4.3, and compared the response times seen for FTP-SMC, ZSRM, PT, FTP-AMC and FJP-AMC with PLRS according to the busy intervals from each job release of τ_4 until the CPU again became idle. Each AMC based approach has two versions, one that directly maps the task or job priority to the OS thread priority (dubbed AMC-OS and PLRS-OS), and another that uses middleware mapped priority, i.e., the manager thread enforces correct task/job priorities (dubbed AMC-MW and PLRS-MW). For ZSRM, the priority of each task was assigned by increasing rate (i.e., $\pi_1 > \pi_3 > \pi_2 > \pi 4$). For FTP-SMC, PT and FTP-AMC, the priorities of the tasks are assigned in the order $\pi_1 > \pi_2 > \pi_3 > pi_4$.

In this benchmark, each task τ_i executed for $C_i(0)$ time units and each experiment ran for 3 seconds for each scheduling approach, so that 100 busy periods were observed. We chose the execution time $C_i(0)$ for every task so that the busy intervals would not vary for different scheduling approaches. If longer execution times were allowed, the overheads couldn't be directly compared.

Table 4.3: A 4 tasks example (in ms)

Task	$C_i(0)$	$C_i(\zeta_i)$	T_i	ζ_i	Z_i
τ_1	4	6	10	1	10
τ_2	3	5	20	1	3
τ_3	6	6	15	0	0
τ_4	2	2	30	0	0

Figure 4.14 shows the average, maximum and standard deviation of the busy periods measured. Since SMC does not require any other run-time mechanism other than the priority based thread dispatching provided by the operating system, it serves as a baseline in our comparison. The average busy interval for FTP-SMC was 27.092 ms while the average busy interval for ZSRM was 27.098 ms. The reason for such low overhead in ZSRM was because no ZSI timer will be expired in our benchmark case and thus no extra preemption will occur in the ZSS task schedule. On the other hand, PT and AMC incur more overhead because of their schedule involves more preemption. In the AMC based approaches, the run-time priority computation of PLRS and middleware-mapped priority mechanisms did incur some extra overhead. However, those overheads are very small. Even comparing PLRS-MW (which has the highest run-time cost) to SMC, there was only a 0.3% increase in the time of busy interval.

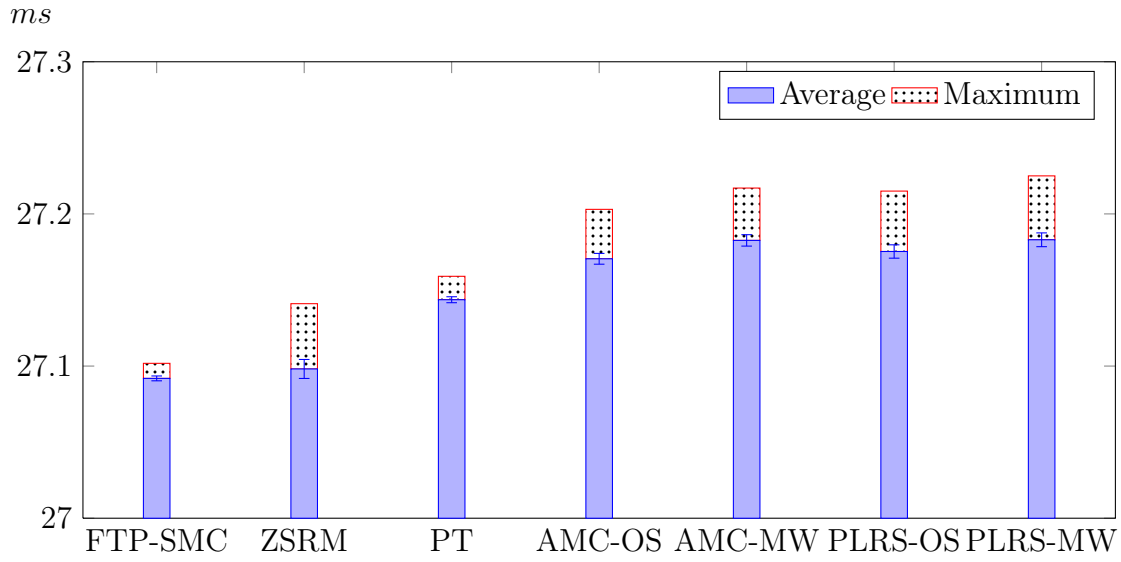


Figure 4.14: Busy period time comparison between different scheduling mechanisms for the task set in Table 4.3

Preemption Overhead Micro Benchmarks: To better understand the overheads of each individual segment of enforcement thread invocation and execution, we developed another test case to measure the preemption overhead in the system. In this test case, we use the ZSS implementation but the ZSIs of tasks were assigned artificially so that overheads could be easily identified and measured rather than using Algorithm 1. The rationale for this artificial assignment is that those overheads are

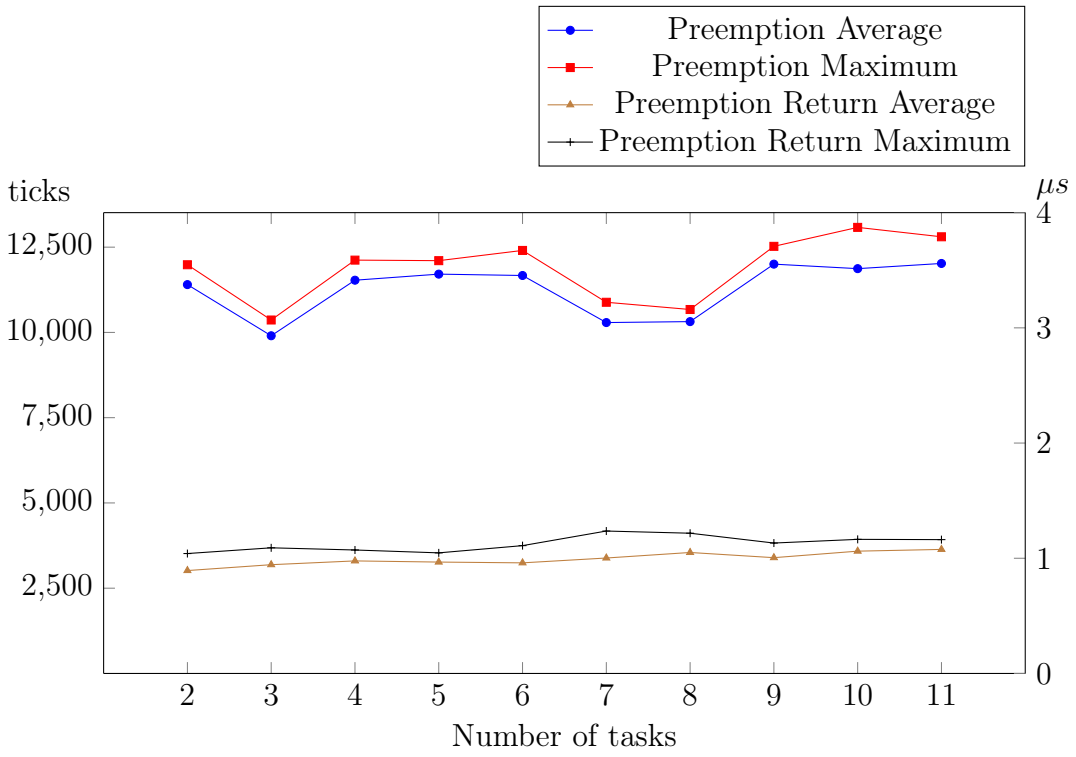


Figure 4.15: Preemption overhead for Linux RT kernel

related to the number of threads and the operation performed during mode switching, rather than to the exact instants when they take place. All tasks were set to have 2 ms execution time and had the same period, 50 ms. The lowest priority task τ_0 was assigned to the highest criticality level with $Z_0 = 1$ ms. The rest of the tasks were assigned in such a way that the priority of a task was equal to its criticality level. By varying the number of tasks in the system, we obtained the overhead of preemption, preemption return, and priority adjustment, as shown in Figures 4.15 and 4.16.

As is shown in Figure 4.15, the overheads of preemption and preemption return are not linked to the number of tasks in the system and are about 4 μs and 1 μs respectively.

From Figure 4.16, we can clearly see that the cost of priority adjustment is linear with regard to the number of threads to be promoted/demoted. However, in our experiment, the overheads for the first invocations of each such adjustment were always far higher than the rest. As a result, we present the cost of first invocations separately from the others, in the curve labeled "First". The curve labeled "2nd Largest" and the curve labeled "Average" show the maximum and mean (respectively) of the rest

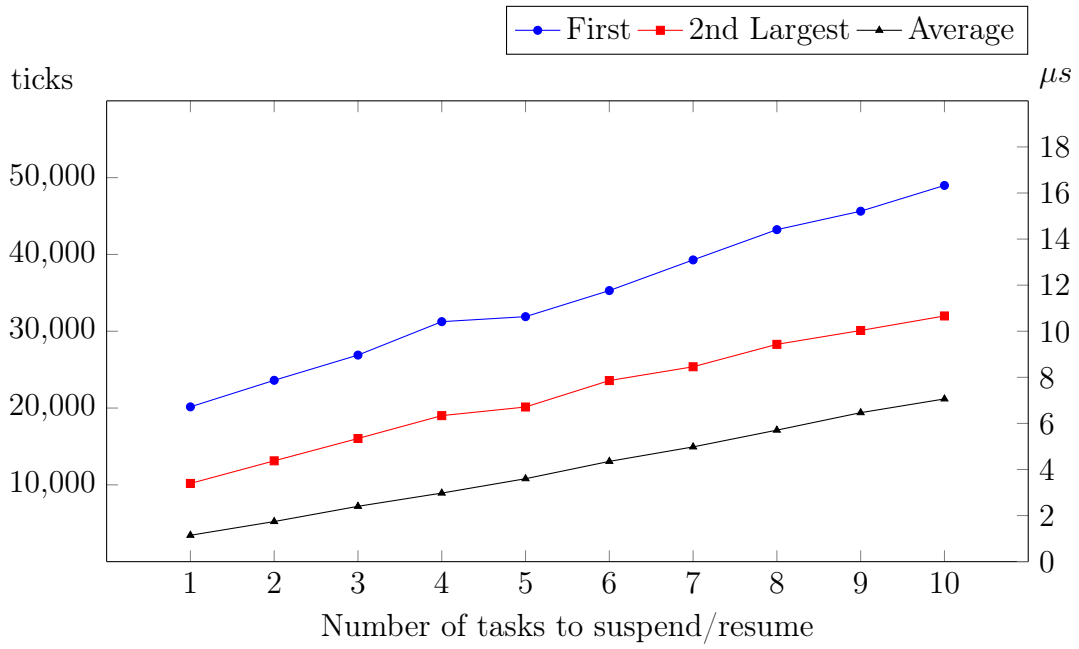


Figure 4.16: The cost of priority adjustment

of the invocations. We observe that the second largest overhead is consistently 2 to 3 μs longer than the average, which occurs when the periodic invocation of Linux scheduling occurs during the priority adjustment.

Based on these results, we can easily estimate the overhead of timer expiration or job termination. For example, the cost of a ZSI timer expiration with 8 tasks to suspend is about $4 + 5 + 1 = 10$ μs.

Similar to our zero-slack implementation, the overhead incurred by our deferrable server and AMC implementation can also be divided into three parts: thread preemption, preemption return, and manager thread handling. In our experiments, the preemption overhead and preemption return overhead for the deferrable server and AMC implementations were very close to what is shown in Figure 4.15; therefore, we omit those details for brevity.

Table 4.4 shows the average and maximum cost of manager thread invocations for our implementations of period transformation and various adaptive mixed-criticality scheduling enforcing mechanisms. As was mentioned previously, the PT manager thread is responsible for budget replenishment and exhaustion and for adjusting server

Table 4.4: The average and maximum cost in cycles of manager thread invocation

	PT	AMC-OS	AMC-MW	PLRS-OS	PLRS-MW
Average	3587 (1 μs)	7111 (2.1 μs)	11656 (3.45 μs)	8306 (2.46 μs)	11592 (3.43 μs)
Maximum	22448 (7.8 μs)	26412 (7.8 μs)	34188 (10.12 μs)	34968 (10.35 μs)	41484 (12.28 μs)

thread priorities, as as well as for canceling the budget exhaustion timer when a job finishes. Regardless of the different functionalities involved, the average and maximum response times of each manager thread invocation were about 3587 and 22448 cycles, respectively, or about 1 and 6.6 μs, respectively.

Compared to our PT implementation, where job release and job termination are handled in their respective task thread, the manager thread of our AMC implementation is also involved whenever a job is released or terminated because the AMC manager thread needs to identify the current running job and sets the wall clock timer to trigger the event for the system criticality level indicator change. As a result, AMC approaches incurs more overhead than PT as shown in Table 4.4. For the middleware-mapped priority implementations, the manager thread also needs to adjust thread priorities to maintain the correct job execution ordering, which increases the cost the manager thread processing. PLRS, on the other hand, requires additional job priority computations on some of the manager thread invocations.

In general, Table 4.4 demonstrates the manager thread overhead of PLRS-MW is the highest among the scheduling approaches we have implemented because it involves more operations than others. On the other hand, the run-time overhead of a scheduling approach is strongly influenced by the number of preemption and task suspension/resumption in a schedule. Because different scheduling approaches likely will produce different run-time schedules, there is no absolute guarantee which scheduling approach could result to the fewest number of manager thread invocations. Therefore, it is hard to say which scheduling approach will impose the highest run-time overhead even though the average overhead of PLRS-MW is the highest per manager thread invocation.

Nevertheless, FTP-SMC relies only on priority, and no extra run-time mechanism for task suspension/resumption is required; thus it is expected to have the lightest overhead in general. Therefore we recommend that system developer consider adopting FTP-SMC if the target task set can be schedulable with the approach. For the cases where more stringent CPU utilization bounds are required, PLRS with our new worst case arrival pattern calculation could be a good solution.

Chapter 5

Related Work

5.1 Related Work on Middleware

The specialized system described in [92] focuses on cyber and physical component integration without middleware support. The middleware described in [54] is based on the work from [92] to target real-time hybrid testing of civil structures. Like MCFlow, it is also based on dependent task graphs; however, it was designed for uniprocessor systems, and lacks the optimizations for multi-core platforms or the ability to reconfigure graphs of dependent subtasks flexibly and transparently. Achieving those capabilities across multi-core platforms is an important motivation for developing MCFlow and is one of the key contributions of this work.

The OMG Real-Time CORBA specification [76] supports network transparency for software component development and provides real-time policies and mechanisms including standard interfaces to specify resource requirements and configure object request broker (ORB) end-system resources such as thread priorities, message buffers, connections, and network signaling, to control ORB behavior. TAO [57] is a full-featured Real-Time CORBA [76] ORB. However, TAO was originally developed for single processor systems, and does not provide mechanisms for fine-grain parallelization of subtasks or for inter-core communication optimizations on multi-core platforms. The CORBA programming paradigm is based on a remote method invocation model, so that implementing end-to-end tasks using CORBA requires far stronger coupling among subtasks than MCFlow. In addition, CPU affinity and release guard mechanisms can only be implemented ad-hoc without direct support from middleware itself.

TAO's Real-Time Event Service [51] provides support for decoupled communication between objects, which makes it more convenient to implement end-to-end tasks. However, the event service requires a centralized event dispatcher which may lead to high synchronization overhead and thus may become a bottleneck on multi-core platforms. CPU affinity mechanisms also must be implemented ad-hoc; nevertheless, release guards can be supported through careful use of event correlation mechanisms provided by the event service.

The MC-ORB [98] middleware was specifically developed for multi-core platforms. However, it is also based on the remote method invocation paradigm used by CORBA and is designed for task sets where little or no dependency exists among tasks. Which core is used to execute a job is determined at run-time based on the current system load. It does not optimize task communication between cores, nor does it provide direct support for release guards.

Parallel programming languages, extensions, and libraries, such as Cilk [27], OpenMP [34] and Intel Thread Building Blocks [81] assume that the programmer should be responsible for exposing parallelism in source code but may defer decisions about how to divide the work between processors to a run-time scheduler. That is, locally and network triggered subtasks must be programmed differently, making it more difficult to reconfigure the allocation of tasks based on the results of scheduling analysis. Furthermore, those technologies use a central work stealing queue [29, 12] for task dispatching, which is not suitable for real-time systems because a subtask in a queue can only be dispatched whenever a thread/core is idle. Even if a priority queue is used, if all threads are running lower priority subtasks, the higher priority subtasks in the queue won't be dispatched which results in a priority inversion.

Dataflow programming languages and frameworks such as SystemC [47], StreamIt [15], and FastFlow [14] also have been developed, but none of them is specifically designed to support real-time distributed applications where (1) computations can be flexibly configured for execution on a single core, between cores of a common host or between multiple hosts, while ensuring that (2) timing constraints such as end-to-end deadlines are strictly enforced.

Table 5.1: The DO-178B standard

Level	Failure Condition	Interpretation
A	Catastrophic	Failure may cause a crash
B	Hazardous	Failure has a largest negative impact on safety or performance, or reduces the ability of crew to operate the plan due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers
C	Major	Failure is significant, but has a lesser impact than a Hazardous failure.
D	Minor	Failure is noticeable, but has a lesser impact than a Major failure.
E	No Effect	Failure has no impact on safety, aircraft operation, or crew workload.

5.2 Related Work on Mixed-Criticality Scheduling

In recent years, multiple papers have been published related to mixed-criticality scheduling. Vestal [95] first proposed a formal model for representing real-time mixed-criticality tasks to support analysis of the safety of software systems based on the RTCA DO-178B software standard. DO-178B is a software development process standard, which assigns criticality levels to tasks categorized by effects on commercial aircraft as shown in Table 5.1, as a means of certifying software in avionics applications. Vestal [95] used fixed-priority scheduling and provided a preliminary evaluation using three real world mixed-criticality workloads which showed that priority assignment [17] and period transformation [85] improved the utilization of the system, in comparison to deadline monotonic analysis.

Baruah and Vestal [21] then studied fundamental scheduling-theoretic issues with fixed task-priority, fixed job-priority and earliest deadline first (EDF) scheduling policies, under Vestal's mix-criticality model. In contrast to the traditional single-criticality task set where EDF is known to be *optimal* for uniprocessor scheduling in the sense that EDF always meets all deadlines for all feasible traditional systems, they showed that EDF was not optimal for mixed-criticality tasks in the uniprocessor environment. They also showed that a mixed-criticality task set is schedulable under an EDF scheduling policy if and only if it is feasible under traditional EDF scheduling analysis where the multiple specifications of worst case execution times are ignored

and treat each mixed-criticality task as a traditional task with the worst execution time of the highest criticality. In fact, fixed task-priority and EDF scheduling are incomparable because neither dominates the other; in other words, a mixed-criticality task set that is schedulable under fixed task-priority scheduling may not be schedulable under EDF scheduling, and vice versa. To achieve better task schedulability, they proposed a hybrid-priority scheduling algorithm which dominated both fixed task-priority scheduling and EDF scheduling. This algorithm generalizes both EDF and Audsley's method such that each task is assigned a priority that is not necessarily unique. Tasks of different priorities are scheduled according their priority settings; tasks of the same priority are scheduled in EDF. The priority assignment algorithm extends Audsley's algorithm to select tasks in increasing priority order. Instead of using the modified Joseph-Pandya worst case response time analysis [59] in each task selection step, Baruah et al. adopted a simulation based approach to test whether a task would fail to be schedulable at a given priority level.

In [20], Baruah et al. investigated the schedulability of mixed-criticality systems consisting of a finite number of (non-recurring) jobs. Their analysis demonstrates that complexity of mixed-criticality schedulability testing is NP-hard even when all jobs are released at the same time and the total number of criticality levels in the system is 2. It also proves that priority-based scheduling offers a processor speedup factor between 1.236 and 2 compared to EDF when the number of criticality levels is 2 or approaches infinity, respectively.

Li [66] further developed an on-line algorithm for scheduling sporadic tasks on a per-job basis. The algorithm is also based on Audsley's approach but with a different function to test the feasibility of a job running at a particular priority. The new function allows Li's algorithm to be more adaptive on the arrival pattern of jobs rather than relying on the static periodic model. However, this algorithm needs an on-line priority assignment recomputation with psudo-polynomial complexity in the worst case. This may make the algorithm unacceptable for systems that require stringent worst-case bounds on timing behavior.

To overcome the limitations of Li's algorithm, Guan [50] presented an algorithm called PLRS to address this issue. Like Li's algorithm, PLRS is also a fixed-job priority scheduling algorithm. However, instead of solely relying on high-complexity

on-line priority recomputation for job scheduling, PLRS separates the algorithm into two stages: an off-line priority computation and an on-line priority assignment stage. The PLRS off-line priority computation is essentially the same as the on-line priority recomputation of Li's method, but PLRS only computes jobs of the worst case busy period when all tasks are released at the same time. The resulting information is then used by a lighter weight on-line algorithm for priority assignment. The complexity of the off-line stage is still psudo-polynomial but that of the on-line stage is reduced to being quadratic in the number of tasks.

In [19, 31], Baruah et al. developed an adaptive approach for scheduling mixed-criticality sporadic task sets on uniprocessors systems. Unlike Li's algorithm and PLRS where released jobs won't change priorities, this algorithm requires the scheduler to demote the priorities of lower criticality jobs once it detects that the execution time of the current running job J_i has exceeded its respective WCET $C_i(\zeta_i)$ for the given criticality level. Notice that this scheduler is similar to the zero-slack scheduler in that they both involve mode change behavior. However this scheduler requires monitoring the run-time progress of jobs in order to determining the exact time instant to change mode whereas the zero-slack instant used by a zero-slack scheduler is a fixed time duration relative to the release time of a job. Therefore, this approach requires a higher degree of run-time support than zero-slack scheduling. The priority assignment algorithm in this approach is based on Audsley's approach with an alternative task feasibility testing function which is tailored to the mode change behavior.

Anderson et al. [16] developed an extension of Linux to support mixed criticality scheduling on multi-core platforms, using a bandwidth reservation server to ensure temporal isolation among tasks with different criticalities. Tasks of the same criticality are executed in one *container* with a predefined period and budget. Intra-container task scheduling for high criticality tasks uses a cyclic executive approach where scheduling decisions are statically pre-determined offline and specified in a dispatching table, whereas EDF can be used for low criticality containers.

Pellizzoni et al. [78] also used a reservation-based approach to ensure strong isolation guarantees for applications with different criticalities. This work focused on a methodology and tool for generating software wrappers for hardware components

that enforce (at run-time) the required behavior rather than focusing on the CPU scheduling policies.

Chapter 6

Conclusions

In this dissertation, we have presented what is to our knowledge the first practical side-by-side implementation and evaluation of mixed-criticality real-time task scheduling. Our evaluation considers for periodic and sporadic mixed-criticality tasks on uniprocessor or distributed systems, under a mixed-criticality scheduling model that is common to all if the evaluated approaches. To make a fair evaluation of mixed-criticality scheduling, we also addressed some previously open issues and proposed modifications to improve correctness and schedulability of the approaches, including zero slack scheduling, fixed task priority adaptive mixed criticality (FTP-AMC) and fixed job priority adaptive mixed criticality (FJP-AMC) approaches.

For zero-slack scheduling, we have offered refinements to the scheduling algorithm and the calculation of zero-slack instants. In particular, we have characterized a scenario in which a deadline miss of a lower-criticality task could affect scheduling guarantees for a higher criticality task, and provide a simple priority demotion rule to address that problem. We also propose a new worst case phasing condition for zero-slack scheduling and show its correctness, provide an analysis of how much interference a task can suffer from other tasks, and develop a new algorithm for calculating zero-slack instants based on that analysis.

For the fixed task priority adaptive mixed criticality approach, we have improved the original analysis from Baruah et al. [19] and extended the existing analysis to more than 2 criticality levels. Our simulation results demonstrated our new analysis provided tighter response time bounds and thus increased task schedulability.

We also developed a new method to calculate the worst case job arrival pattern to be used in conjunction with the fixed job priority adaptive mixed criticality scheduling approaches such as Li's algorithm [67] or PLRS [50]. The original worst case job arrival pattern calculation method published by Li et al. [67] was based on the *load* of a task set. However, as we have shown in our experiments, that method may perform very poorly in terms of schedulability.

We conducted simulations to examine how the different mixed-criticality scheduling methods may impact task schedulability on uniprocessor systems and distributed systems with end-to-end tasks. Our results showed that the FJP-AMC approach together with our worst case job arrival pattern calculation performs best in schedulability among all the mixed-criticality scheduling approaches we have evaluated both on uniprocessor and distributed systems. However, especially in the distributed end-to-end cases, FJP-AMC did not dominate in terms of schedulability; in other words, there were some task sets that were schedulable under other approaches but not under FJP-AMC.

In addition, we have described the design and implementation of MCFlow, a novel middleware designed specifically to support subtask parallelization for distributed mixed-criticality real-time applications on multi-core platforms. MCFlow provides a simple but flexible component-based development model in which an application developer does not need to program networking or data synchronization semantics directly, but rather uses a deployment tool to specify how components are connected and to specify their criticality and other real-time constraints. MCFlow can optimize communication between components based on their location and connection topology and whether each connection is intra-core, inter-core, or across a network.

Results of the experiments presented in Section 2.3 show that MCFlow performs comparably to TAO when only one core is used, and outperforms the widely used TAO real-time object request broker when multiple cores are involved for traditional real-time applications. MCFlow also implemented mechanisms to support various mixed-criticality scheduling approaches including zero-slack scheduling, period transformation, FTP-AMC and PRLS. Our empirical evaluation showed that PLRS imposed the heaviest overhead among those mechanisms we implemented. Nevertheless,

compared to traditional fixed-priority scheduling, PLRS only imposed 0.3% additional overhead, which demonstrates its viability.

Our micro-benchmarks also showed that the run-time overhead of those mixed-criticality scheduling approaches was strongly influenced by the number of preemptions and the number tasks to suspend/resume in a schedule. Because different scheduling approaches likely likely will produce different run-time schedules, there is no absolute guarantee which scheduling approach could result to the fewest number of preemptions, task suspensions or resumptions. However, FTP-SMC relies only on priority, and no extra mechanism for task suspension/resumption is required; thus, it is expected to have the lightest overhead in general. Therefore, we recommend that system developers consider adopting FTP-SMC if the target task set can be schedulable with the approach. For cases where more stringent CPU utilization bounds are required, PLRS with our new worst case arrival pattern calculation could be a good solution.

As future work, we plan to extend MCFlow's support for additional communication protocols and configuration parameters. Adding admission control for teams of dependent subtasks, and the ability to remap the subtask topology dynamically at run-time without performing thread migrations, are also useful potential extensions to MCFlow as future work. On the mixed-criticality scheduling side, the current PLRS scheduling approach does not offer a means to deal with dependent tasks with mutually exclusive synchronization constraints. We are considering how to extend the worst case arrival pattern calculation as well as the PLRS run-time scheduling algorithm to accommodate this sort of constraint.

Appendices

Appendix A

Original Zero Slack Instant Calculation Algorithms

Algorithm A.1 Compute Final Zero-Slack Instants ($t = D_i$)

```
∀i Z_i^1 ← 0
repeat
  ∀i Z_i^0 ← Z_i^1
  for all i in taskset do
    V_i^n ← GetSlackVector(i, Γ_i^n)
    V_i^c ← GetSlackVector(i, Γ_i^c)
    Z_i^1 ← GetSlackZeroInstant(i, V_i^c, V_i^n, t)
  end for
until ∀i Z_i^0 = Z_i^1
return Z_i^1
```

Algorithm A.2 GetSlackZeroInstant(i, V^c, V^n, t) : Calculate Instant of Slack $= 0$ before time t

```
C_i^c ← C_i^0 ; C_i^n ← 0
repeat
  t_1 ← StartOfTrailingSlack(i, C_i^c, V^c)
  if t_1 ≥ 0 and t_1 ≤ t then
    k_u ← SlackUpToInstant(V^n, t_1) − C_i^n
    k_u ← max(min(k_u, C_i^c), 0)
    C_i^c ← C_i^c − k_u
    C_i^n ← C_i^n + k_u
  else
    k_u ← 0
  end if
until k_u = 0
return t_1
```

Algorithm A.3 GetSlackVector$(i, \Gamma, t = T_i)$: Slack Vector Calculation

```
index ← 0 ; C_i^v ← 0
repeat
  R_current ← C_i^v ; b ← 0
  repeat
    R_previous ← R_current
    R_current ← C_i^v + Σ_{j∈Γ} ⌈R_previous / T_j⌉ C_j^e
    b ← t ; I_m ← i
    for (j ∈ Γ) do
      A ← ⌈R_previous / T_j⌉ T_j
      if A < b then
        b ← A ; I_m ← j
      end if
    end for
    if R_previous = R_current and R_current = b then
      R_current ← R_current + C_{I_m}^e
    end if
  until R_previous = R_current or ~~R_current ≤ t~~ R_current ≥ t
  if R_current < t
    V_i[index].slack ← min(b, t) − R_current
    V_i[index].time ← R_current
    C_i^v ← C_i^v + min(b, t) − R_current
    index + +
  end if
until R_current ≥ t
return V_1
```

References

[1] Boost C++ Library. http://www.boost.org.

[2] d{SPACE} Systems. http://www.dspace.de/ww/en/inc/home.cfm.

[3] JTC1/SC22/WG21. Draft Technical Report on C++ Library Extensions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.

[4] MATLAB - The Language Of Technical Computing. http://www.mathworks.com/products/matlab/.

[5] Open{S}ees, a software framework for developing applications to simulate the performance of structural and geotechnical systems subjected to earthquakes. http://opensees.berkeley.edu/.

[6] Suites of Earthquake Ground Motions for Analysis of Steel Moment Frame Structures: 10 pairs of horizontal ground motions for Los Angeles with a probability of exceedence of 10% in 50 years. http://nisee.berkeley.edu/data/strong_motion/sacsteel/.

[7] The Object Management Group. 2005. UML Profile for Modeling and Analysis of Real-Time and Embedded systems. OMG Request For Proposals, real-time/05-02-06. http://www.omg.org/cgi-bin/doc?real-time/05-02-06.pdf.

[8] Working draft for the C++ language. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf.

[9] x{PC} Target 4.1, Perform real-time rapid prototyping and hardware-in-the-loop simulation using {PC} hardware. http://www.mathworks.com/products/xpctarget/.

[10] *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies {&} Systems, May 3-7, 1999, The Town {&} Country Resort Hotel, San Diego, California, USA*. USENIX, 1999.

[11] *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*. IEEE Computer Society, 2009.

[12] Kunal Agrawal and C.E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. IEEE, July 2006.

[13] M Ahmadizadeh, G Mosqueda, and A M Reinhorn. Compensation of actuator delay and dynamics for real-time hybrid structural simulation. *Earthquake Engineering and Structural Dynamics*, 37(11):21–42, 2008.

[14] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. Porting decision tree algorithms to multicore using FastFlow. In José L Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Proceedings of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, volume 6321 of *LNCS*, pages 7–23, Barcelona, Spain, September 2010. Springer.

[15] Saman Amarasinghe, Michael l. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M Rabbah, and William Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2-3):261–278, June 2005.

[16] James H. Anderson, Sanjoy Baruah, and Björn B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[17] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report November, University of York, York, 1991.

[18] Sanjoy Baruah and Alan Burns. Implementing mixed criticality systems in Ada. *Reliable Software Technologies-Ada-Europe 2011*, 2011.

[19] Sanjoy Baruah, Alan Burns, and R. I. Davis. Response-Time Analysis for Mixed Criticality Systems. pages 34–43, 2011.

[20] Sanjoy Baruah, Haohan Li, and Chapel Hill. Towards the design of certifiable mixed-criticality systems. pages 13–22, 2010.

[21] Sanjoy Baruah and Steve Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. pages 147–155, July 2008.

[22] Andrea Bastoni and B Brandenburg. Cache-Related Preemption and Migration Delays : Empirical Approximation and Impact on Schedulability. *Proceedings of Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44, 2010.

[23] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. pages 14–24. IEEE, November 2010.

[24] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[25] Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, May 2005.

[26] A Blakeborough, M S Williams, A P Darby, and D M Williams. The Development of Real-Time Substructure Testing. In *Philosophical Transactions: Mathematical, Physical and Engineering Sciences: Dynamic Testing of Structures*, volume 359, pages 1869–1891. The Royal Society, 2001.

[27] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995.

[28] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[29] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[30] William J Bolosky and Michael L Scott. False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, pages 53–71, San Diego, California, 1993. USENIX Association.

[31] Alan Burns. Timing Faults and Mixed Criticality Systems. In Lloyd and Jones, editors, *Dependable and Historic Computing*, pages 147–166. Springer, lncs 6875 edition, 2011.

[32] J Carrion and B F Spencer. Model-based strategies for real-time hybrid testing. Technical Report NSEL-006, University of Illinois at Urbana-Champaign, 2007.

[33] J E Carrion and Spencer B.F. A model based delay compensation approach for real time hybrid testing. In *Fourth International Conference on Earthquake Engineering*, Taipei,Taiwan, 2006.

[34] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[35] C Chen and J M Ricles. Development of Direct Integration Algorithms for Structural Dynamics Using Discrete Control Theory. In *Journal of Engineering Mechanics*, volume 134, pages 676–683, 2008.

[36] A P Darby, A Blakeborough, and M S Williams. Improved control algorithm for real-time substructure testing. In *Earthquake Engineering and Structural Dynamics*, volume 30, pages 431–448, 2001.

[37] A P Darby, M S Williams, and A Blakeborough. Stability and delay compensation for real-time substructure testing. In *Journal of Engineering Mechanics*, volume 128, pages 1276–1284. ACSE, 2002.

[38] Robert Davis and A.J. Wellings. Dual Priority Scheduling. page 100, 1995.

[39] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. pages 291–300, December 2009.

[40] Romulo Silva De Oliveira and Joni Da Silva Fraga. Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems. *Journal of Systems Architecture*, 46(11):991–1004, 2000.

[41] Clarence W DeSilva. *Control Sensors and Actuators*. Prentice Hall, Inc., 1989.

[42] S J Dyke, B F Spencer, P Quast, and M K Sain. Role of control-structure interaction in protective system design. In *Journal of Engineering Mechanics*, volume 121, pages 322–338. ACSE, 1995.

[43] D Faggioli, A Mancina, F Checconi, and G Lipari. Design and implementation of a posix compliant sporadic server for the Linux kernel. In *10th realtime Linux workshop*, pages 65–80, 2008.

[44] N. Fisher, T.P. Baker, and S. Baruah. Algorithms for Determining the Demand-Based Load of a Sporadic Task System. *RTCSA*, pages 135–146, 2006.

[45] J.J.G. Garcia and M.G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems*, pages 124–132. IEEE Comput. Soc. Press, 1995.

[46] Christopher D Gill, Ron K Cytron, and Douglas C Schmidt. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. In *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 2002.

[47] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[48] D O C Group. nORB - Special Purpose Middleware for Networked Embedded Systems, 2005.

[49] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficent Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. pages 13–23, 2011.

[50] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. pages 13–23, November 2011.

[51] Timothy H Harrison, David L. Levine, and Douglas C Schmidt. The design and performance of a real-time CORBA event service. *ACM SIGPLAN Notices*, 32(10):184–200, October 1997.

[52] T Horiuchi, M Inoue, T Konno, and Namita Y. Real-time hybrid experimental system with actuator delay compensation and its application to a piping system with energy absorber. In *Earthquake Engineering and Structural Dynamics*, volume 28, pages 1121–1141, 1999.

[53] Huang-ming Huang, Chenyang Lu, and Christopher Gill. Implementation and Evaluation of Mixed-Criticality Scheduling Algorithms for Periodic Tasks. Technical report, Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, 2011.

[54] Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. Cyber-physical systems for real-time hybrid structural testing. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '10*, number 2, page 69, New York, New York, USA, 2010. ACM Press.

[55] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/.

[56] Institute for Software Integrated Systems. Component Synthesis using Model Integrated Computing (CoSMIC). www.dre.vanderbilt.edu/cosmic.

[57] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/.

[58] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/.

[59] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.

[60] Ben Kao and Hector Garcia-Molina. Deadline assignment in a distributed soft real-time system. pages 428–437, 1993.

[61] Ben Kao and Hector Garcia-Molina. Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks. pages 172–181, 1994.

[62] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Mixed-Criticality Task Synchronization in Zero-Slack Scheduling. pages 47—-56, 2011.

[63] Karthik Lakshmanan, Dionisio de Niz, Ragunathan Rajkumar, and Gabriel Moreno. Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems. pages 169–178, 2010.

[64] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. pages 201–209, 1990.

[65] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. pages 166–171, 1989.

[66] Haohan Li and Sanjoy Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. pages 183–192, 2010.

[67] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 99–108, New York, New York, USA, 2010. ACM Press.

[68] C. L. Liu and James W Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM (JACM)*, (1):46–61, 1973.

[69] Jane W. S. W. Liu. *Real-time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[70] S A Mahin and P B Shing. Pseudodynamic method for seismic testing. *Journal of Structural Engineering*, 111(7):1482–1503, 1985.

[71] S A Mahin, P B Shing, C R Thewalt, and R D Hanson. Pseudodynamic test method. {C}urrent status and future directions. *Journal of Structural Engineering*, 115(8):2113–2128, 1989.

[72] P. Mejia-Alvarez, R. Melhem, and D. Mosse. An incremental approach to scheduling during overloads in real-time systems. Number 1, pages 283–293.

[73] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy Baruah, and John A. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, pages 1864–1871, 2010.

[74] M Nakashima and N Masaoka. Real time on-line test for MDOF systems. In *Earthquake Engineering and Structural Dynamics*, volume 28, pages 393–420, 1999.

[75] S A Neild, D P Stoten, D Drury, and D J Wagg. Control issues relating to real-time substructuring experiments using a shake table. In *Earthquake Engineering and Structural Dynamics*, volume 34, pages 1171–1192, 2005.

[76] Object Management Group. Real Time-CORBA Specification, Version 1.2, November 2003.

[77] Shuichi Oikawa and Ragunathan Rajkumar. Linux/RK: A Portable Resource Kernel in Linux, 1998.

[78] Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. Handling mixed-criticality in SoC-based real-time embedded systems. In *Proceedings of the seventh ACM international conference on Embedded software - EMSOFT '09*, page 235, New York, New York, USA, 2009. ACM Press.

[79] Irfan Pyarali, Carlos O'Ryan, Douglas C Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha S Gokhale. Applying optimization principle patterns to design real-time ORBs. In *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems*, pages 145–160. USENIX, 1999.

[80] Irfan Pyarali, D.C. Schmidt, and R.K. Cytron. Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, July 2003.

[81] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[82] Steven Rostedt and Darren V. Hart. Internals of the RT Patch. In *Proceedings of the Linux symposium*, 2007.

[83] Douglas C Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the 6ˆ{th} USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[84] Douglas C Schmidt, M Stal, H Rohnert, and F Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[85] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for Some Practical Problems. pages 181–191, 1986.

[86] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, pages 1175–1185, 1990.

[87] P B Shing, Z Wei, R Y Jung, and E Stauffer. NEES fast hybrid test system at the University of Colorado. In *13th World Conference on Earthquake Engineering*, Vancouver, Canada, 2004.

[88] Mark Stanovich, Theodore P Baker, and Michael Gonzalez Harbour. Defects of the POSIX Sporadic Server and How to Correct Them. pages 35–45, April 2010.

[89] Venkita Subramonian, Liang-Jui Shen, Christopher Gill, and Nanbor Wang. The Design and Performance of Configurable Component Middleware for Distributed Real-Time and Embedded Systems. pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[90] Venkita Subramonian, Guoliang Xing, Christopher Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. 2004.

[91] Jun Sun. *Fixed-priority end-to-end scheduling in distributed real-time systems*. PhD thesis, University of Illinois, Urbana-Champaign, USA, 1997.

[92] Terry Tidwell, Xiuyu Gao, Huang-Ming Huang, Chenyang Lu, Shirley Dyke, and Christopher Gill. Towards configurable real-time hybrid structural testing: a cyber-physical system approach. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 37–44, Tokyo, Japan, March 2009. IEEE, IEEE.

[93] K. W. Tindell, A. Burns, and a. J. Wellings. Allocating Hard Real Time Tasks. *Real-Time Systems*, 4(2):145–165, June 1992.

[94] K. W. Tindell, Alan Burns, and A.J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[95] Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. pages 239–243, December 2007.

[96] Y C Wang and K J Lin. The implementation of Hierarchical Schedulers in the RED-Linux Scheduling Framework. page 231, 2000.

[97] Yuanfang Zhang, Christopher Gill, and Chenyang Lu. Reconfigurable Real-Time Middleware for Distributed Cyber-Physical Systems with Aperiodic Events. pages 581–588, Washington, DC, USA, 2008. IEEE Computer Society.

[98] Yuanfang Zhang, Christopher Gill, and Chenyang Lu. Real-time performance and middleware for multiprocessor and multicore Linux platforms. In *15th IEEE*

International Conference on Embedded and Real-Time Computing Systems and Applications, number 314, pages 437–446. IEEE, August 2009.

[99] J Zhao, C French, C Shield, and T Posbergh. Considerations for the development of real-time dynamic testing using servo-hydraulic actuation. In *Earthquake Engineering and Structural Dynamics*, volume 32, pages 1773–1794, 2003.

[100] Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems. pages 161–170, 2007.

Vita

Huang-Ming Huang

Date of Birth March 6, 1970

Place of Birth Taipei, Taiwan, ROC

Degrees
Ph.D. Computer Science, August 2012
M.S. Computer Science, May 2010
M.S. Applied Mathematics, June 1994
B.S. Mechanical Engineering, June 1992

Publications

Huang-Ming Huang, Christopher D. Gill, and Chenyang Lu. MCFlow: a real-time multi-core aware middleware for dependent task graphs. To appear in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, Seoul, Korea, August 2012.

Huang-Ming Huang, Christopher Gill, Chenyang Lu. Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*, pages 23–32, Beijing, China, April 2012

Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. Cyber-physical systems for real-time hybrid structural testing: a case study. In *ICCPS '10: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems,* pages 69-78, New York, NY, USA, 2010. ACM.

Huang-Ming Huang and Christopher D. Gill. Verification of component-based distributed real- time systems. Technical Report WUCSE-2008-12, Washington University in St. Louis, 2008.

Huang-Ming Huang and Christopher D. Gill. Design and performance of a fault-tolerant real- time CORBA event service. In *18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, Dresden, Germany, July 5-7, 2006, pp. 33-42.

Huang-Ming Huang, Chang-Biau Yang, and Kuo-Tsung Tseng. Broadcasting on uni-directional hypercubes and its applications. *Journal of Information Science and Engineering*, Vol.19 No.2, pp.183-203 (March 2003).

Terry Tidwell, Xiuyu Gao, Huang-Ming Huang, Chenyang Lu, Shirley Dyke, and Christopher D. Gill. Towards configurable real-time hybrid structural testing: A cyber-physical system approach. In *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '09)*, Tokyo, Japan, March 17-20, 2009.

Xiaorui Wang, Ming Chen, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu, and Christopher D. Gill. Control-based adaptive middleware for real-time image transmission over bandwidth-constrained networks. In *IEEE Transactions on Parallel and Distributed Systems*, 19(6): 779-793, June 2008.

Xiaorui Wang, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu, and Christopher D. Gill. CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, May 2004

Christopher D. Gill, Venkita Subramonian, Jeff Parsons, Huang-Ming Huang, Stephen Torri, Douglas Niehaus, and Douglas Stuart. ORB middleware evolution for networked embedded systems. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS '03)*, Guadalajara Mexico, January 15-17, 2003, pp. 169-176.

Venkita Subramonian, Huang-Ming Huang, Seema Datar, and Chenyang Lu. Priority scheduling in TinyOS - a case study. Technical report, Washington University in St. Louis, 2002.

August 2012

MCFlow: Mixed-Criticality Middleware, Huang, Ph.D. 2012