River Publishers Series in Information Science and Technology

Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing

Leandro Soares Indrusiak, Piotr Dziurzanski and
Amit Kumar Singh

River Publishers

Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing

RIVER PUBLISHERS SERIES IN INFORMATION SCIENCE AND TECHNOLOGY

The "River Publishers Series in Information Science and Technology" covers research which ushers the 21st Century into an Internet and multimedia era. Multimedia means the theory and application of filtering, coding, estimating, analyzing, detecting and recognizing, synthesizing, classifying, recording, and reproducing signals by digital and/or analog devices or techniques, while the scope of "signal" includes audio, video, speech, image, musical, multimedia, data/content, geophysical, sonar/radar, bio/medical, sensation, etc. Networking suggests transportation of such multimedia contents among nodes in communication and/or computer networks, to facilitate the ultimate Internet.

Theory, technologies, protocols and standards, applications/services, practice and implementation of wired/wireless networking are all within the scope of this series. Based on network and communication science, we further extend the scope for 21st Century life through the knowledge in robotics, machine learning, embedded systems, cognitive science, pattern recognition, quantum/biological/molecular computation and information processing, biology, ecology, social science and economics, user behaviors and interface, and applications to health and society advance.

Books published in the series include research monographs, edited volumes, handbooks and textbooks. The books provide professionals, researchers, educators, and advanced students in the field with an invaluable insight into the latest research and developments.

Topics covered in the series include, but are by no means restricted to the following:

- Communication/Computer Networking Technologies and Applications
- Queuing Theory
- Optimization
- Operation Research
- Stochastic Processes
- Information Theory
- Multimedia/Speech/Video Processing
- Computation and Information Processing
- Machine Intelligence
- Cognitive Science and Brian Science
- Embedded Systems
- Computer Architectures
- Reconfigurable Computing
- Cyber Security

For a list of other books in this series, www.riverpublishers.com

Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing

Leandro Soares Indrusiak

Piotr Dziurzanski

Amit Kumar Singh

River Publishers

Published, sold and distributed by:
River Publishers
Alsbjergvej 10
9260 Gistrup
Denmark

River Publishers
Lange Geer 44
2611 PW Delft
The Netherlands

Tel.: +45369953197
www.riverpublishers.com

ISBN: 978-87-93519-08-4 (Hardback)
978-87-93519-07-7 (Ebook)

Cover design by Andrea Monteiro and Leandro Soares Indrusiak

Contents

Preface

The availability of many-core computing platforms enables a wide variety of technical solutions for systems across the embedded, high-performance and cloud computing domains. However, large scale many-core systems are notoriously hard to optimise. Choices regarding resource allocation alone can account for wide variability in timeliness and energy dissipation (up to several orders of magnitude). This book covers dynamic resource allocation heuristics for many-core systems, aiming to provide appropriate guarantees on performance and energy efficiency. It addresses different types of systems, aiming to harmonise the approaches to dynamic allocation across the complete spectrum between systems with little flexibility and strict performance guarantees all the way to highly flexible systems with soft performance guarantees.

Resource allocation is one of the most complex problems in large multiprocessor and distributed systems, and in general it is considered NP-hard. The theoretical evidence shows that the number of possible allocations of application tasks grows exponentially with the increase of the number of processing cores. The empirical evidence points in the same direction, with case studies showing that for a realistic multiprocessor embedded system (40–60 application components, 15–30 processing cores) a well-tuned search algorithm had to statically evaluate hundreds of thousands of distinct allocations before it finds one that meets the systems performance requirements.

In this book, we argue that the only way to cope with such complexity is to design systems that are capable to explore the allocation space during runtime. This is commonly done in cloud and high-performance computing, mainly because the workload of such systems cannot be accurately predicted in advance and static allocations are thus impossible. In embedded systems, the workload is more predictable in terms of its worst-case behaviour, but static allocations that take such characterisation into account tend to produce underutilised platforms. We therefore set the scene for dynamic resource allocation mechanisms by identifying and evaluating allocation heuristics that can be used to provide different levels of performance guarantees, and that cope with different levels of dynamism on the application workload.

The book starts with a description of the common practices and challenges in dynamic resource allocation, highlighting the peculiarities of each domain: embedded, HPC and cloud computing. Then, each of the challenges is addressed in detail within the following chapters, which are largely self-contained and therefore can be read in any order. To facilitate understanding, all of them follow the same structure: a specific challenge is motivated and the respective problem is precisely formulated; a detailed description of a solution to the problem is then given, followed by experimental work showing quantitative evidence of the strengths and weaknesses of that solution; related work is reviewed; and a summary of the chapter is given at the end.

The technical work that resulted in this book was done within the frame of the DreamCloud project, and the project website[1] makes available a number of reference implementations of the models and heuristics described here. Updates to this book will also be made available on that website.

Leandro Soares Indrusiak,
Piotr Dziurzanski,
and Amit Kumar Singh,

York, summer of 2016.

[1] http://www.dreamcloud-project.org

Acknowledgements

The research work that resulted in this book was done within the frame of the DreamCloud project, funded by the European Commission under Framework Programme 7 (Ref. 611411). The authors would like to acknowledge and thank the commission for the funding, as well as the project officers and reviewers for their suggestions and feedback on the project outcomes.

The authors would like to thank Hashan Roshantha Mendis, whose work provided most of the foundations and the experimental results reported in Chapter 6.

The authors would also like to thank all the members of the DreamCloud consortium, particularly Scott Hansen, Björn Saballus, Devendra Rai, Manuel Selva, Abdoulaye Gamatié, Gilles Sassatelli, Luciano Copello Ost, Leonardo Zordan, David Novo, Alexey Cheptsov, Dennis Hoppe, Thomas Baumann, Fridtjof Siebert, Malek Ben Salen, Raj Patel, José Miguel Montañana and Neil Audsley.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ACPI	Advanced Configuration and Power Interface
ALU	Arithmetic Logic Unit
AMIGO	Approximate M-constraint Integral Gain Optimization
AT	Arrival Time
AUTOSAR	Automotive Open System Architecture
BCET	Best Case Execution Time
CL	Closed Loop
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DBC	Deadline and Budget Constraints
DSE	Design Space Exploration
DSP	Digital Signal Processing
DVFS	Dynamic Voltage and Frequency Scaling
ECG	Electrocardiogram
ECU	Electronic Control Unit
EDF	Early Deadline First
ET	Execution Time
FCPS	Fixing Cores Power States
FIFO	First In First Out
flit	Flow control digIT
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTPS	Fixing Tasks Power States
GA	Genetic Algorithm
GoP	Group of Pictures
HLRS	High Performance Computing Center Stuttgart
HP	HPC Platform
HPC	High-Performance Computing
HRT	Hard Real-Time
IA	Interval Algebra

IQR	Inter-Quartile Range
MMKP	Multi-Choice Knapsack Problem
MPSoC	Multiprocessor System-on-Chip
NBA	Non-profiling Based Approach
NoC	Network-on-Chip
OL	Open Loop
OS	Operating System
P	Proportional
PBA	Profiling Based Approach
PE	Processing Element
PG	Platform Graph
PI	Proportional-Integral
PID	Proportional-Integral-Derivative
PID-AC	PID-based Admission Control
PMF	Probability Mass Function
PS	Pheromone Signalling
QoS	Quality of Service
RM	Resource Manager
RMA	Rotating Mapping Algorithm
RTA	Response Time Analysis
RTM	Real-Time Manager
RTS	Real-Time Scheduling
SDF	Synchronous Dataflow
SoC	System-on-Chip
ST	Spanning Tree
TG	Task Graph
TLM	Transaction-Level Modelling
ValOpt	Value Optimization
VC	Value Curve
VM	Virtual Machine
VS	Voltage Scaling
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time

1

Introduction

The availability of highly parallel computing platforms based on multi and manycore processors enables a wide variety of technical solutions for systems across the embedded and high-performance computing domains. However, large scale manycore systems are notoriously hard to design and manage, and choices regarding resource allocation alone can account for wide variability in timeliness and energy dissipation, up to several orders of magnitude. For example, the allocation of many computation-centric jobs to the same processing core, or communication-intensive jobs to cores linked by a low bandwidth interconnect, can significantly impair system performance specially in applications with many dependencies between jobs.

Techniques to allocate computation and communication workloads onto processor platforms have been studied since the early days of computing. However, this problem has become significantly harder because of **scale** and **dynamicity**: compute platforms now integrate hundreds to thousands of processing cores, running complex and dynamic applications that make it difficult foresee the amount of load they can impose to those platforms.

Elementary combinatorics provides us with evidence of the problem of **scale**. For a simple formulation of the problem of allocating jobs to processors (one-to-one allocation), one can see that the number of allocations grows with the factorial of the number of jobs and processors. For example, a system with 4 jobs and 4 processing cores can have $P(4, 4) = 24$ possible allocations, but simply by doubling the number of jobs and cores the number of allocations becomes $P(8, 8) = 40320$ (where $P(n, k)$ denotes the k-permutations of n). The empirical evidence points in the same direction, as it can be seen in [110] that for realistic manycore embedded systems (40–60 jobs, 15–30 processing cores) a well-tuned search algorithm had to statically evaluate hundreds of thousands of distinct allocations before it finds one that meets the systems performance requirements.

To cope with **dynamicity**, a dynamic approach to resource management is the most obvious choice, aiming to dynamically learn and react to changes to the load characteristics and to the underlying compute platform. The baseline, which is a static allocation decided before deployment based on the (nearly) complete knowledge about the load and the platform, is no longer viable. For example, static resource allocation in high-performance computing (HPC) has often been referred as a significant cause of low utilization of servers, which results in cost increases on hardware and energy [17]. Static allocation is also commonly used by aerospace and automotive industries to provide worst case performance guarantees that are required by certification authorities. However, it is well known that such an approach usually leads to under-utilised computing and communication resources at run-time [113].

The problems of scale and dynamicity are also made harder with the increasing density of computing and communication resources. The definition of density used here is not necessarily spatial, but rather on connectivity (i.e., dense graph). In densely connected systems, a resource allocation algorithm may have to make decisions very often due to the system dynamics, and may have to consider dozens or hundreds of potential allocation possibilities at each decision point (i.e., which processor should execute each job, which communication links should be used when those jobs exchange data). Furthermore, such algorithms have to work in a distributed way due to the difficulty to obtain the up-to-date state of the whole system. And despite such levels of complexity, the algorithms themselves are also subject to tight constraints in performance and energy. It is then evident that optimal resource allocation algorithms cannot cope with this type of problem, and that lightweight heuristic solutions are needed.

This book is therefore concerned with the kinds of resource allocation heuristics that can cover different levels of dynamicity, while coping with the scale and complexity of high-density manycore platforms.

1.1 Application Domains

The level of dynamicity of a system denotes how often it changes its characteristics. In this book, we are concerned with resource allocation, so dynamicity means how much variation can be found on the system workload (e.g., arrival patterns, computation and communication requirements, value to the end-user) and on the underlying compute platform (e.g., degradation or lost of performance due to faults, increase in capacity due to upgrades). Different application domains can be characterised by their typical levels of dynamicity.

For example, deeply embedded systems such as those in automotive, aerospace and medical domains have low dynamicity, and often their entire functionality and behaviour is known at design time, prior to deployment. The low dynamicity makes the performance of such systems easier to predict, and therefore guarantees regarding timeliness can be made (e.g., ECG signal of a complete cardiac cycle will be processed in less than 10 ms). Such guarantees are often enforced by means of resource reservation and isolation, which can lead to very low levels of resource utilisation: a processing core can be exclusively allocated to a given job for the sake of performance predictability, but that job only needs the core to its full capacity for a limited period of its lifetime, leaving it subutilised for the rest of the time.

On the other hand, HPC and cloud computing have high dynamicity due to the wide variety of workloads they have to handle. That makes it harder to make performance guarantees, because one never knows what comes next. And due to the cost of deploying and maintaining such platforms, they are often only viable if operated at saturation point, with nearly 100% utilisation, which undermines performance guarantees even further by making nearly impossible to rely on resource reservation or isolation.

Figure 1.1 below shows both domains, embedded and HPC/cloud over the dimensions of dynamicity, typical resource utilisation and the ability to sustain performance guarantees. State-of-the-art resource allocation in the embedded domain is static, relying on the low dynamicity of those systems and producing allocations that can be derived at design time and used for the whole lifetime of the system, while ensuring the performance requirements are met even in worst case scenario. For HPC and cloud, the resource allocation is completely dynamic and often based on instantaneous metrics such as order of arrival of jobs and current utilisation of cores, which can certainly keep the platforms running at saturation point but cannot offer any performance guarantees.

Recently, the dichotomy described above became less visible. Embedded systems are becoming increasingly complex, having to cope with dynamic workloads, and using less predictable platforms (i.e., multi-level caches, speculative execution), while still having to fulfil strict performance requisites. HPC and cloud computing, in turn, critically need to address fundamental problems in energy efficiency and performance predictability, as they become more widespread and critical to our daily lives. This points to the importance

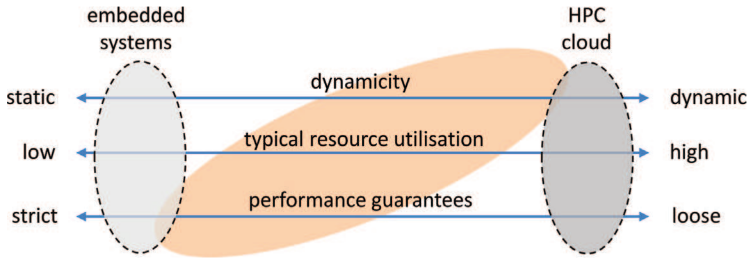


Figure 1.1 Application domains and their characteristics with regard to dynamicity, resource utilisation and performance predictability.

of the areas in the central part of Figure 1.1, which represents increasingly dynamic embedded systems and predictable HPC and cloud systems.

The goal of this book is to identify and present resource allocation heuristics that can be used to achieve different levels of performance guarantees, and that can cope with different levels of dynamicity of the application workload.

1.2 Related Work

The problem of allocating tasks to platform elements is a classic problem in multiprocessor and distributed systems. Most formulations of this problem cannot be solved in polynomial time, and many of them are equivalent to well known NP problems such as graph isomorphism [18] and the generalised assignment problem [58].

This problem was first addressed from the cluster/grid point of view, but more recently the fine-grained allocation of tasks within manycore processors has also received significant attention due to its critical impact on performance and energy dissipation. In the following subsections, we consider allocation mechanisms at both grid and manycore CPU level, and review the most significant trends and achievements in terms of guaranteed performance and energy efficiency.

1.2.1 Allocation Techniques for Guaranteed Performance

There are numerous multiprocessor scheduling and allocation techniques that are able to meet real-time constraints, each of them under a different set of assumptions. A very comprehensive survey is given by [41], covering techniques that can be applied both at the grid or many-core level, but all of them assume that the platform is homogeneous and tasks are independent (i.e., do not explicitly consider communication costs). Many of them also assume that the allocation is done statically, or do not take into account the overheads of dynamically allocating and migrating tasks (i.e., context saving and transferring). In [96], heterogeneous platforms are considered but communication costs and overheads are still not taken into account.

Significant research on resource reservation has been done, aiming to increase time-predictability of workflow execution over HPC platforms [90]. Many approaches use a priori workflow profiling and use estimation of task execution times and communication volumes to plan ahead which resources will be needed when tasks become ready to execute. Just like in static allocation, resource reservation policies significantly reduce the utilisation of HPC platforms. A reduction of 20–40% in the utilisation is not unusual [150].

Allocation and scheduling heuristics based on feedback control have been used in HPC systems [44–83], aiming to improve platform utilisation without sacrificing performance constraints. Most cases concentrate on controlling the admission and allocation of tasks over the platform based on a closed-loop approach that monitors utilisation of the platform as well as performance metrics such as task response times [54].

Many cloud-based and grid-based HPC systems use allocation and scheduling heuristics that take into account not only the timing constraints of the tasks but also their value (economic or otherwise). This problem been well-studied under the model of Deadline and Budget Constraints (DBC) [27], where each task or taskflow has a fixed deadline and a fixed budget. State-of-the-art allocation and scheduling techniques target objectives such as maximising the number of tasks completed within deadline and/or budget [139], maximising profit for platform provider [76] or minimising cost to users [130] while still ensuring deadlines. Several approaches to the DBC problem use market-inspired techniques to balance the rewards between platform providers and users [154]. A comprehensive survey given in [157] reviews several market-based allocation techniques supporting homogeneous or heterogeneous platforms, some of them supporting applications with dependent tasks modeled as DAGs.

At the many-core level, there are a few allocation techniques that take into account both the computation and communication performance guarantees. Such techniques are tailored for specific platforms e.g., many-cores based on Network-on-Chip (NoC). To guarantee timeliness, all state-of-the-art approaches rely on a static allocation of tasks and communication flows. In [6], a multi-criteria genetic algorithm is used to evolve task allocation templates over a NoC-based many-core aiming to reduce their average communication latency. The approach in [110] also used a genetic algorithm that could find an allocation that can meet hard real-time guarantees on end-to-end latency of sporadic tasks and communication flows over many-cores that use priority-preemptive arbitration. Stuijk [136] proposed a constructive heuristic to do static allocation of synchronous dataflow (SDF) application models [133], which constraint all tasks to read and write the same number of data tokens every time they execute. The allocation guarantees the timeliness of the application if the platform provides fixed-latency point-to-point connection between processing units. In [161], the same author relaxes some of the assumptions of SDF applications (i.e., allows for changes on token production and consumption rates during runtime) and proposes analytical methods to evaluate worst-case throughput and to find upper bounds for buffering for a given static allocation.

1.2.2 Allocation Techniques for Energy-efficiency

Most allocation techniques addressing energy efficiency operate at the many-core processor level, mainly because of the difficulties of dealing with energy-related metrics at larger system granularities.

Hu et al. [60] and Marcon et al. [88] estimate the energy consumption according to the volume of data exchanged by different application tasks over the interconnection network. Such approaches lack in accuracy as they do not take into account runtime effects such as network congestion or time-varying workloads. Thus, research approaches on energy-aware dynamic allocation techniques have been proposed.

In [129], an iterative hierarchical dynamic mapping approach aims to reduce energy consumption of the system while providing the required QoS. In such strategy, tasks are firstly grouped by assigning them to a system resource type (e.g., FPGA, DSP, ARM), according to performance constraints. Then, each task within a group is mapped, minimising the distance among them and reducing communication cost. Finally, the resulting mapping is checked, and if it does not meet the application requirements, a new iteration is required.

Chou and Marculescu [37] introduce an incremental dynamic mapping process approach, where processors connected to the NoC have multiple voltage levels, while the network has its own voltage and frequency domain. A global manager (OS-controlled mechanism) is responsible for finding a contiguous area to map an application, and for defining the position of the tasks within this area, as well. According to the authors, the strategy avoids the fragmentation of the system and aims to minimize communication energy consumption, which is calculated according to Ye et al. [155]. This work was extended in [36, 38], incorporating the user behaviour information in the mapping process. The user behaviour corresponds to the application profile data, including the application periodicity in the system and data volume transferred among tasks. For real applications considering the user behaviour information, the approach achieved around 60% energy savings compared to a random allocation scenario.

Holzenspies et al. [58] investigate a run-time spatial mapping technique with real-time requirements, considering streaming applications mapped onto heterogeneous MPSoCs. In the proposed work, the application remapping is determined according to information that is collected at design time (i.e., latency/throughput), aiming to satisfy the QoS requirements, as well as to optimize the resources usage and to minimise the energy consumption. A similar approach is proposed in Schranzhofer et al. [120], merging pre-computed template mappings (defined at design time) and online decisions that define newly arriving tasks to the processors at run-time. Compared to the static-mapping approaches, obtained results reveal that it is possible to achieve an average reduction on power dissipation of 40–45%, while keeping the introduced overhead to store the template mappings as low as 1 KB.

Another energy-aware approach is presented in Wilderman et al. [151]. This approach employs a heuristic that includes a Neighborhood metric inspired by rules from Cellular Automata, which allows decreasing the communication overhead and, consequently, the energy consumption imposed by dynamic applications. Lu et al. [85] propose a dynamic mapping algorithm, called Rotating Mapping Algorithm (RMA), which aims to reduce the overall traffic congestion (take in account the buffer space) and communication energy consumption of applications (reduction of transmission hops between tasks).

In turn, Mandelli et al. [87] propose a power-aware task mapping heuristic, which is validated using a NoC-based MPSoC described at a cycle-accurate level. The mapping heuristic is performed in a given processor of the system that executes a preemptive operating system. Due to the use of a low level description, accurate performance evaluation of several heuristics (execution time, latency, energy consumption) is supported. However, the scope of the work is limited to small systems configurations due to the long simulation time. In the previous works, only one task is assigned to each processing core. A multi-task dynamic mapping approach was proposed in [128]. Singh et al. [128] extends the work described in [32], which evaluates the power dissipation as the product of number of bits to be transferred and distance between source-destination pair.

Research in energy-efficient allocation for HPC and cloud systems is still incipient, with existing works addressing only the time and space fragmentation of resource utilisation at a very large granularity (server level), aiming to minimise energy by rearranging the load and freeing servers that are then turned off [12, 101].

1.3 Challenges

While the approaches mentioned in the previous section have presented sophisticated resource allocation approaches that can provide performance guarantees and/or improve energy efficiency, there are still challenges that require more advanced resource allocation approaches. The following subsections briefly describe some of those challenges, which are precisely the ones addressed in this book.

1.3.1 Load Representation

Load models are internal representations used by allocation algorithms to evaluate different allocation alternatives. Such models may use information that is available a priori about the load (such as job dependencies, communication volumes, worst case execution times), but can be also extended with information obtained during runtime (e.g., actual execution and communication times). In dynamic resource allocation, it is very challenging to define a load model that includes sufficient information about static and dynamic characteristics of the load, and that is lightweight enough to be used by allocation heuristics to quickly evaluate and compare alternative allocation possibilities during runtime.

Chapter 2 addresses this challenge and presents a load model based on an interval algebra, aiming to allow quickly compose the load of multiple computation and communication jobs (represented as series of time intervals), enabling the evaluation of the impact of resource allocation (and thus resource sharing) on system performance and timeliness.

1.3.2 Monitoring and Feedback

In large-scale systems, obtaining updated information about the load during runtime is not trivial. Often, such information only makes sense when coupled with information about the underlying computation and communication platform. Furthermore, the costs of monitoring and transferring all such data to the resource allocation mechanism is already prohibitive. The major challenge in such scenarios is then to define a sufficiently meaningful set of metrics to monitor, and to design algorithms that can make meaningful resource allocation decisions based on the changes on those metrics over time.

Feedback control algorithms have been used for decades to make decisions based on time-series data, so in Chapters 3 and 4 we describe possible uses of such closed-loop algorithms to support resource allocation. In Chapter 3, we show that they can be used to increase throughput and energy-efficiency in HPC and cloud workloads. In Chapter 4, on the other hand, we show that it can be used to efficiently perform admission control tasks, aiming to maximise system utilisation without jeopardising predictability in performance-sensitive HPC applications.

1.3.3 Allocation of Modal Applications

Allocation heuristics may have to guarantee hard real-time constraints to critical jobs. This is possible for applications that have been profiled a priori so their execution and communication patterns can be accurately represented by an accurate load model. Such applications will not be highly dynamic, and will exhibit modal behaviour, so that distinct modes of operation can be analysed at design time, so the dynamic allocation can be based on pre-defined alternatives (thus the number of allocation decisions during runtime is minimal).

To address such scenario, modal allocation heuristics can guarantee hard real-time constraints by allowing different different allocations for each operation mode while minimising the amount of remappings during mode transitions. Chapter 5 describes search-based heuristics that identify allocations that are optimised for specific operation modes, but also for coping with dynamic mode changes. It uses automotive applications and Network-on-Chip platforms as case studies, and shows that it is possible to guarantee hard real-time constraints during each of the system's modes as well as during transitions.

1.3.4 Distributed Allocation

In closed-loop systems, a centralised resource manager continuously receive feedback from the system so that it can have an up-to-date representation of its state. This usually comes with a significant communication overhead, specially in large-scale systems. Fully distributed approaches, on the other hand, offer higher scalability by relying on decision-making done by individual system components using only locally-available information. However, due to the lack of global knowledge, it is harder to achieve a reasonable level of performance predictability.

Chapter 6 presents a bioinspired approach based on the notion of swarm intelligence, aiming to support a fully distributed approach to load remapping. It can be used on its own or in conjunction with centralised approaches, aiming to fine-tune allocation decisions based on up-to-date local data. A case study based on multi-stream video processing over Network-on-Chip platforms shows the strengths and weaknesses of such approach.

1.3.5 Value-based Allocation

Many of the quality metrics associated to resource allocation in HPC and cloud computing are platform-specific. For instance, metrics that are often used to formulate optimisation objectives (such as job execution times, communication volumes and throughput) are not comparable across different computational platforms. There are other metrics, however, that are completely independent of the computational platform and relate instead to the requirements of the end-user. One of such metrics is the value of the completion of a job. This can be seen as a simple value, perhaps associated to a particular currency. More commonly, such value will be a function of time: the result of a job is very likely to lose value over time, and can even become worthless if it takes too long to be obtained.

Chapter 7 addresses resource allocation heuristics that are designed to optimise such time-varying notion of value. It presents approaches that can be configured to rely more or less on load models obtained in advance, and shows how much can be gained in value if these models are available.

2

Load and Resource Models

The efficient allocation of computational resources requires some understanding of the resources themselves and their availability, as well as the load that must be allocated to them. Possibly under different names, the concept of resource and load modelling is commonly used in embedded, high-performance and cloud computing. For example, workflow models in HPC and task graphs in embedded computing are common ways to represent application load, while platform and resource models are used to represent the processing, networking and storage capabilities of the computer systems that run those applications.

With the help of meaningful resource and load models, it is possible to evaluate the impact of different resource allocation techniques on the efficiency of resource usage and on application performance requirements. The more accurate the models, the better they can predict the performance of a computer system under a given load. On the other hand, dynamic and complex systems are harder to model accurately, so there is clearly a trade-off here.

In real-time embedded computing, for example, it is common practice to constrain the execution of software to sporadic and bounded time intervals, and to disable advanced features of microprocessors such as out-of-order execution and caching, aiming to simplify the system's behaviour and enable the creation of accurate load and resource models. At that level of accuracy, system designers can use such models to evaluate different resource allocation alternatives and identify the ones under which the system will never violate any of its performance guarantees, not even in a worst-case scenario.

Such practice, however, requires a complete knowledge of the system resources as well as the load to be allocated to them. In many embedded systems, and in the large majority of high-performance and cloud computing systems, that is not the case. Therefore, recent modeling approaches have ways to represent load and resources under different levels of uncertainty. Stochastic models of the arrival and execution times of application-specific load or of the availability of computational resources, for example, are now commonly used to characterise average-case system performance.

2.1 Related Work

In real-time systems, load models are often variations of the sporadic task model [42] or the time-triggered model [71], focusing explicitly on timing and on the repetitive nature of tasks (e.g., data from a sensor must be processed every 2 ms; a new gene sequencing job will be launched at least every millisecond) rather than the functional dependencies between them.

Dataflow application models are usually untimed, and different tasks are synchronised by the data flowing through the system. Dataflows are usually modelled through graphs that represent the functional dependencies between tasks and some information about the nature of the data transfer. Many different dataflow models exist [134], with different types of constraints on the execution of tasks and communication aiming to allow different types of analysis (e.g., statically schedulable, time predictable, bounded communication buffering). Many HPC workflows are also modelled as dependency graphs, often as directed acyclic graphs (DAGs) [144]. Such graphs can be annotated with estimations of execution time and communication volumes, which can be used to optimise resource allocation or implement resource reservation mechanisms [90]. Similarly, estimations of inter-arrival times, execution times and communication volumes can be modelled stochastically, allowing for a more general understanding of the characteristics of a given workload [51]. That approach can also be used to support the generation of synthetic application models that follow the specific characteristics of a realistic scenario.

Advanced workflow management systems augment HPC and scientific computing workflow models with execution semantics [86], allowing such workflows to be analysed in similar ways as in time-triggered and dataflow models mentioned above. Finer granularity models are also used in HPC [10, 111], where application load is represented as a series of computation and communication bursts (often obtained from execution traces), but such models are too complex to be analysed and therefore are used only to drive abstract simulation.

A number of application modelling approaches try to capture characteristics that are critical to specific domains. Within the automotive domain, a component-based software specification standard is established, called AUTOSAR (more in www.autosar.org). Within this standard, software components covering runnable entities can be defined by specifying interfaces, execution rates and timing constraints. However, AUTOSAR takes a conservative stand and does not allow the dynamic allocation of runnable entities to different computational units.

Advanced approaches in application modelling supports the creation of hybrid models, i.e., models created using different underlying rules.

Ptolemy [47] is a modelling and simulation framework supporting hybrid application modelling for embedded systems using actor-orientation (a flexible model for representing concurrent behaviour). It supports different types of time-triggered and dataflow modelling approaches, among others, and is amenable to extensions to specific domains.

2.2 Requirements

Within the scope of this book, we use a model of load and resources that can cater to both worst-case and average-case system performance. It supports complete and accurate description of a system's load and resources, but is also able to accommodate different levels of uncertainty by allowing stochastic descriptions of load.

We therefore define a load model using the notion of jobs, which should represent the different parts of an application and, more specifically, the load each of those parts imposes on platform resources. This is a general-purpose model, aiming to have constructs that are flexible enough to represent multiple types of application components. For example, a job could represent the execution of a software task over a CPU, the transmission of a stream of data over a network, or the dynamic reconfiguration of an FPGA device.

In order to model different types of applications, from embedded to HPC systems, such load models must be powerful enough to cover characteristics, e.g., functional properties, that are commonly found in such systems, as well as non-functional properties that can be used to evaluate the impact of different allocation mechanisms. In the subsections below, we present the requirements for such load modelling approach along four distinct categories: structure, temporal behaviour, resource constraints and load characterisation.

2.2.1 Requirements on Modelling Load Structure

Load models should be able to support multiple levels of abstraction, exposing more or less details of the application architecture according to the level of accuracy that is needed when evaluating the impact of a particular resource allocation mechanism. For instance, it may be useful to assume that all application jobs are completely independent, abstracting away their inter-communication, if the overheads due to data exchange are negligible. Therefore, the application structure denotes how an application can be broken in multiple jobs and how these jobs relate to each other.

Regarding the application structure, we list requirements for a load modelling approach, so that the model is powerful enough to represent the most common types of applications.

2.2.1.1 Singleton

Ability to model applications that are composed of a single job.

2.2.1.2 Independent jobs

Ability to model applications that are composed of an arbitrary number of jobs that do not depend on or communicate with other jobs. It is assumed that jobs constantly have access to all information they need.

2.2.1.3 Single-dependency jobs

Ability to model applications that are composed of an arbitrary number of jobs that can depend on one and only one other job. Therefore, the application model must explicitly have the notion of dependencies between jobs.

2.2.1.4 Communicating jobs

Ability to model applications that are composed of an arbitrary number of job pairs. Intuitively, each pair includes a computing job and a communication job, but the strict definition of a communicating job should be a pair of dependent jobs that cannot be allocated to the same resource type (see requirements on resourcing in Subsection 3.3). This enforces the notion that, in this kind of application, communication can only be performed once the respective computation has completed.

2.2.1.5 Multi-dependency jobs

Ability to model applications that are composed of an arbitrary number of computation jobs, each of them depending on an arbitrary number of communication jobs, and also initiating an arbitrary number of communication jobs. The structure of this type of model constrains the application in such a way that the communication jobs initiated by a given computation job must not depend on computation jobs that depend directly or indirectly on their initiator (no cyclic dependencies).

2.2.2 Requirements on Modelling Load Temporal Behaviour

The temporal behaviour of the load defines the release of application jobs, i.e., when a job can actually be executed over a resource. Behaviours can be generally classified in time-driven (requirements 2.2.2.1, 2.2.2.2 and 2.2.2.3 below) or event-driven (remaining requirements).

Regarding application temporal behaviour, we list the following requirements for a load modelling approach, so that it is powerful enough to represent the following types of application jobs.

2.2.2.1 Single appearance

Ability to model an application job that is not part of a series, and is released at a specific point in time.

2.2.2.2 Strictly periodic

Ability to model an application job that is part of a series of jobs with release times separated by a constant time interval. If the release time of a job and its order within the series is known, the release time of all other jobs can be derived from it.

2.2.2.3 Sporadic

An application job that is part of a series of jobs with release times separated by a time interval that has a known lower bound. For every release of a job, it is therefore known that the release of the subsequent job of the series will not happen before that lower bound.

2.2.2.4 Aperiodic

An application job that can be released at any arbitrary time. It can be used to model event-driven systems where no assumptions can be made about the event sources. If an assumption can be made about the minimum time interval between successive events, such job series can be conservatively (but perhaps not accurately) modelled as sporadic jobs.

2.2.2.5 Fully dependent

An application job that is released immediately after the completion of all jobs that it depends on.

2.2.2.6 N out of M dependent

An application job that is released immediately after the completion of any N jobs out of all M jobs that it depends on, $(M > N)$.

2.2.3 Requirements on Modelling Load Resourcing Constraints

The resourcing of applications defines which kind of resources a given job requires for its execution. This requires a taxonomy of resources over different types. The load model addressed here makes no assumption about such taxonomy, and it may work under different typing systems (e.g., flat type hierarchy, single-parent type hierarchy, multiple-inheritance type systems), as different resource allocation mechanisms might benefit from them. Regarding this classification, we list the following requirements for a load modelling approach, so that it is powerful enough to represent the following types of application jobs.

2.2.3.1 Untyped job

Ability to model an application job that can be executed on any type of resource.

2.2.3.2 Single-typed job

An application job that must be executed over a specific type of resource.

2.2.3.3 Multi-typed job

An application job that can be executed over multiple types of resource.

2.2.4 Requirements on Modelling Load Characterisation

The characterisation of the application load defines how long each of its jobs uses the resources they were allocated. Regarding this classification, we list the following requirements for a load modelling approach, so that it is powerful enough to represent the following types of application jobs.

2.2.4.1 Fixed load

An application job that always occupies a resource for a constant amount of time, regardless of the resource. The load of such a job can be characterised by a scalar.

2.2.4.2 Probabilistic load

An application job that occupies a resource for a probabilistic amount of time, regardless of the resource. The load of such a job is a random variable, and can be characterised by a histogram or a probability density function.

2.2.4.3 Typed fixed load

A multi-typed application job that occupies resources of different types by a potentially different, yet constant amount of time. The load of such a job can be characterised by a vector of scalars, and the length of the vector is equal to the number of types of resources that the job can occupy.

2.2.4.4 Typed probabilistic load

A multi-typed application job that occupies resources of different types with a potentially distinct stochastic behaviour on each of them. The load of such a job can be characterised by a vector of probability density functions or histograms, and the length of the vector is equal to the number of types of resources that the job can occupy.

2.3 An Interval Algebra for Load and Resource Modelling

Within this book, we will rely on a novel approach to load and resource modelling based on an interval algebra (IA). It will be used throughout the book to ease our understanding of the impact of different resource allocation mechanisms. But more importantly, it can be used by the resource allocation mechanisms themselves as an internal representation of the resources and the load that they are supposed to manage.

Our IA represents non-functional characteristics of application load using the mathematical concept of intervals. It can be used to analytically derive the impact of using different resource allocation policies on the original application characteristics. The main concerns of this book are performance and time predictability, so most of our examples focus on the representation of time intervals, but the interval algebra can naturally be extended to support other non-functional properties such as energy dissipation.

In a simplistic example, we can consider an application with three jobs A, B and C, and a homogeneous platform composed of two processors with first-come-first-serve scheduling. Each of the jobs can be represented by an interval that denotes the time they need to run: $A = [0, 30[$, $B = [0, 45[$, $C = [0, 20[$ (assuming in this example that they are all independent and ready to run at time $= 0$). By using simple interval algebra operations, a resource allocation heuristic can estimate the response time R of the three tasks under different allocation schemes (e.g., $R_A = 30$, $R_B = 45$ and $R_C = 50$ if A and C are allocated, in that order, to one of the processors and B is allocated to the other), and thus can dynamically decide whether it is likely to meet the applications constraints when using a given allocation.

While trivial, such example can be made arbitrarily complex by allowing different resource scheduling disciplines, a larger number of tasks and processors. For the interval algebra, however, the analysis of the response times under a specific allocation would still involve the application of the same interval manipulation rules.

The advantages of such an approach are numerous, including the following.

- It enables dynamic allocation mechanisms to have an appropriate level of confidence on whether the chosen allocation meets the applications' timing constraints.
- The approach can be used as a fitness function of search-based allocation heuristics, if the algebraic operations are sufficiently lightweight as they have to be applied over a potentially large search space (some examples of integrating IA to genetic algorithms are provided in Chapter 5).
- The solution of algebraic operations can be found in multiple ways, with different levels of performance. Therefore, resource allocation heuristics

can be improved simply by optimising the solution of the employed algebraic operations.

- If absolute predictability is not required (i.e., in soft real-time and best-effort applications), algebraic operations can be solved faster by applying approximations that sacrifice the accuracy of the final result. This enables allocation mechanisms that can be applied to systems with different levels of strictness of their timing requirements.

Let us now introduce the main principles behind this interval algebra. Our goal in this book is not to be overly formal, so we will favour intuitive descriptions over mathematical formalism whenever possible (i.e., without sacrificing precision). In general terms, an algebra is a definition of symbols and the rules for manipulating those symbols. Our interval algebra, therefore, establishes rules for the manipulation of intervals. It defines different types of intervals, which represent the amount of time a particular piece of application load requires from a notional resource. For example, a single job can be represented by a time interval using the notation below:

$$\#A\#0\#40 \tag{2.1}$$

where the first element of the tuple is a unique job identifier, the second is a non-negative real number representing the release time of the job and the third is a positive real number representing the job's load, i.e., the actual length of the time interval. In the example above, job A is released at time 0 and requires 40 time units of a resource. The same concept can also be represented using the mathematical notation for a left-closed right-open bounded interval $[0, 40[$.

Following the definition above, our IA must also define rules for manipulations of such intervals: what happens when an interval is allocated to a specific type of resource, what if two intervals are allocated to the same resource, etc. Widely used algebras define a small number of basic operations (e.g., addition, multiplication) and then define more complex operations as composites of those basic operations (e.g., matrix multiplication). Our IA defines two basic algebraic operations: *time displacement* and *partition*. Time displacement changes the endpoints of an interval by an arbitrary value t, and denotes that the job has to wait for its allocated resource (i.e., its starting and ending times were moved t time units to the future). Partition simply breaks one interval in two, and denotes that a job was preempted from a resource (and the second interval produced by the partition is likely to be time-displaced). All other interval-algebraic operations of IA, which can represent an arbitrarily large set of allocation and scheduling mechanisms, can be expressed as compositions of these two. By applying these operations, it is possible to investigate the impact of different resource allocation and scheduling mechanisms on the endpoints of the intervals, which in turn denote the completion times of each application component.

In the following subsections, we show how our IA addresses the requirements described in Section 2.2.

2.3.1 Modelling Load Structure

The interval-based representation of a job presented above is sufficient to express a singleton. By using a set of such intervals, independent jobs can be also represented. To denote a dependency between two tasks A and B, the notation can be extended to include a job identifier instead of the release time of a job:

$$\#B\#A\#50 \tag{2.2}$$

This notation is capable of denoting single dependency jobs, and conveys that interval B's start-point depends on interval A. Multiple dependencies can also be specified as a dependency set, and thus multi-dependency jobs can be covered:

$$\#C\#\{A, B\}\#260 \tag{2.3}$$

This notation assumes that whenever an interval has dependencies, its start-point lies exactly at the highest endpoint among all the intervals it depends on. In this example, assuming that jobs A and B are defined as in examples (2.1) and (2.2), this leads to: $A = [0, 40), B = [40, 90), C = [90, 350)$.

2.3.2 Modelling Load Temporal Behaviour

The intervals described in the previous subsection are single-appearance and have a fixed release time, therefore express singleton jobs. A strictly periodic series of jobs can be characterised by its release time, the period after which a new job is released, and the time interval each job requires from a notional resource. We denote such job series with the notation exemplified below, which is exactly the same as the notation of a singleton task followed by the period:

$$\#D\#0\#40\#100 \tag{2.4}$$

Mathematically, it represents an infinite series of intervals, such as: $D = [0, 40), [100, 140), [200, 240), \ldots$. This extension is expressive enough to represent strictly periodic tasks.

The release time of sporadic tasks is not deterministic but has well defined bounds. In case of aperiodic tasks, those bounds do not exist. To model those cases, IA represents release times with *aleatory variables*. Those variables are associated with probability distributions that can constrain assumed values. We will cover that approach in Section 2.3.5 when we discuss intervals with stochastic representations of time.

2.3.3 Modelling Load Resourcing Constraints

IA represents a resource as the dimension over which jobs are operated upon. Jobs, each represented by its respective interval, are allocated onto a resource; algebraic operations determine how the resource is shared between all of them, and how the resource sharing affects their timings. We denote a resource with the notation exemplified below:

$$+Z_1(\#A\#0\#40) \tag{2.5}$$

where the algebraic operation $+Z_1$ is applied to the set of intervals surrounded by brackets (only A in the example above). The example below shows the same resource, but this time with two distinct jobs mapped to it:

$$\begin{aligned}&+Z_1(\#A\#0\#40, \#B\#0\#50) =\\&+Z_1(\#A\&40, \#B\&90) =\\&+Z_1([0,90))\end{aligned} \tag{2.6}$$

In this example, we introduce two different ways to evaluate the operator $+Z_1$ (which we can intuitively understand as a resource serving jobs under a FIFO schedule). The first evaluation of the operator preserves the identities of the mapped jobs, and it indicates the completion times of each one of them after the symbol "&". We will refer to this type of evaluation as *information-preserving* (or simply *preserving*). The second way to evaluate the operator is equivalent to the first, but it does not preserve any information about the individual operands. It simply determines the busy period(s) of the resource with one or more intervals. We refer to this type of evaluation as *information-collapsing* (or simply *collapsing*).

Comparing with elementary algebra, the two evaluations of the operator are akin to solving an expression like $(3+5)+(1+2)$ using an intermediate step $8+3$ before arriving to the final result 11. In both algebras, there is an infinite set of possible operands that could lead to a particular result, and there is no information in the final result that could allow the backtracking of the initial operands.

A slightly different example is shown below, using the same jobs but this time mapped onto resource $+Z_2$ that uses a time-division multiplexing (TDM) scheduler with a quantum of 8 time units:

$$\begin{aligned}&+Z_2(\#A\#0\#40, \#B\#0\#50) =\\&+Z_2(\#A\&72, \#B\&90) =\\&+Z_2([0,90))\end{aligned} \tag{2.7}$$

It is worth noticing that only the intermediate expression (i.e., after the preserving evaluation) differs, and the final result after the collapsing evaluation is the same. This is always the case if the operand denotes a work-preserving scheduler (i.e., a resource is never idle if there are jobs ready to be served).

The two following examples show jobs allocated to a resource that is shared under a priority-preemptive scheduler, assigning priorities in the same order the jobs are passed to the operator (higher to lower):

$$\begin{aligned}&+Z_3(\#C\#15\#40, \#D\#10\#50, \#E\#0\#50) = \\ &+Z_3(\#C\&55, \#D\&100, \#E\&140) = \\ &+Z_3([0,140))\end{aligned} \tag{2.8}$$

$$\begin{aligned}&+Z_4(\#F\#10\#4, \#G\#0\#18, \#H\#26\#5, \#I\#24\#8) = \\ &+Z_4(\#F\&14, \#G\&22, \#H\&31, \#I\&37) = \\ &+Z_4([0,22),[24,37))\end{aligned} \tag{2.9}$$

In both cases, the algebraic operations abstracts away the specific interleaving patterns of the execution of every job. Each of the evaluation types focusses solely on, respectively, the finish times of each job or the idleness of the resource. For example, formula (2.9) represents the following: job G starts to be executed at time zero, but after 10 time units it is preempted by job F which runs to completion for 10 time units; then G resumes and runs for its remaining execution time until time equals 22 units; resource Z_4 becomes idle until job I is released at 24 time units, which in turn suffers a preemption from H between times 26 and 31 units and then executes until time equals 37 units
.

Just like single appearance jobs, periodic jobs can be allocated to resources:

$$\begin{aligned}&+Z_1(\#A\#0\#40\#100, \#B\#0\#50) = \\ &+Z_1(\#A\&40, \#B\&90, \#A\#100\#40\#100) = \\ &+Z_1([0,90), \#A\#100\#40\#100)\end{aligned} \tag{2.10}$$

It is important to notice that a periodic job series always remains as a distinct interval in the result of both preserving and collapsing evaluations of an operator. This reflects the infinite nature of the series.

One of the crucial properties of a job is its affinity, which means that it can be served only by the designated resources. The job that can be executed on any resource available in a system is referred to as *untyped job*. If a job can be executed on a single type of resources only, it is a *single-typed* job. A *multi-typed* job can be executed on a few (enumerated) resource types, possibly with different execution times on each of them. In all examples so far, only untyped jobs have been used. To describe a single-typed or multi-typed job, the notation should support the definition of different types of resources and different types of resource affinity. This can be expressed as follows, where each scalar in pointy brackets denotes a different type and the absence of type constraints implies untyped jobs or resources (as in examples above):

$$+\, Z_5\langle 2\rangle(\#J\langle 2\rangle\#0\#15, \#K\langle 2,3,8\rangle\#0\#20, \#L\#0\#14) \tag{2.11}$$

By allowing the definition of resources types and resource requirements, it is also possible to present communicating jobs by modelling the job as two fully dependent intervals with distinct resource requirements, one for computation and one for communication (i.e., the job can only communicate over resource 2 once it has finished being computed over resource 1):

$$\begin{aligned} &\#L\langle 1\rangle\#0\#14 \\ &\#M\langle 2\rangle\#L\#340 \end{aligned} \tag{2.12}$$

2.3.4 Modelling Load Characterisation

The representation of load as the interval length, denoted by a positive real number (as defined in Subsection 2.3.1), is already capable of representing a fixed load.

To represent a typed fixed load, we allow the specification of different interval lengths for different resource types using a similar notation as the one introduced at the end of Subsection 2.3.3:

$$\#N\langle 2,4,6\rangle\#0\#\langle 10,20,20\rangle \tag{2.13}$$

To represent a probabilistic load or typed probabilistic load, we have to rely on aleatory variables to represent the load. This can be done for both typed and untyped jobs.

2.3.5 Stochastic Time

In many cases, it may be desirable to represent intervals with non-deterministic temporal behaviour or load characterisation. In these cases, IA allows the use of aleatory variables, which follow a probability distribution, instead of scalars. It does not impose any limitation on the choice of probability distributions, and their parameters should be provided following a well established notation. For example, a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with parameters mean $\mu = 2$ and variance $\sigma^2 = 1$, $\mathcal{N}(2, 1)$ can be used to denote the release time of job P, and similarly $\mathcal{N}(40, 1)$ can denote its execution time:

$$\#P\#normal(2,1)\#normal(40,1) \tag{2.14}$$

The time when job P finishes its execution is described by the convolution of two Gaussians: $\mathcal{N}(2, 1) * \mathcal{N}(40, 1)$.

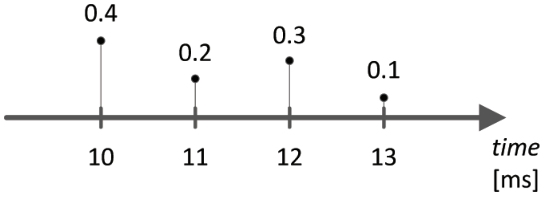


Figure 2.1 Example of a probability mass function of a discrete random variable describing a job's execution time.

It is particularly convenient to represent time as a discrete random variable described by a probability mass function (PMF), i.e., a function giving the probability that a discrete random variable is equal to a provided value. All values of a PMF should be non-negative and sum up to 1. Using this function for describing a job's execution time, the best-case execution time (BCET) and the worst-case execution time (WCET) correspond to the first and the last probability value of the distribution, respectively. Between these two extremes, the probabilities of the remaining possible execution times are described. For example, the PMF of a job execution time whose BCET and WCET equals to 10 ms and 13 ms is shown in Figure 2.1. This job can be described using IA as:

$$\#Q\#0\#pmf(10, 0.4), (11, 0.2), (12, 0.3), (13, 0.1) \tag{2.15}$$

To show how tasks with stochastic timing can be mapped to a resource, we use job T whose release time is described by discrete uniform distribution $\mathcal{U}\{0, 1\}$ and is executed in 40 time units, and job U depending on T executed in $\mathcal{U}\{1, 4\}$ time units by a notional resource $+Z_1$ with FIFO scheduling. Then:

$$\begin{aligned} &+Z_1(\#T\#pmf(0, 0.5), (1, 0.5)\#40,\\ &\#U\#T\#pmf(1, 0.25), (2, 0.25), (3, 0.25), (4, 0.25)) =\\ &+Z_1([pmf(0, 0.5), (1, 0.5), pmf(41, 0.125), (42, 0.25),\\ &(43, 0.25), (44, 0.25), (45, 0.125))) \end{aligned} \tag{2.16}$$

2.4 Summary

This chapter presented the requirements for workload and platform models that are suitable to support resource allocation mechanisms in embedded, high-performance and cloud computing. Such models can be used as internal representations, allowing resource allocation mechanisms to evaluate different allocation alternatives. A specific modelling approach has been introduced, based on an interval algebra, which fulfils the listed requirements and is amenable to compact and efficient implementations. A reference implementation of the presented algebra is available from the DreamCloud project website[1].

[1] http://www.dreamcloud-project.org

3

Feedback-Based Admission Control Heuristics

Applying feedback mechanisms to monitor the capacity of computing resources and quality-of-service (QoS) levels can guarantee a bounded time response, stability, bounded overshoot even if the exact knowledge of a system workload and service capacity is not available a priori [2]. Thus, in case of a careful fine-tuning of parameters, they can be successfully applied even to systems with real-time constraints (see the Related Work section). It was verified that this approach helps to find a trade-off between multiple objectives of a workflow management system, e.g., minimal slacks and maximum core utilisation [53].

The feedback-control dynamic resource allocation heuristics impose some requirements on the target system, which should guarantee that the appropriate input data is available and that the generated output can be used to perform the proper resource allocation. Usually to perform a resource allocation decision we can rely on various metrics, provided by the monitoring infrastructure tools and services, such as utilization and the time latency between input and output timestamps [81]. The system should also guarantee an appropriate level of responsiveness to the decisions made by the heuristics, as well as update the values of the metrics used as inputs in the algorithm frequently enough for the particular application. The platform should support scheduling on distributed-memory infrastructure resources. It is important to provide the heuristic algorithm with realistic data about system workload, service capacity, worst-case execution time and average end-to-end response time [84].

The task mapping process presented in this chapter is comprised of the resource allocation and task scheduling. The technique proposed in this chapter assumes the presence of a common task queue, which is used by the global dispatcher. The resource allocation process is executed on a particular processing unit. Its role is to send the processes to be executed to other processing units, putting them into the task queue of a particular core. The process dispatching, i.e., selecting the actual process to run, is also a part of the scheduling algorithm and is carried out locally on each core. It is assumed that task scheduling is performed in a non-preemptive early deadline first (EDF) or first-in-first-out (FIFO) based manner.

Later in this chapter we propose an algorithm to map firm real-time tasks into multi-core systems dynamically, using dynamic voltage and frequency scaling (DVFS) to decrease energy dissipation in cores. According to simulation results, the proposed method leads to more than 55% of dynamic energy reduction.

3.1 System Model and Problem Formulation

3.1.1 Platform Model

The controlling process of dynamic behaviour of a target system can be performed in two ways: feed-forward and feed-back, presented in Figure 3.1. Although the closed-loop scheme includes larger number of functional blocks and requires measuring output values, it requires less accurate model of the target system and is also more resistant to disturbances [7]. A closed-loop system is characterised with a feedback loop, which carries values of *measured output* (*y(t)*, aka *controller value*). These values are subtracted from their desired value (r, *reference signal*, *setpoint*). The result of this operation forms *error* ($e(t)$) signal, which is used to compute *control input* ($u(t)$). This value is sent back to the target system.

A proportional-integral-derivative (PID) controller is particularly often used in various industrial control system, recently including computing systems [57].

The PID controller in the time-domain form is described in the following way:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau) d\tau + k_d \frac{d}{dt} e(t). \tag{3.1}$$

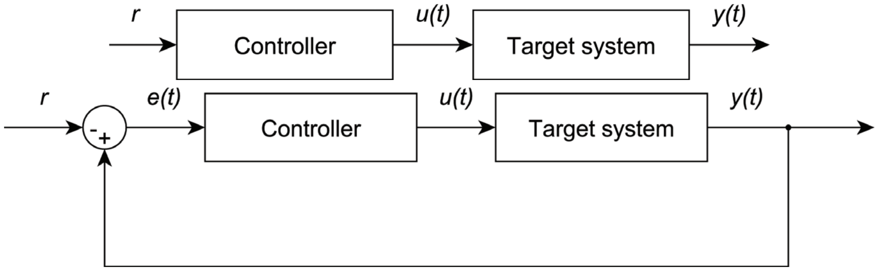


Figure 3.1 Block diagrams of control system architectures: feed-forward (*above*) and feedback (*below*).

The determination of proportional (k_p), integral (k_i) and derivative (k_d) constant components of PID controller is known as PID controller tuning.

The PID controller is often presented in an equivalent form in the frequency domain, where function (3.1) of time t is presented as a function of complex frequency s using the Laplace transform, leading to

$$K(s) = k_p + \frac{k_i}{s} + k_d s. \tag{3.2}$$

A PID controller is often described using other constant parameters: k – so called proportional gain, T_i – integral time constant and T_d – derivative time constant

$$K(s) = k\left(1 + \frac{1}{sT_i} + sT_d\right). \tag{3.3}$$

Since increasing the value of parameter k_d enhances noise, the derivative component is often omitted in numerous practical applications [57]. It is also not used in the work described in this chapter despite its positive influence on stability or speed.

In Figure 3.8, a general view of the proposed architecture is presented.

3.1.2 Application Model

We consider a workflow of a particular structure. There is no dependencies between tasks and the deadline of each task computation is set as a sum of its computation time multiplied by an arbitrary value and arrival time. There is only one priority of task; tasks cannot be preempted during their execution. During simulation we measure cluster core utilisation, which is the percentage of cores in the clusters executing tasks in particular simulation time t.

3.2 Distributed Feedback Control Real-Time Allocation

After releasing task t_i, the role of the dispatcher is to decide which of the clusters C_j, $j = 1, \ldots, m$, is to execute the task. This decision can be made using various metrics, we decide to apply a choice of the cluster whose cores are currently the most idle. If more than one core satisfies the chosen condition, one of them is chosen randomly. For the comparison purpose we also allowed the dispatcher to choose the target cluster C_j in the round-robin manner. The task t_i is then placed in the j-th queue.

Each j-th cluster includes one admission control block, AC_j. Its role is to decide whether a task t_i, read from the j-th input queue, should be executed by the cluster. The first condition of admittance is that the deadline of t_i, D_i, is not lower than the sum of its computation time, C_i and the current simulation

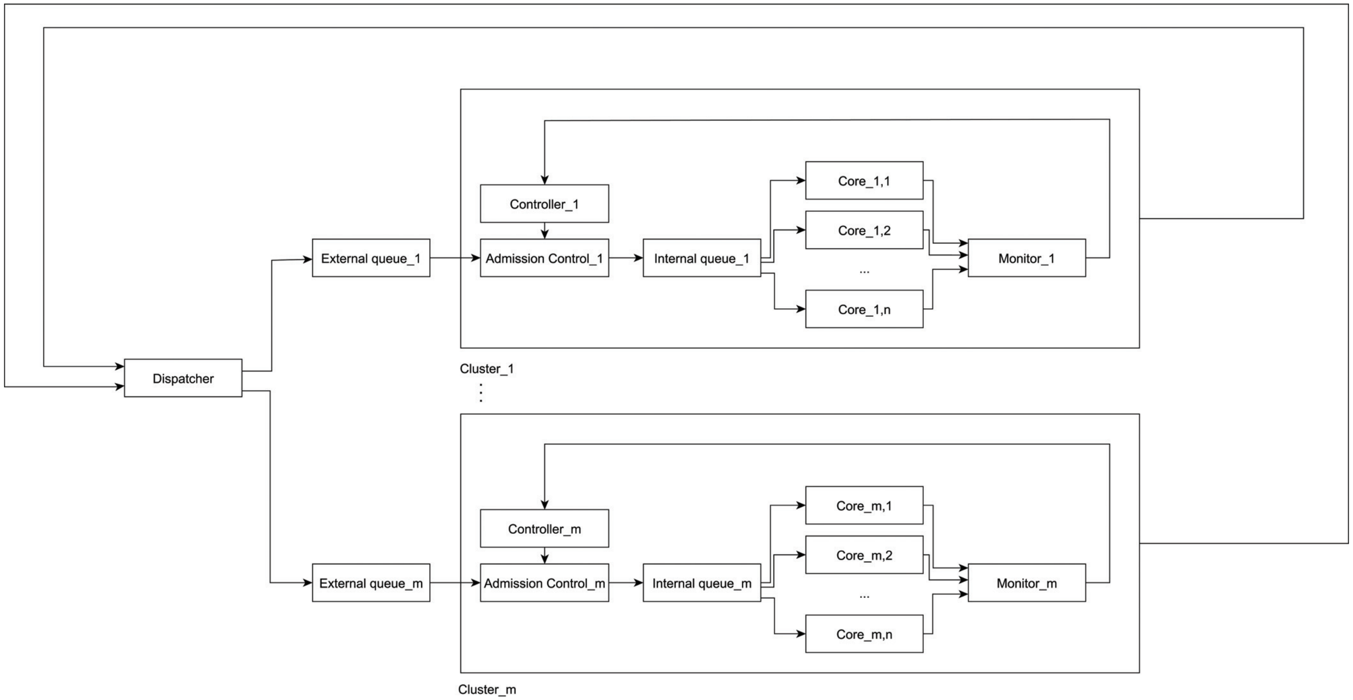


Figure 3.2 Distributed feedback control real-time allocation architecture.

time t. Then the input value from the controller, $u_j(t)$, is tested. If this value is positive, the task is admitted, otherwise it is rejected. Admitted tasks are placed in the internal cluster queue. This queue is planned to be rather short to minimise the delay between decision about admittance and the execution of the task, and to keep the timeliness of the lateness input.

To control the admittance in each cluster, we use discrete-time controllers in two variants. The first of them is a PI (i.e., a PID controller without the derivative component) whose controlled value is an average lateness of a (parameterisable) number of previous tasks computed by the cluster cores, where lateness is defined as the difference between a task response time and its deadline. If a lateness is negative, the task has been finished before its deadline, and positive otherwise. The current value of lateness is compared with the setpoint, r, and an error $e_j(t)$ is computed. It is provided as an input to a controller, which computes admittance allowance value $u_j(t)$. The second variant includes a P controller (i.e., a PID controller with the proportional component only) whose output value $u_j(t)$ depends on the difference between the current core utilisation and the setpoint. The output value of $u_j(t)$ is sent to AC_j, where it is used to perform a task admittance decision. In both situations, as long as value of control input $u_j(t)$ is positive, the task is allowed to be submitted to cores, otherwise it is rejected. The admitted tasks are placed in the queue.

An idle core $Core_{j,k}$, $j = 1, \ldots, m$, $k = 1, \ldots, n$, fetches a task t_i from the j-th core queue and then executes it in a non-preemptive manner. After execution, the lateness of the i-th task, $L_i = D_i - t$, is computed. Each core also informs the observer whether it is occupied or idle.

The role of observer $Monitor_j$ is to compute two metrics based on the performance of all cores in the j-th cluster. The first metric is core utilisation and the second metric is an average lateness of the previous q tasks computed by the cores in the j-th cluster. These data are provided to the j-th controller and the dispatcher.

3.3 Experimental Results

3.3.1 Controller Tuning

In order to check the efficiency of the proposed feedback-based admission control and real-time task allocation, we developed a simulation model using SystemC language. We firstly configured it to operate in the open-loop manner.

In Figure 3.3 we present the maximum task lateness in the open-loop system consisted of three clusters, each including three cores. In every situation, at 5,000 ns a number of tasks, ranging from 5 to 500, each requiring execution time equal to 50,000 ns, has been generated. Then we looked at the

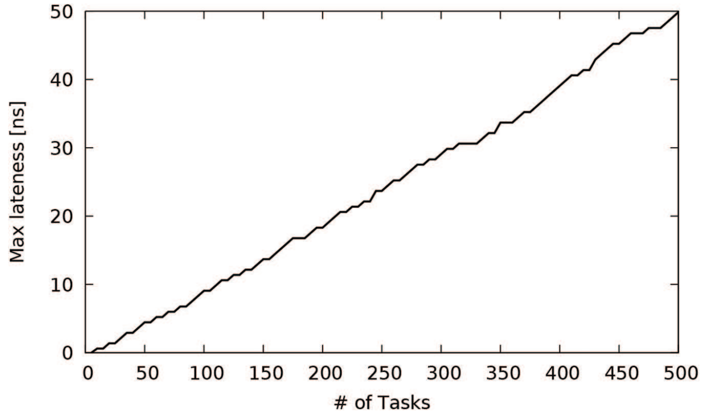


Figure 3.3 Maximum normalised task lateness (with execution time equal to 50,000 ns) in step responses for a number of tasks (3 clusters, 3 cores in each).

maximal task lateness, where each lateness has been normalized by dividing it with the deadline.

In order to tune the controller, we analysed the step-input maximum normalised task lateness response in the open-loop system. As an input we have used a burst release of 500 tasks (with execution time equal to 50,000 ns) at 5,000 ns. The system was comprised of 3 clusters, each including 3 computing cores. The obtained result confirms the accumulating (or integrating) nature of the process, which can be described by the following model [64]:

$$F(s) = \frac{V}{s} e^{-s\tau}, \tag{3.4}$$

where τ is the dead time, i.e., the delay between changing input and the observable output reaction, and V is the velocity gain, which is the slope of the asymptote of the process output.

In such kind of processes, to choose proper values of PI controller components, AMIGO (Approximate M-constrained Integral Gain Optimisation) tuning formulas can be applied [7]. According to these formulas, the parameters k and T_i are equal:

$$k = \frac{0.35}{V\tau}, \tag{3.5}$$

$$T_i = 13.35\tau. \tag{3.6}$$

Both these parameters can be determined using the step output illustrated in Figure 3.4: $k = 0.2741$ and $T_i = 1108$.

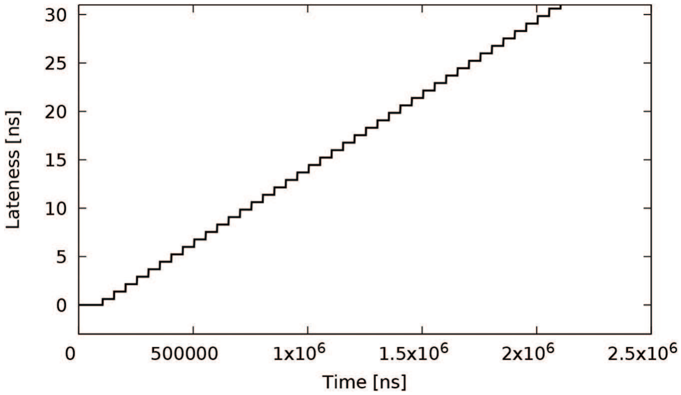


Figure 3.4 Maximum normalised task lateness step response for 500 tasks (with execution time equal to 50,000 ns) released at 5,000 ns (3 clusters, 3 cores in each).

The usage of the core utilisation in a cluster as a controlled value is a bit more tricky due to its non-linearity. Because of the obvious saturation at 100 per cent (see Figure 3.5) in the case of step response, to compute parameters of a controller we limit the considered operating region to the proportional range before the saturation, which ranges from 1 to 4 ns. Its maximum slope tangent can be described by linear formula $y = 0.33\,x - 0.33$. According to classic Ziegler-Nichols method [64], the k parameter of the P controller can be computed as

$$k = \frac{1}{\lambda}, \tag{3.7}$$

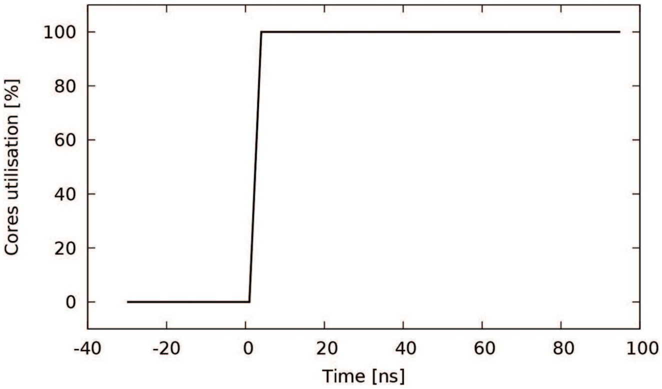


Figure 3.5 Cores utilisation step response for 500 tasks (with execution time equal to 50,000 ns) released at 0 ns (1 cluster with 3 cores).

where λ is the absolute value of the y-coordinate of the intersection of the max slope tangent with the OX axis. In our case $\lambda = 0.33$ and, consequently, $k = 3$.

3.3.2 Stress Tests

The workload used in our introductory experiment consists of 900 independent tasks, one released every 5,000 ns, whose computation time equal to 50,000 ns and deadline is set to the sum of computation time multiplied by 1.2 and the task release time. In Table 3.1, the number of rejected tasks, tasks executed before and after their deadlines in various controlling environment settings is presented.

The two first rows present the result obtained in the open-loop systems. The choice of the external queue has only slight influence on the number of tasks executed before their deadlines. In these situations task can be rejected by the admission control only if the task slack (computed as $D_i - C_i - t$) is negative.

Applying a closed-loop approach improves the system performance significantly, but the proper choice of the measured output is also essential. In case of the core utilisation not a single task finishes after its deadline, as the tasks are submitted to the queue only when there is at least one idle core, that can start executing the task instantly. The lateness is less correlated with the real temporal availability of computational power, so as many as 116 tasks

Table 3.1 Number of rejected tasks, tasks executed before and after their deadlines in various controlling environment configurations for a periodic task workload simulation scenario (3 clusters, 3 cores in each): configuration parameters (*above*) and obtained results (*below*)

Config. No.	Architecture	Queue	Controller Value	Controller	Allocation
1	Open-loop	FIFO	–	–	min core util.
2	Open-loop	EDF	–	–	min core util.
3	Closed-loop	Both	core utilisation	P	min. CPU util.
4	Closed-loop	Both	lateness	PI	min. CPU util.
5	Closed-loop	Both	core utilisation	P	RR
6	Closed-loop	Both	lateness	PI	RR

Config. No.	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
1	149	661	90
2	154	655	91
3	738	0	162
4	614	116	170
5	675	0	225
6	607	127	166

have been sent to the queue despite not a single core was capable of computing the task before its deadline.

To asses the improvement of the core-utilisation-based allocation, we performed simulations where the tasks are allocated to clusters in a round-robin manner. In both P- and PI-based architectures we obtained worse results by 8.5 and 1.14 per cent, respectively. Importantly, the higher improvement has been observed in the architecture leading to the overall better results.

The clusters' cores utilisation for this architecture during the first 500 ns is presented in Figure 3.6. Except for the initialisation (and finalisation, not shown in the figure) there is no time when any of the clusters has less then 66 per cent of the core utilisation. After computing the average core utilisation during the whole simulation time we get 90.12%, 90.20%, and 90.16% for the first, second and the third core, respectively. The tasks have been sent by the dispatcher to the first cluster 296 times and to the 2nd and the 3rd core respectively 292 and 313 times, which can be viewed as a quite even distribution.

The control signal (generated by a controller, sent to the admission control) for the first 500 ns of the simulation is shown in Figure 3.7. The positive value of this signal means that at least one core from the given cluster has finished the previous task computation and is being idle. In this situation the next task should be submitted to the cluster as soon as possible.

Similar results have been observed in other workloads of a periodic nature with uniform (or nearly uniform) execution time.

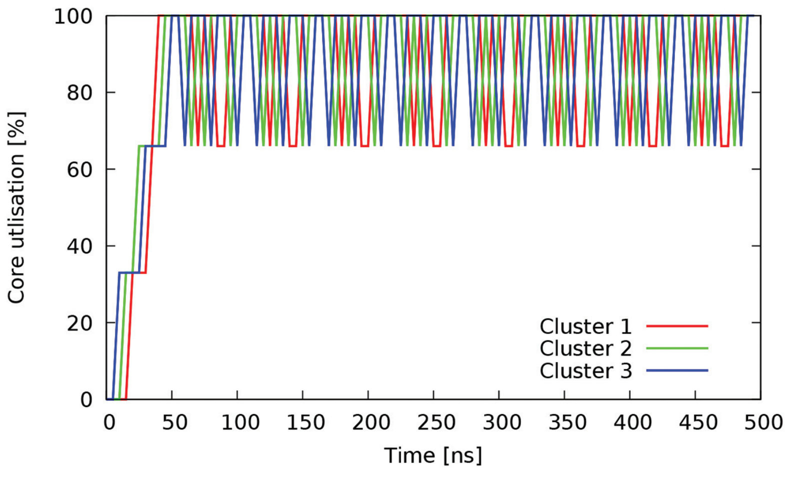


Figure 3.6 Core utilisation measured during the first 500 ns of the simulation.

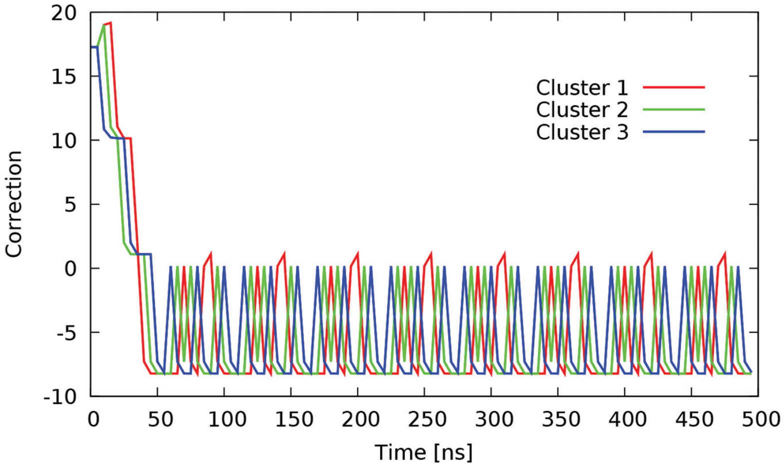


Figure 3.7 Control signal observed during the first 500 ns of the simulation.

3.3.3 Random Workloads

In our next experiment, summarised in Table 3.2, we analysed 30 randomly bursty workloads, generated according to the method described in [22], including from 827 to 962 tasks of diverse execution time, ranging from 1 to 2,67,582 ns. Three target system configurations have been checked: open-loop with EDF and closed-loop with CPU utilisation as the controller value, where the allocation is performed using the minimal core utilisation metric and in the round-robin way. Once again, the closed-loop approach leads to better results but, in comparison with the periodic-task scenario in the

Table 3.2 Total number of rejected tasks, tasks executed before and after their deadlines in various controlling environment configurations for 30 random bursty task workload simulation scenarios (3 clusters, 3 cores in each): configuration parameters (*above*) and obtained results (*below*)

Config. No.	Architecture	Queue	Controller Value	Controller	Allocation
1	Open-loop	EDF	–	–	min. core util.
2	Closed-loop	Both	core utilisation	P	min. core util.
3	Closed-loop	Both	core utilisation	P	RR

Config. No.	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
1	10603	1752	14059
2	12296	753	13365
3	11946	675	13793

experiment described above, the improvement, equal to about 16%, is slightly less impressing. Similarly, the difference between allocating task under the minimal core utilisation criteria and round-robin is rather slight and equals 3 per cent. It is worth stressing, however, that the tasks in the analysed workloads are characterised with very diverse time of computations, but despite this variance they are not differentiated by our model. Consequently, one task can occupy a core for longer time, not allowing other (submitted a bit later) tasks to be executed on this core because of the lack of preemption.

3.4 Dynamic Voltage Frequency Scaling

Dynamic Voltage Frequency Scaling (DVFS) is a power saving technique, omnipresent in CMOS circuits, benefiting from the fact that their dynamic (or switching) power P is proportional to the the square of core supply voltage V, and its clock frequency f, i.e., $P \propto fV^2$. Since any reduction of core voltage requires an adequate decrease of the clock frequency, some trade-off between energy savings and computation performance is achieved. Some guidance in real-time systems stems from the fact that there is usually no additional benefits from faster task execution as long as it is before the deadline. Moreover, for typical workloads the required peak computational performance is usually much higher than the average [106]. Thus sustaining a lower voltage/frequency for most of time and increasing it only when required by a workload growth, in a way it risks missing some deadlines, seems to be a sensible strategy. To perform a proper voltage scaling decision, it is possible to rely on various metrics, provided by the monitoring infrastructure tools and services, such as utilization and time latency between input and output timestamps [81]. In multiprocessor domain, the cores can operate on different voltage at a given instant, so allocating a task to the most suitable core starts to be a more sophisticated task even in case of homogeneous cores, since assigning a task to a core with lower voltage can lead to missing the deadline that would be met in case of a different decision. The term voltage scheduling has been introduced to refer to scheduling policies using DVFS facility to improve energy efficiency.

In Multiprocessor Systems on Chips (MPSoCs) a task can be mapped to a core either statically or dynamically, just before its execution, which is particularly beneficial in case of workloads not known a priori [123]. In DVFS-based systems, the problem of dynamic task mapping is even more difficult, since not only resource utilisation and application structure have to be analysed, but also the present voltage level of each processor needs to be considered. Modern operating systems, including both Windows and Linux (2.6 Kernels and above) support dynamic frequency scaling for systems with Intel (SpeedStep technology) and AMD (PowerNow! or Cool‘n’Quiet technology) processors. Frequency levels in these chips are not continuously available, but a limited number of discrete voltage/frequency levels is offered. They follow the Advanced Configuration and Power Interface (ACPI) open standard, defining such processor states as C0 (operating state), C1 (halt), C2 (stop-clock), and C3 (sleep). In C states with higher numbers less energy is consumed, but returning to the normal operating state imposes more latency. In some device families additional C-states have been introduced, such as C6 in Intel Xeon when an idle core is power gated and its leakage is almost entirely reduced to zero [52]. While core is in the C0 state, it operates with one of several power-performance states, known as P-States. In P0, a core works with the highest frequency and voltage level, and subsequent P-States offer less performance but also require less energy. The most recent ACPI specification can be found at Unified Extensible Firmware Interface Forum[1].

In operating systems, frequency scaling depends on an applied governor. In case of Linux, the *ondemand* governor switches frequency to the highest value instantly in case of high load, whereas the *conservative* governor increases frequency gradually [77]. These policies aim to keep processor utilization close to 90%, progressively decreasing or increasing frequency using heuristics [102]. This approach may, however, negatively impact applications with timing constraints. To overcome this limitation, a *custom* governor can be developed and applied. These governors can then operate on per-core and per-chip basis, taking into account utilisation of other machines in a cluster, etc. A valuable comparison between per-core and per-chip DVFS is presented in [68], where per-core DVFS is shown to offer even more than 20% energy savings in comparison with the conventional chip-wide DVFS with off-chip regulators. However, per-core DVFS is rarely implemented and, for example, all active cores in contemporary Intel i7 processors must operate with the same frequency in the steady state, whereas AMD processors allows their cores work with different frequencies, but one voltage value, appropriate to the core with the highest frequency, is to be provided to all the cores [52].

In the remainng part of this chapter, we propose a custom governor algorithm for per-chip DVFS. The algorithm performs dynamic resource allocation and assumes the presence of a common task queue, which is used by a global dispatcher. The resource allocation process is executed on a particular processing unit, whose role is to send the processes to be executed to other processing units, putting them into the task queue of that core. The task dispatching, i.e., selecting the actual task to run, is also a part of the mapping algorithm and is carried out locally on each processor. It is assumed that task

[1] http://www.uefi.org

scheduling is performed in a non-preemptive first-in-first-out (FIFO) based manner for simplicity, but another scheduler can be used instead.

3.5 Applying Controllers to Steer DVFS

In Figure 3.8, a general view of the proposed architecture is presented, where dashed lines are used for steering P-States. We consider workflows of a particular structure. All tasks are assumed to be firm real-time, so certain number of missing deadlines is allowed, but the task executed after its deadline is invaluable to the user. There are no dependencies between tasks and all tasks have equal priorities. Further, tasks cannot be preempted during their execution.

After releasing task T_i, the role of the dispatcher is to decide which of the processors $Processor_j, j = 1, \ldots, m$, is to execute the task. This decision can be made using various metrics. We measure processor core utilisation, which is the percentage of busy cores in the processors executing tasks in particular simulation time t, and choose the processor whose cores are currently the least utilized. If more than one processor have the same lowest utilisation, one of them is chosen randomly. The task T_i is then placed in the j-th external queue.

To control the admittance in the j-th processor, we use a discrete-time PI controller (i.e., a discrete-time PID controller without the derivative component) whose output value $u_j(t)$ depends on the difference between the current core utilisation and the setpoint. The output value of $u_j(t)$ is sent

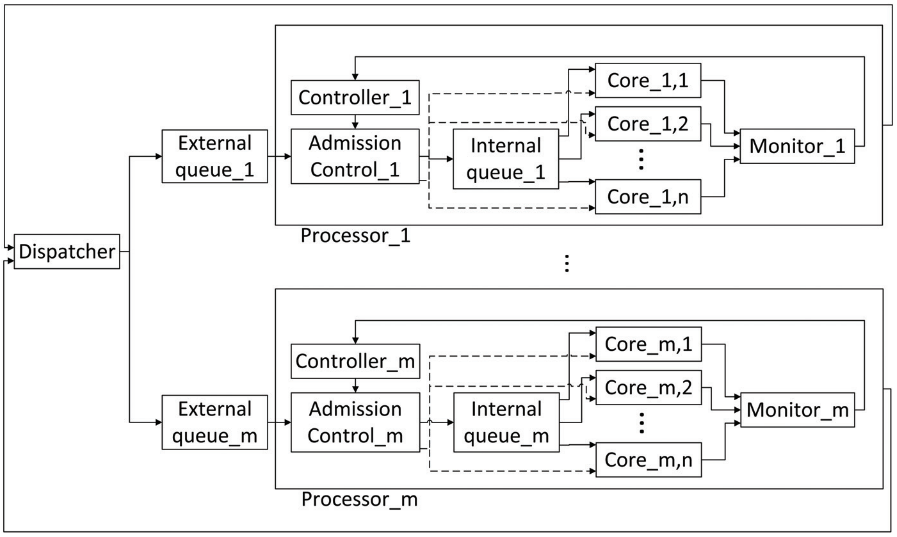


Figure 3.8 Distributed feedback control real-time allocation with DVFS architecture.

to admission control block AC_j, where it is used to perform a task admittance decision.

The role of block AC_j is to decide whether a task T_i, fetched from the j-th external input queue, should be executed by the processor. The first condition of admittance is that the deadline of T_i, D_i, is not lower than the sum of its worst-case computation time, C_i and the current simulation time t. Then the output controller value, $u_j(t)$, is checked and it influences the decision of the task rejection or admission as described in the next paragraph. The admitted tasks are placed in the internal processor queue. This queue shall be rather short to minimise the delay between decision about admittance and the execution of the task, and to keep the timeliness of the lateness input.

The additional role of block AC_j is to scale the voltage of the cores. The controller output value, $u_j(t)$, is tested against two threshold values $+\Upsilon$ and $-\Upsilon$. If $u_j(t) > +\Upsilon$, the processing cores are more utilised than the setpoint r for relatively long period (depending on the I-Window length and k_i value) and thus increasing the frequency (and voltage) of the set of cores is desirable. On the other hand, if $u_j(t) < -\Upsilon$, the processing cores are too idle for relatively long period and it is recommended to decrease the frequency (and voltage) of the cores to conserve energy. It is important to select the value of Υ wisely, taking into account that $u_j(t)$ depends on the current error value (multiplied by k_p) and on the sum of the previous errors (multiplied by k_i) and the length of I-Window used during this sum calculation. After choosing these three values, it is possible to assign an appropriate value to this threshold. Identification of these values and the threshold is performed in Section 3.3.

Since in any core transferring between various voltage levels is penalised both in terms of switching time and energy [52], some mechanism preventing too frequent transitions is needed. In our case, we decided to use threshold Γ, which determines the minimal time between two consecutive voltage level alterations. Each P-State change request issued earlier than Γ is ignored. This value should be determined by taking into account the hardware parameters as a trade-off between the system flexibility (lower parameter value) and efficiency (higher parameter value), which is presented in Section 3.3.

The proposed admission control algorithm is composed in two parts, described respectively by lines 1–28 and 29–36 in Figure 3.9, which are executed concurrently. The first part consists of the following steps.

Step 1. Invocation and initialization (lines 1–3, 27): The block functionality is executed in an infinite loop (line 1), activated every time interval Δt (line 27). The current P-State is set to the lowest value (i.e., the highest performance – line 2), and the time of the previous P-State change, γ, is set to 0 (line 3).

Step 2. Task fetching and schedulability analysis (lines 4–5): The tasks input FIFO queue is checked if empty (line 4) and a task T_i is fetched (line 5).

```
Inputs: Task $T_i$ (from Task queue)
    Controller output value $u$
    Admission controller invocation periods $\Delta t$
Outputs: Task executing or rejection decision
    New P-State of Cores
Constants: $P_{max}$ - maximal P-State available in processor
    $\Upsilon$ - threshold value of cumulated error from controller
    $\Gamma$ - minimal time elapsed between P-State change
Variables: $P$ - current P-State
    $\gamma$ - time of the previous P-State change

1:  while (true) do
2:      $P = 0$
3:      $\gamma = 0$
4:      while (task queue is not empty) do
5:          Fetch $T_i$
6:          if ($P = 0$ and $u < 0$)
            or ($u < 0$ and $current\_time \leq \gamma + \Gamma$) then
7:              if $P > 0$ and $current\_time > \gamma + \Gamma$ then
8:                  $P = P - 1$
9:                  $\gamma = current\_time$
10:                 Clear I-Window in Controller
11:             end if
12:             Reject task $T_i$
13:         else
14:             if $u < -\Upsilon$ and $P > 0$
                and $current\_time > \gamma + \Gamma$ then
15:                 $P = P - 1$
16:                 $\gamma = current\_time$
17:                 Clear I-Window in Controller
18:             else if $u > +\Upsilon$ and $P < P_{max}$
                and $current\_time > \gamma + \Gamma$ then
19:                 $P = P + 1$
20:                 $\gamma = current\_time$
21:                 Clear I-Window in Controller
22:             end if
23:             Admit task $T_i$
24:             Send $T_i$ to FIFO
25:         end if
26:     end while
27:     Wait $\Delta t$
28: end while

29: while (true) do
30:     if (task queue is empty for $\Gamma$ and $P < P_{max}$) then
31:         $P = P + 1$
32:         Clear I-Window in Controller
33:         $\gamma = current\_time$
34:     end if
35:     Wait $\Delta t$
36: end while
```

Figure 3.9 Pseudo-code of the proposed admission controller functionality.

Step 3. Task conditional rejection (lines 6–12): If the output value of controller (u) is negative and the cores operate with the highest performance

(P-State set to P0) or the cores operate below the highest performance and the previous change of P-State (at time γ) was done early enough (determined by condition $current\_time > \gamma + \Gamma$), task T_i should be rejected (line 6). Moreover, if P-State is different from P0 and there was no recent change of P-State (line 7), P-State is decreased (line 8) and variable keeping the previous P-State change time, γ, is set accordingly (line 9). The buffer storing the previous error values of the controller, I-Window, used by the integral component of the PID controller, is cleared (line 10), since the previous errors have been obtained in a different P-State and thus should not influence future admittance decisions.

Step 4. Task conditional admittance (lines 14–25): If the controller output value is below threshold $-\Upsilon$, the processor performance is not the highest possible and the previous change of P-State was done early enough (line 14), P-State is decreased (line 15) and the $current\_time$ is substituted to γ (line 16). Similarly, provided the controller output value is above threshold $+\Upsilon$, the processor performance is not the lowest possible (P-State is different from P_{max}, the highest P-State available in the processor) and the previous change of P-State was done early enough (line 18), the processor P-State is increased (line 19) and the current time is assigned to γ (line 20). Task T_i is sent to the FIFO queue (line 24).

The second part of the algorithm consists of the following steps.

Step 1. Invocation (lines 29, 35): The block functionality is executed in an infinite loop (line 29), activated every time interval Δt (line 35).

Step 2. P-State conditional increase (lines 30–33): If no new tasks have been fetched from the task queue for time Γ and the processor performance is not the lowest possible (P-State is different from P_{max}), the processor P-State is increased (line 31) and the current time is assigned to γ (line 33).

3.6 Experimental Results

In order to check the efficiency of the proposed feedback-based admission control and real-time task allocation, we developed a simulation model using SystemC language.

3.6.1 Controller Tuning

Firstly, the controller constant components k_p, k_i and k_d have to be tuned by analysing the corresponding open-loop system response to a bursty workload. Then random workloads of various weight have been tested to observe the system behaviour under different conditions and to find the most beneficial operating region.

To tune the parameters of the controller, the task slack growth after applying a step-input in the open-loop system (i.e., without any feedback) has been analysed, as mentioned earlier in this chapter. This is a typical way in control-theory-based approaches [7]. The workload used for this case consists of 500 independent tasks. They are split into five groups. Tasks belonging to one group are released every 5 ms each. After this bursty activity, during the following 500 ms no task is released. Then the tasks of the next group are released at the same pace. This process is repeated until all tasks from all groups are released. The computation time of each task is equal to 50 ms and its deadline is set to the sum of computation time multiplied by an arbitrary constant (equal to 1.5) and the task release time. This constant has been introduced to provide some flexibility in task scheduling; otherwise, all tasks would be required to start execution at their release time to meet the timing constraints.

The obtained results have confirmed the accumulating (integrating) nature of the process, and thus the accumulating process version of Approximate M-constraint Integral Gain Optimization (AMIGO) tuning formulas have been applied to choose the proper values of the PID controller components [7]. As a reference point, we executed a simulation without DVFS on a system comprised of one processor with three cores. As many as 140 tasks have been executed before the deadline, no task missed its deadline, and 360 tasks have been rejected by the dispatcher.

To use the DVFS features efficiently, it is crucial to find an appropriate value of threshold Υ. It should be large enough not to switch a core voltage too frequently – the switching should be performed not only due to the high value of $u(t)$ generated by the proportional component, but also with the relatively large value of the integral component, meaning that the error has been large for a longer interval. Simulations results for selected values of Υ are presented in Table 3.3. Form this table it follows that too high values of Υ result in keeping the current frequency too long at the beginning of a busy period, decreasing the performance of the system significantly (see the number of tasks executed before the deadline for $\Upsilon \in (30, 60)$). Particularly, keeping the lowest frequency too long results in executing some tasks after their deadlines. At some point ($\Upsilon = 70$ in the considered case), the threshold is too high for the given idle period and it does not manage to perform any voltage scaling before the next busy period. For further experiments, based on the above observations, we have chosen $\Upsilon = 10$ as a trade-off between performance and flexibility of the voltage switching.

In order to determine the threshold Γ of the task number that has to be processed by the dispatcher between subsequent alterations of the core voltage, we performed a series of simulation, where the threshold ranged from 25 ms to 400 ms. The highest number of tasks executed before their deadlines is

Table 3.3 Total number of tasks executed before and after their deadlines and rejected tasks with various Υ threshold in the introductory experiment (1 processor with 3 cores)

Threshold Υ	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
5	120	0	380
10	120	0	380
20	120	0	380
30	64	84	352
40	52	92	356
50	52	92	356
60	52	92	356
70	140	0	360
∞	140	0	360

observed with threshold $\Gamma \in \{25 \text{ ms}, 50 \text{ ms}, 100 \text{ ms}\}$. The threshold set to 350 ms or above leads to the behaviour not differentiated from the simulation without DVFS. Two values $\Gamma = 50$ ms and $\Gamma = 300$ ms have been used in the further experiments.

To estimate the energy used by a processor, ACPI data for Pentium M processor (with Intel SpeedStep Technology) has been used, but with slight modification the proposed technique can be applied to any processor with ACPI implemented[2]. In Pentium M, there are six levels of allowed frequency and voltage pairs, known as P-States. In P-State P0, a core works with 1.6 GHz and 1.484 V, whereas for P5 – 600 MHz and 0.956 V, which uses 24.5 W and 6 W, respectively.

3.6.2 Random Workloads

Having selected all the required constants, the efficiency of the system has been checked against 11 sets of 10 random workloads, whose release and execution time probability distributions are based on the grid workload of an engineering design department of a large aircraft manufacturer, as described in [22]. Each workload is comprised of 100 tasks, including a random number (between 1 and 20) of independent jobs. The execution time of every job is selected randomly between 1 and 99 ms. All jobs of a task are released at the same instant, and the release time of the next task is selected randomly between $r_i + range\_min \cdot C_i$ and $r_i + range\_max \cdot C_i$, where C_i is the total worst case computation time of the current tasks T_i released at r_i, and $range\_min, range\_max \in (0, 1)$, $range\_min < range\_max$. These values are inversely proportional to the workload weight.

[2] For example AMD Family 16h Series Processors ACPI parameters are provided in AMD Family 16h Models 00h – 0Fh Processor Power and Thermal Data Sheet, AMD, 2013.

We have measured the numbers of tasks computed before their deadlines and the number of tasks rejected by the admission controller block in a 3-processor system with 3 processing cores and 2-processor system with 4 processing cores each for systems with DVFS and without DVFS (i.e., with $\Gamma = \infty$) and presented it in Figures 3.10 and 3.11, respectively.

The number of tasks admitted with $\Gamma = 300$ ms is, in total, 26% higher than with $\Gamma = 50$ ms. The reason for this is that in case of $\Gamma = 50$ ms, P-States are changed more often and thus it is more likely to have a processor with a lower frequency and voltage level while a task is fetched, and since decrease of P-States is performed gradually (lines 8, 15, 19 and 31 in the algorithm in Figure 3.9), tasks are attempted to be executed with lower processor

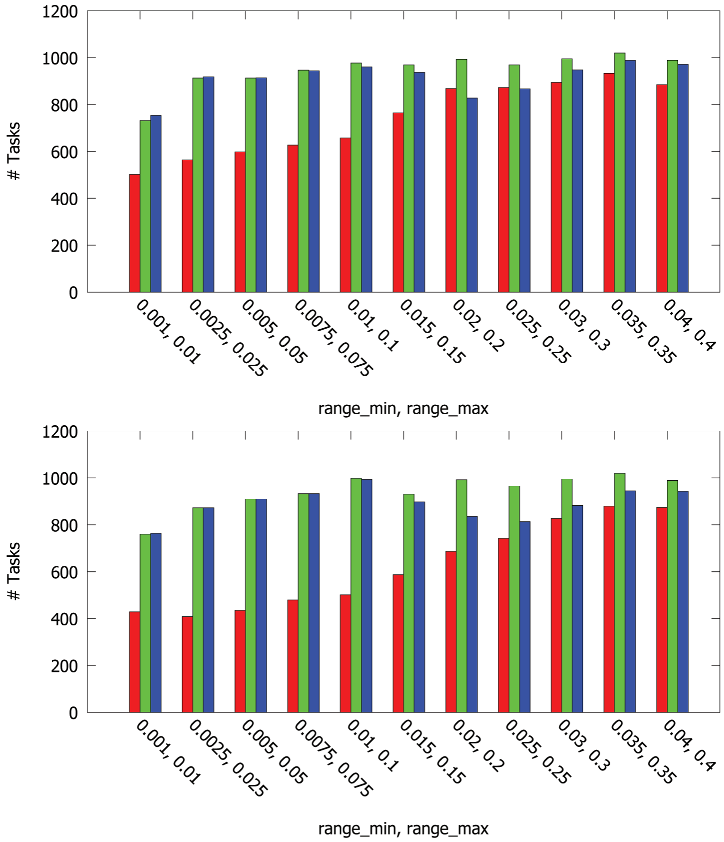


Figure 3.10 Tasks executed before their deadline in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

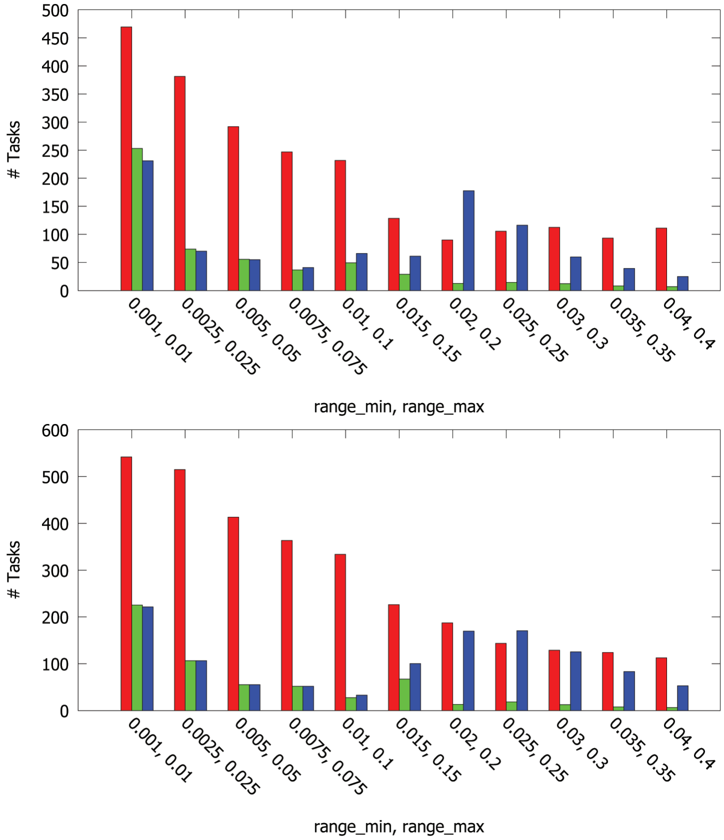


Figure 3.11 Tasks rejected in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

performance. It has been observed that this strategy leads to significant (about 39%) energy reduction. It may be, however, surprising, that the number of the executed tasks is higher with DVFS when $\Gamma = 300$ ms than in the system without DVFS for lighter workloads. This phenomena is innate to the proposed technique and can be explained using the pseudo-code in Figure 3.9. In a system without DVFS, each processor is always in its lowest P-State, P0. The admission controller has then no flexibility in decreasing the P-State while the Controller output is negative (checked in line 6) and then to clean the I-Window in the PID controller and, finally, admit the task (line 21).

Different size of the systems does not influence the relationship between obtained results. The number of tasks executed before deadlines for assorted weights is almost linearly dependent between the two considered architectures

(Pearson Correlation Coefficient $\rho = 0.96$; similarly for the number of rejected tasks $\rho = 0.97$, and for dissipated energy $\rho = 0.98$).

The dynamic energy dissipation, normalised with respect to the highest obtained value during the experiment, is presented in Figure 3.12. In general, almost 58% of the dissipated dynamic energy have been saved via the DVFS approach. For heavier loads from the random workloads, choosing a lower Γ value leads to significant energy reduction, whereas for lighter loads the result difference between the two chosen Γ values is almost negligible.

Looking at the normalized energy dissipation per task (Figure 3.13), computed for the system with three processors, it can be concluded that parameter $\Gamma = 50$ ms leads to more even energy per task usage in comparison with $\Gamma = 300$ ms, which is slightly more beneficial for lighter workloads only,

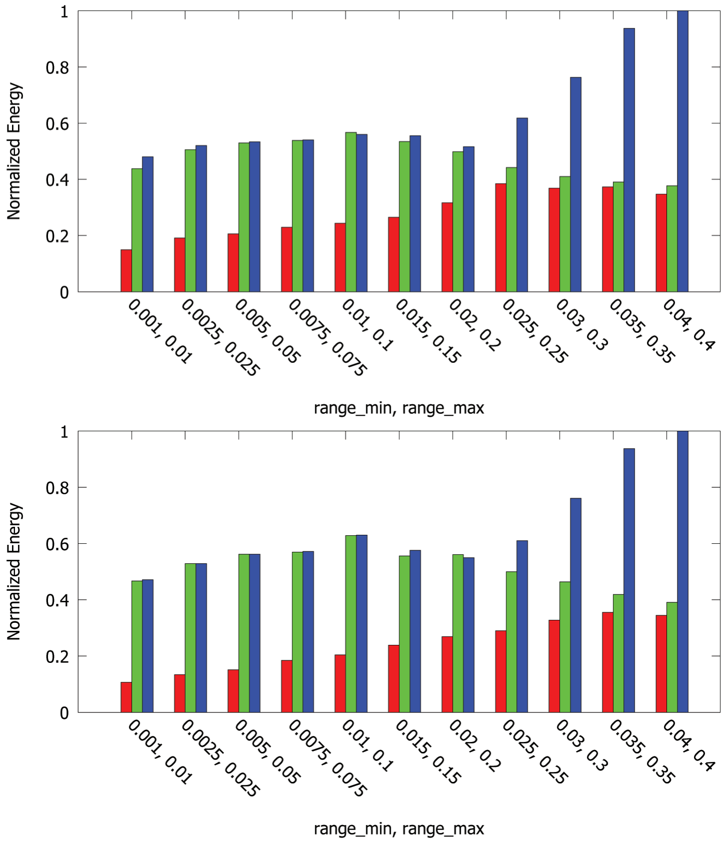


Figure 3.12 Normalized dynamic energy dissipation in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

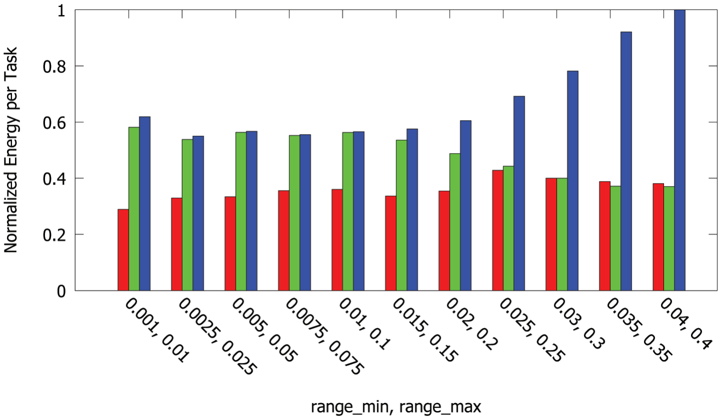


Figure 3.13 Normalized dynamic energy dissipation per task meeting its deadline in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue).

but leads to similar energy per task value than in the system without DVFS (i.e., $\Gamma = \infty$) for heavier loads.

3.7 Related Work

Dynamic real-time scheduling (RTS) algorithms like EDF and rate monotonic support RTS characteristics (worst-case computation time, release rate, etc.), but they remain open-loop: once the schedule is built, it stays fixed [80]. Such algorithms perform well in meeting QoS levels in predictable systems. However, their performance degrade when dealing with unpredictable workloads which cannot be modelled accurately a priori [2]. In unpredictable systems, sophisticated schedulers like Spring depends on worst-case parameters which can result to resource under-utilisation based on pessimistic estimation of workloads [135].

It is desired to feed the system states back to the scheduler [26, 121], so it can be aware of sudden/unpredicted changes and act accordingly in order to meet the QoS levels. A system state can be defined as the system performance with respect to service capacity, QoS levels etc. Real-time systems analysis includes observing how tasks' RTS characteristics affect (1) meeting QoS levels e.g., high processor utilisation, and (2) compute resource availability. This can help adapting towards the varying system states by enforcing scheduling decisions [26]. This variance can be considered as system error which is the deviance from the system output and the desired one (QoS level(s)). Some related work like [26, 78] show the lack of adaptivity to varying system states in traditional RTS algorithms.

In the realm of control-theoretic RTS algorithms, there are adaptive approaches that are cost-effective for performance guarantees in systems with varying system states [26, 80]. For instance, Lu offers regulating the workload of a single-CPU RTS system via a PID-based admission control (PID-AC) algorithm to reduce the deadline miss ratio [80]. His algorithm guaranteed 95% CPU utilisation and 1% deadline miss ratio in comparison to an EDF algorithm with 100% and 52% respectively. PID-AC algorithms are plausible in some RTS systems where controlling tasks release rate is difficult, instead, the scheduler rejects specific tasks to meet QoS levels.

Also, in [81], Lu uses two PID controllers to meet two QoS levels; maximum CPU utilisation and minimum deadline miss ratio. The results confirm the findings of [80] in ensuring performance guarantees as opposed to basic EDF algorithm. The motivation behind Lu's 2-PID algorithm was to address the stability and transient response issues with the single PID algorithm due to PID control limitations handling multiple QoS levels. Our work, in this chapter, addresses minimising dependent tasks' latency, not deadline miss ratio, via a PID-based admission control algorithm.

Control-theory-based voltage level selection of unicore portable devices has been firstly proposed in [106]. Varma et al. in [147] choose Proportional-Integral-Derivative (PID) controllers to determine voltage of systems dealing with the workload not accurately known in advance and interpreted the meaning of the discrete PID equation terms with regards to dynamic workloads. They proposed a heuristic to predict the future system load based on the workload rate of change, leading to significant energy reduction. They also demonstrated that the design space is not particularly sensitive to changes in the PID controller parameters. However, the controller is used to predict the future workload and does not use any feedback information from the system about the processing core status.

In [162], a feedback-based approach to manage dynamic slack time for conserving energy in real-time systems has been proposed. A PID controller is used to predict a real execution time of a task, usually lower than its worst case execution time (WCET). Then each execution time slot for a task is split into two parts and the first part is executed with a lower voltage assuming the execution time predicted by the controller. If the task does not finish by this time, a core is switched to its highest voltage guaranteing that the task finishes its execution before its deadline.

In [153] a formal analytic approach for DVFS dedicated to multiple clock domain processors benefits from the fact that the frequency and voltage in each functional block or domain can be chosen independently. A multiple clock domain processor is modelled as a queue-domain network where queue occupancies linking two clock domains are treated as feedback signals.

A clock domain frequency is adapted to any workload changes by a proportional-integral (PI) controller.

The queue occupancy also drives PI controllers in [31]. In contrast to previous research, the limitation of using single-input queues only have been lessened and multiple processing stages have been allowed, but a pipelined configuration is still required. A realistic cycle-accurate, energy-aware, multiprocessor virtual platform is used for demonstrating the superiority of feedback techniques over the local DVFS policies during simulation of signal processing streaming applications with soft real-time guarantees. It is assumed that as long as the queues are not empty, a sufficient number of deadlines is met and no further analysis or simulation of deadline misses are provided.

From the literature survey it follows that there is no previous work on mapping the task dynamically to an MPSoC system and using DVFS together with control-theory based algorithm.

3.8 Summary

In this chapter, we have explored the possibility of applying feedback controlled values to dynamic task allocation and admission control for high-performance computing clusters executing real-time tasks. Two real-time metrics, task lateness and core utilisation, have been applied to perform admission control, whereas the former has been also used as a metric for dynamic task allocation. Two queue types (EDF and FIFO) have been used in the open-loop system. The P and PI controllers have been tuned using classic AMIGO and Ziegler-Nichols methods.

We have prepared simulation models in SystemC language and performed a number of experiments. In case of uniform periodic workload the closed-loop system has executed almost 5 times more tasks before their deadline in comparison with an adequate open-loop system. The queue type and controller value have slightly influenced the outcome. The metric-based dynamic allocation, in the best configuration, has been about 8.5 per cent better than the round-robin method.

In the case of bursty random workloads with large computation time variance, a closed-loop-based system has been about 16% better than the corresponding open-loop approach. However, to proper asses the difference between these systems in more accurate way, the differentiate between tasks of long and short execution time should be introduced.

We have also explored the possibility of applying feedback control values to dynamic task allocation and admission control for multi-core processors supporting DVFS while executing firm real-time tasks. Core utilisation has been applied as a run-time metric to perform admission control and dynamic task allocation. The proposed governor algorithm has been tested with various parameter values and some guidance for tuning has been provided.

Even in case of relatively difficult bursty scenarios a significant power reduction has been obtained in exchange for executing lower number of tasks before their deadlines. It is a role of a system designer to choose proper parameter values to obtain a satisfiable trade-off between energy consumption and performance. The proposed approach leads to similar results in two considered systems of different sizes, thus it may be viewed as quite robust to different system configurations.

The minimal interval allowed between two consecutive switchings of P-States (threshold Γ) influences workloads of various weights in different, but predictable way. An adaptive choice of Γ value can be then viewed as a simple yet effective improvement of the proposed technique.

4

Feedback-Based Allocation and Optimisation Heuristics

The vast majority of existing research into hard real-time scheduling on many-core systems assumes workloads to be known in advance, so that traditional scheduling analysis can be applied to check statically whether a particular taskset is schedulable on a given platform [42]. The hard real-time scheduling is desired in several time critical systems such as atomotive and aerospace domains [56]. Under dynamic workloads, admitting and executing all hard real-time (HRT) tasks belonging to a taskset can jeopardise system timeliness. The decision of task admittance is made by *admission control*. Its role is to fetch a task from the task queue and check whether it can be executed by any core before its deadline and without forcing existing tasks to miss theirs. If the answer is positive, the task is admitted, and rejected otherwise. The benefits of this early task rejection are twofold: (*i*) the resource working time is not wasted with a task that will probably violate its deadline, and (*ii*) a possibility of early signalling the lack of admittance can be employed to perform an appropriate precaution measures in order to minimize the negative impact of the task rejection.

Dynamic workloads do not necessarily follow the relatively simple periodic or sporadic task models and it is rather difficult to find a many-core system scheduling analysis that relies on more sophisticated models [42], [67]. Computationally-intensive workloads not following these basic models are more often analysed in High Performance Computing (HPC) domain, for example in [35]. The HPC community experience with these tasksets could help introducing novel workload models to many-core system schedulability analysis [42]. In HPC systems, tasks allocation and scheduling heuristics based on feedback control proved to be valuable for dynamic workloads [82], improving platform utilisation while maintaining timing constraints. Despite a number of successful implementations in HPC community, these heuristics are to the best of our knowledge never used in many-core embedded platforms with hard real-time constraints.

The Roadmap on Control of Real-Time Computing Systems [5], one of the results of the EU/IST FP6 Network of Excellence ARTIST2 program, states clearly that feedback scheduling is not suitable for applications with hard real-time constraints, since feedback acts on errors. However, further research [140, 162] show that although the number of deadline misses must not be used as an observed value (since any positive error value would violate the hard real-time constraints), observing other system's parameters, such as dynamic slack, created when tasks are executed earlier than their worst-case execution time (WCET), or core utilisation, could help in allocating and scheduling tasks in a real-time system.

The feedback-based dynamic resource allocation heuristics impose some requirements on the target system. Usually, to perform resource allocation decision one can rely on various metrics provided by the monitoring infrastructure tools and services, such as utilization and the time latency between input and output timestamps [81]. The system should also guarantee an appropriate level of responsiveness to the decisions made by the heuristics, as well as update the values of the metrics used as inputs in the algorithm. Moreover, it is important to provide the heuristic algorithm with realistic data about system workload, service capacity, worst-case execution time and average end-to-end response [84].

In order to address the aforementioned issues, we present a novel task resource allocation process, which is comprised of the *resource allocation* and *task scheduling*. The *resource allocation* process is executed on a particular core. Its role is to send the processes to be executed to other processing cores, putting them into the task queue of a particular core. Task scheduling is carried out locally on each core and selects the actual process to run on the core. The proposed approach adopts control-theory based techniques to perform runtime admission control and load balancing to cope with dynamic workloads with hard real-time constraints. It is worth stressing that, to the best of our knowledge, no control theory based allocation and scheduling method aiming at hard real-time systems has been proposed to operate in an embedded system with dynamic workloads.

4.1 System Model and Problem Formulation

In Figure 4.1 the consecutive stages of a task life cycle in the proposed system are presented. The task τ_l is released at an arbitrary instant. Then an approximate schedulability analysis is performed, which can return either fail or pass. If the approximate test is passed, the exact schedulability, characterised with a relatively high computational complexity [42], is performed. If this test is also passed, the task is assigned to the appropriate core, selected during the schedulability tests, where it is executed before its deadline.

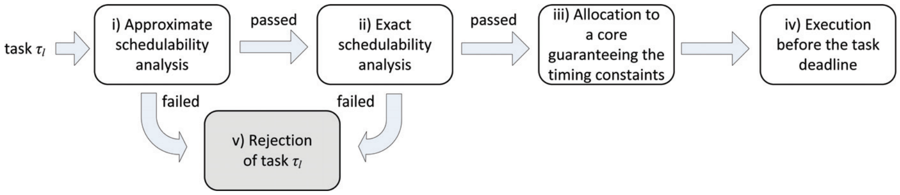

task τ_l
i) Approximate schedulability analysis
passed
ii) Exact schedulability analysis
passed
iii) Allocation to a core guaranteeing the timing constaints
iv) Execution before the task deadline
failed
failed
v) Rejection of task τ_l

Figure 4.1 Building blocks of the proposed approach.

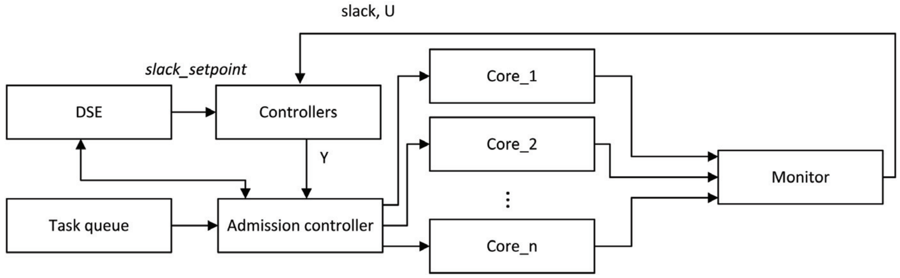


Figure 4.2 A proposed many-core system architecture.

4.1.1 Application Model

A taskset Γ is comprised of an arbitrary number of tasks, $\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots\}$ with hard real-time constraints. The j-th job of task τ_i is denoted with $\tau_{i,j}$. If a *task* is comprised of only one *job*, these terms are used interchangeably in this chapter. In case of tasks with job dependencies it is assumed that all jobs of a task are submitted at the same time, thus it is possible to identify the critical path at the instant of the task release. Periodic or sporadic tasks can be modelled with an infinite series of job. The taskset is not known in advance, thus the tasks can be released at any instant.

4.1.2 Platform Model

The general architecture of the proposed solution is depicted in Figure 5.3. The system is comprised of n cores, whose dynamic slacks (slack vector whose length $|\text{slack}| = n$) and busyness (vector U, $|\text{U}| = n$) are observed constantly by the Monitor block.

In the Controllers block, one discrete-time PID controller for each core is invoked every dt time. The controllers use dynamic slacks of the corresponding cores as the observed values.

The Admission controller block receives a vector of controllers' outputs, $\mathrm{Y} = [y_1(t), \ldots, y_n(t)]$, from the Controllers block. Based on its elements' values it performs, as shown in Figure 4.1, *(i) approximate schedulability analysis* of a task admittance or rejection decision. If the decision is positive, an *(ii) exact schedulability analysis* is performed by the Design Space Exploration (DSE) block. If at the second stage the result of the task schedulability analysis is negative, the task is rejected. Otherwise it is *(iii) allocated to a core where the execution before the deadline is guaranteed* based on the schedulability analysis performed in block DSE.

4.1.3 Problem Formulation

Given an application and platform models, the problem is to quickly identify tasks whose hard timing constraints would be violated by the processing cores and then to reject such tasks without performing costy exact schedulability analysis. The number of rejected tasks should be reasonably close to the number of tasks rejected in a corresponding open-loop system, i.e., the system without the early rejection prediction. Meeting the deadlines for all admitted tasks shall be guaranteed.

4.2 Performing Runtime Admission Control and Load Balancing to Cope with Dynamic Workloads

In dynamic workloads, admitting and executing all hard real-time (HRT) tasks belonging to a taskset G can jeopardise system timeliness. The role of the admission control is to detect the potential deadline violation of a released task, τ_l, and to reject it in such the case. Then the resource working time is not wasted for a task that would probably violate its deadline and early signaling of the rejection could be used for minimizing its negative impact.

The j-th job of task τ_i, $\tau_{i,j}$, is released at $r_{i,j}$, with the deadline $d_{i,j}$ and the relative deadline $D_{i,j} = d_{i,j} - r_{i,j}$. The slack for $\tau_{i,j}$ executed on core π_a, where $\tau_{p,k}$ was the immediate previous job executed by this core, is computed as follows:

$$s_{i,j} = \begin{cases} C_p - c_{p,k} & \text{if } r_{i,j} \leq I_{p,k} + c_{p,k}, \\ F_{p,k} - r_{i,j} & \text{if } I_{p,k} + c_{p,k} \leq r_{i,j} < F_{p,k}, \\ 0 & \text{if } r_{i,j} \geq F_{p,k}, \end{cases} \tag{4.1}$$

where $r_{i,j}$ is release time of $\tau_{i,j}$, $I_{p,k}$ – initiation time of $\tau_{p,k}$ (also known as the job execution starting time), $c_{p,k}$ and C_p – computation time and worst-case execution time (WCET) of $\tau_{p,k}$, and $F_{p,k}$ – its worst-case completion time. A similar slack calculation approach is employed in [162]. The three possible slack cases (Equation (4.1)) are illustrated in Figure 4.3 (top, centre,

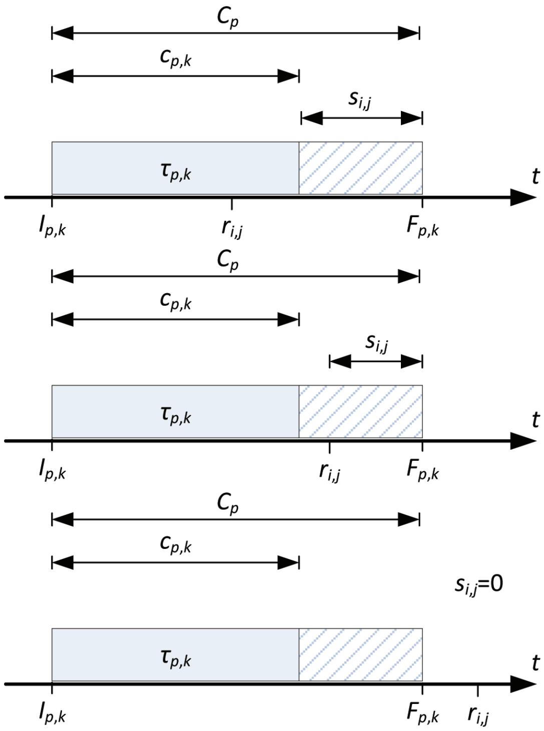


Figure 4.3 Illustration of task $\tau_{i,j}$ slack in three cases from Equation (4.1).

bottom, respectively). In these figures the solid rectangle illustrates execution time (ET) of $\tau_{p,k}$, whereas the striped rectangle shows the difference between WCET and ET of this task.

The normalised value of slack of currently executed job $\tau_{i,j}$ on core π_a is computed as follows:

$$slack_a = \frac{D_{i,j} - s_{i,j}}{D_{i,j}}. \tag{4.2}$$

This value is returned by a monitor and compared by a controller with setpoint $slack\_setpoint$. An error $e_a(t) = slack_a - slack\_setpoint$ is computed for core π_a, as schematically shown in Figure 5.3. Then the a-th output of the Controllers block, reflecting the past and previous dynamic slack values in core π_a, is computed with formula

$$y_a(t) = K_P e_a(t) + K_I \sum_{i=0}^{IW} e_a(t-i) + K_D \frac{e_a(t) - e_a(t-1)}{dt}, \tag{4.3}$$

where K_P, K_I and K_D are positive constant components of the proportional, integral and derivative terms of a PID controller. Their values are usually determined using one of the well-known control theory methods, such as root locus technique, Ziegler-Nichols tuning method or many others, to obtain the desired control response and preserve the stability. In our research, we have applied Approximate M-constrained Integral Gain Optimisation (AMIGO), as it enables a reasonable compromise between load disturbance rejection and robustness [8]. This method has been outlined in Chapter 3.

The value of $slack\_setpoint$ is bounded between values: $min\_slack\_setpoint$ and $max\_slack\_setpoint$, which should be chosen appropriately during simulation of a particular system. Similarly, the initial value of $slack\_setpoint$ can influence (slightly, according to our experiments) the final results. In this chapter, it is initialised with the average between its minimal and maximal allowed values to converge quickly with any value from the whole spectrum of possible controller responses.

The slacks of the tasks executed by a particular processing core accumulate as long as the release times of each task are lower than the worst-time completion time of the previous task, which correspond to the first two cases in Equation (4.1) and are illustrated in Figure 4.3 (top and centre). It means that the slacks of subsequent tasks executed on a given core can be used as a controller input value. However, previous values of dynamic slack are of no importance when the core becomes idle, i.e., the core finishes execution of a task and there is no more tasks in the queue to be processed, which corresponds to the third case in Equation (4.1) illustrated in Figure 4.3 (bottom). To reflect this situation, the current value of $slack\_setpoint$ is provided as an error $e_a(t)$, to enhance the task assignment to this idle core (since it corresponds with the situation that the normalised slack would be twice as large as the current setpoint value, i.e., behaves in the way the task would finish its execution two times earlier than expected). Substituting this value not only positively estimates the task schedulability at the given time instant, but also influences future computation of the controller output, as it appears as a prior error value in the integral part in Equation (4.3).

The Controllers block output value $\mathrm{Y} = [y_1(t), \ldots, y_n(t)]$ is provided as an input to the Admission controller block, where it is used to perform a task admittance decision. If all Controllers' outputs (errors) $y_a(t), a \in \{1, \ldots, n\}$ are negative, the task τ_l fetched from the Task queue is rejected. Otherwise, a further analysis is conducted by the Design Space Exploration (DSE) block to perform exact schedulability analysis. The available resources are there checked according to any exact schedulability test (e.g., from [42]), which is performed for each core with task τ_l added to its taskset as long as a schedulable assignment is not found. In our implementation, this analysis has been carried out using the interval algebra described in Chapter 2. If no resource is found that guarantees the task execution before its deadline, it is rejected.

The pseudo-code of the control strategy is presented in Algorithm 3.9. This algorithm is comprised of two parts, described respectively by lines 1–18 and 19–24, which are executed concurrently. The first part consists of the following steps.

- **Step 1. Invocation (lines 1, 17).**
  The block functionality is executed in an infinite loop (line 1), activated every time interval dt (line 17).

Algorithm 4.1 Pseudo-code of Admission controller involving DSE algorithm

```
inputs    : Task $\tau_l \in \Gamma$ (from Task queue)
            Vector of errors $\mathrm{Y}[1..n]$ (from Controller)
            Controller invocation period $dt$
            $slack\_setpoint$ decrease period $dt_1$, $dt_1 > dt$
outputs   : Core $\pi_a \in \Pi$ executing $\tau_l$ or job rejection
            Value of $slack\_setpoint$
constants: $min\_slack\_setpoint$ - minimal allowed value of $slack\_setpoint$
            $max\_slack\_setpoint$ - maximal allowed value of $slack\_setpoint$
            $slack\_setpoint\_add$ - value to be added to $slack\_setpoint$
            $slack\_setpoint\_sub$ - value to be subtracted from $slack\_setpoint$

1  while true do
2      while task queue is not empty do
3          fetch $\tau_l$;
4          forall $\mathrm{Y}_a > 0$ do
5              if taskset $\Gamma_a \cup \tau_l$ is schedulable then
6                  assign $\tau_l$ to $\pi_a$;
7                  break;
8              end
9              if $\tau_l$ not assigned then
10                 reject $\tau_l$;
11                 if $\exists \mathrm{Y}_a : \mathrm{Y}_a > 0 \wedge slack\_setpoint < max\_slack\_setpoint$ then
12                     increase $slack\_setpoint$ by $slack\_setpoint\_add$;
13                 end
14             end
15         end
16     end
17     wait $dt$;
18 end

19 while true do
20     if $slack\_setpoint > min\_slack\_setpoint$ then
21         decrease $slack\_setpoint$ by $slack\_setpoint\_sub$;
22     end
23     wait $dt_1$;
24 end
```

- **Step 2. Task fetching and schedulability analysis (lines 2–8).**
  All tasks present in the Task queue are fetched sequentially (lines 2–3). For each task, the Controllers' outputs are browsed to find positive values, which are treated as an early estimation of schedulability (line 4). If such value is found in an a-th output, an exact schedulability test checks the schedulability of the taskset Γ_a of the corresponding core π_a extended with task τ_l using any exact schedulability test (line 5), e.g., from [42]. If the analysis proves that the taskset is schedulable, τ_l is assigned to π_a (line 6). Otherwise, the next core with the corresponding positive output value is looked for.
- **Step 3. Task rejection and setpoint increase (lines 9–15).**
  If all cores have been browsed and none of them can admit τ_l due to either a negative controller output value or the exact schedulability test failure, the task τ_l is rejected (line 10). In this case, if there exists at least one positive value in the Controllers' output vector, the $slack\_setpoint$ is increased by constant $slack\_setpoint\_add$ provided that it is lower than constant $max\_slack\_setpoint$ (lines 11–12) to improve the schedulability estimation in future.

The second part consists of two steps.

- **Step 1. Invocation (lines 19, 23).**
  The block functionality is executed in an infinite loop (line 19), activated every time interval dt_1, $dt_1 > dt$ (line 23).
- **Step 2. Setpoint decrease (lines 20, 21).**
  The value of $slack\_setpoint$ is decreased by constant $slack\_setpoint\_sub$ (provided that it is higher than constant $min\_slack\_setpoint$), which encourages a higher number of tasks to be admitted in future.

4.3 Experimental Results

In order to check the efficiency of the proposed feedback-based admission control and real-time task allocation process, a simple Transaction-Level Modelling (TLM) simulation model has been developed in SystemC language. Firstly, the controller components K_P, K_I and K_D have to be tuned by analysing the corresponding open-loop system response to a bursty workload. Then three series of experiments have been performed. Firstly, a heavy periodic workload has been used to observe the behaviour of the overloaded system. Due to the regularity in the workload, some convergence of the setpoint has been expected. In the second series, workloads of various weight have been tested to observe the system behaviour under different conditions and to find the most beneficial operating region. Then industrial workloads with dependent jobs have been used to determine the applicability of the proposed approach in real-life scenarios.

To tune the parameters of the controller, the task slack growth response on a step-input in the open-loop system (i.e., without any feedback) has been analysed. This is a typical way in control-theory-based approaches [8]. As an input, a burst release of 500 tasks (with execution time equal to 50 μs each) has been chosen. The modelled system has been comprised of 3 computing (homogeneous) cores. However, any number of tasks can be released, their execution time may vary and the number of cores can be higher, which is shown in further experiments. The obtained results have confirmed the accumulating (integrating) nature of the process, and thus the accumulating process version of AMIGO tuning formulas have been applied to choose the proper values of PID controller components [8], similarly as it has been presented in Chapter 3. With a series of trial-and-error processes, the following constant values have been selected: $min\_slack\_setpoint = 5$, $max\_slack\_setpoint = 95$, $slack\_setpoint\_add = 1$, $slack\_setpoint\_sub = 5$, the first part of the proposed algorithm (Algorithm 3.9) is executed five times more often than the second one.

During the first experiment, the system with the chosen parameters has been experimentally evaluated under a periodic workload, consisting of 900 independent jobs (i.e., each task is comprised of a single job only), one released every 5 μs, whose WCET equals to 50 μs and the relative deadline is equal to 60 μs. These parameters have been chosen appropriately to make the taskset heavy enough to saturate the system. The exact schedulability test has been performed for each task passing the early estimation based on the controller's output value. The systems with the number of processing cores ranging from 1 to 11 have been considered. The real execution time (ET) of each task varies randomly between 60% and 100% of its WCET, which results in creation of a dynamic slack. In the schedulability analysis, since the already executed tasks may influence the execution of the task whose schedulability is being tested, it is less pessimistic but still safe to provide the ET of these tasks instead of their WCET.

The regularity of the workload should cause convergence of the setpoint and decrease the variance of the normalised slack time. If the dynamic slack time normalised to task deadlines is close to 0%, it can be treated as an indication of well-chosen admission controller algorithm and the controller parameters, since it implies that the controller managed to minimize the steady-state error. The time needed to obtain this steady state indicates the responsiveness of the system, which should not be too long.

To check the system response to tasksets of various heaviness, nine sets of 10 random workloads have been generated. Each workload is comprised of 100 tasks, including a random number (between 1 and 20) of independent jobs. The execution time of every job is selected randomly between 1 and 99 µs. All jobs of a task are released at the same instant, and the release time of the subsequent task is selected randomly between $r_i + range\_min \cdot C_i$ and $r_i + range\_max \cdot C_i$, where C_i is the total worst-case computation time of the current tasks τ_i released at r_i, and $range\_min, range\_max \in (0, 1)$, $range\_min < range\_max$. These values influence the workload heaviness which can be described with $Param$ parameter, which we define as the total execution time of all jobs divided by the latest deadline of these jobs. For example, for pair $range\_min = 0.001$, $range\_max = 0.01$, ten random workloads have been generated with $Param$ ranging from 208.65 to 237.62, with the average value $\overline{Param} = 224.52$. The average $Param$ values for the generated workloads are given in Table 4.1. The value of $\lceil Param \rceil$ can be viewed as a lower bound of the number of cores needed for computing all tasks in the workload before their deadlines. It is a rather optimistic value due to the bursty nature of the workloads and their deadlines. For example, only 71% of tasks are executed before their deadlines from a certain generated workload with $Param = 4.58$ in an open-loop 5-core system, whereas to execute all these tasks as many as 13 cores are needed.

4.3.1 Number of Executed Tasks, Rejected Tasks and Schedulability Tests

4.3.1.1 Periodic workload

The number of tasks executed before their deadlines while using both ET and WCET for schedulability analysis has been compared with the corresponding open-loop system in Figure 4.4 (top). As expected, by using actual execution time (ET), the number of tasks executed before their deadlines is slightly increased. On average, an improvement of 3.7% is achieved. The results obtained by closed-loop approaches are clearly worse than those obtained with the open-loop approach, where schedulability of each task is analysed

Table 4.1 Average $Param$ values for random workloads generated with different $range\_min$ and $range\_max$ parameters

$range\_min$	$range\_max$	$\overline{Param}$
0.001	0.01	224.52
0.0025	0.025	77.07
0.005	0.05	38.71
0.0075	0.075	25.56
0.01	0.1	18.90
0.02	0.2	9.11
0.03	0.3	6.12
0.04	0.4	4.60

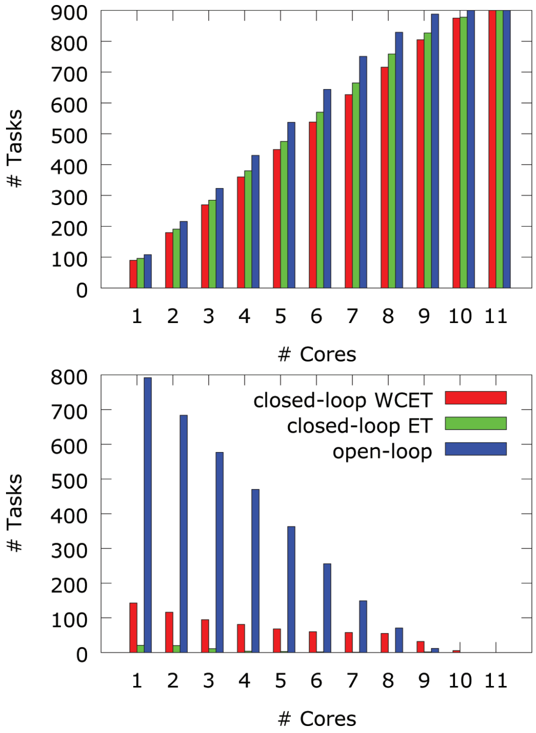


Figure 4.4 Number of executed tasks (*top*) and number of tasks rejected by the exact schedulability test (*bottom*) in closed-loop WCET, closed-loop ET and open-loop systems for the periodic task workload simulation scenario.

with an exact schedulability test only. The open-loop approach admits about 7.6% and 10.9% higher number of tasks than the closed-loop ET and WCET case, respectively. However, this improvement is achieved with a significant timing overhead. In an extreme case of one core system, 117 schedulability tests are to be conducted for the closed-loop ET case (and 233 for WCET) in comparison with 900 test executions in the open-loop system. It is important to note that only 21 exact schedulability tests for the ET case (and 143 for WCET) returned negative results, which demonstrates high accuracy of the proposed estimation scheme for open loop system. For higher number of cores, the differences are lower since each admitted task is to be checked by the exact schedulability test to ensure its hard deadlines compliance. On average, more than 38% and 34% of the schedulability tests can be omitted for the estimation based on ET and WCET, respectively. This difference is caused by the number of tasks rejected by the exact schedulability test, which is illustrated in Figure 4.4 (bottom).

4.3.1.2 Random workload

Figures 4.5 and 4.6 present the number of tasks computed before their deadlines, rejected tasks and the number of the exact schedulability test executions with respect to the number of processing cores (ranging from 1 to 9) and average values of $Param$, respectively, for open-loop and closed-loop (ET) systems.

The numbers of executed tasks with respect to $Param$, both for the open-loop and closed-loop systems, are approximated better with power than linear regression (residual sum of squares is lower by one order of magnitude in case of power regression; logarithmic and exponential regression approximations were even more inaccurate). This regression model can be then used to determine the trend of executed task number with respect to different workload weights. Similarly, the difference between the number of admitted tasks by open and closed loop systems can be relatively accurate approximated with a power function (power regression result: $y = 960.87x^{-1.18}$, residual sum of squares $rss = 3646.06$). This relation implies that the closed-loop system admits relatively low number of tasks when the workload is light. In such lightweight condition, the number of schedulability tests to be performed is only 12% lower in the extreme case of the set with $\overline{Param} = 4.60$. Thus, there is no reasonable benefits of using controllers and schedulability estimations. In heavier loaded systems, however, the number of admitted tasks in both configurations are more balanced, and the number of schedulability test executions is significantly varied. For example, for the two heaviest considered workload sets (i.e., with $\overline{Param}$ equal to 224.52 and 77.07) the schedulability tests are executed about 65% rarer in the closed-loop system.

The number of executed tasks grows almost linearly with the number of processing cores in both configurations and the slopes of their linear regression approximations (both with correlation coefficients higher than 0.99) are almost equal. This implies that both configurations are scalable in a similar way and the difference between the number of executing tasks in open-loop and closed-loop systems is rather unvarying. The number of schedulability test executions is almost constant in the open-loop system regardless the number of cores. However, for the closed-loop configuration, it changes in a way relatively good approximated with a power regression model (power regression result: $y = 1476.29x^{-0.30}$, residual sum of squares $rss = 14216.21$). Since the growing number of processing cores corresponds to less computation on each of them, the conclusion is similar as in the $Param$ variation case: the higher the load for the cores, the more beneficial is applying of the proposed scheme.

The number of tasks rejected in the open-loop systems using the exact schedulability test is considerably higher for heavier random workloads and

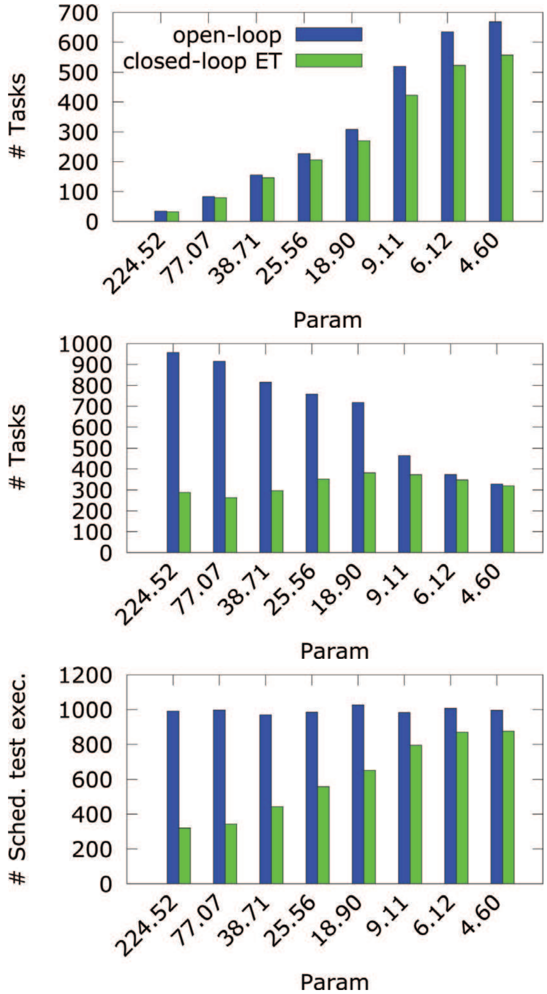


Figure 4.5 Number of tasks executed before their deadlines (*top*), the number of rejected tasks (*centre*) and number of the exact schedulability test executions (*bottom*) in baseline open-loop and proposed closed-loop ET systems for the random workloads simulation scenario with different weight of workloads.

lower number of cores, whereas for lighter random workloads or higher number of cores it is similar to the closed-loop ET system. For the closed-loop ET systems these figures illustrate the number of false positive errors of the approximate schedulability analysis, whereas for the open-loop systems it complements the number of tasks executed before deadlines.

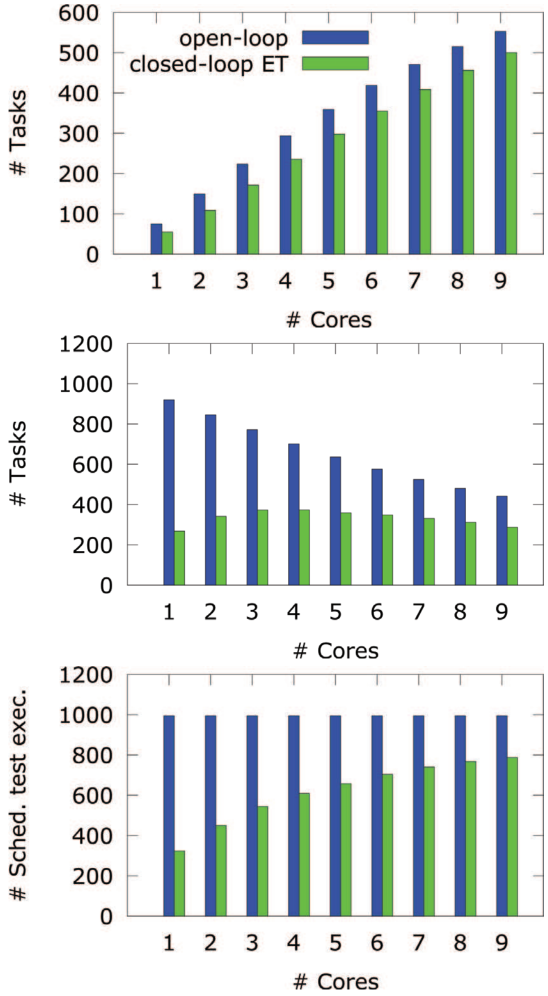


Figure 4.6 Number of tasks executed before their deadlines (*top*), the number of rejected tasks (*centre*) and number of the exact schedulability test executions (*bottom*) in baseline open-loop and proposed closed-loop ET systems with different number of processing cores for the random workloads simulation scenario.

4.3.2 Dynamic Slack, Setpoint and Controller Output

4.3.2.1 Periodic workload

Figures 4.7 shows dynamic slack, setpoint and controller outputs during the simulation for the periodic task workload executed on a system with 3 cores. In Figure 4.7 (top), the initial relative large values of the slack of three cores

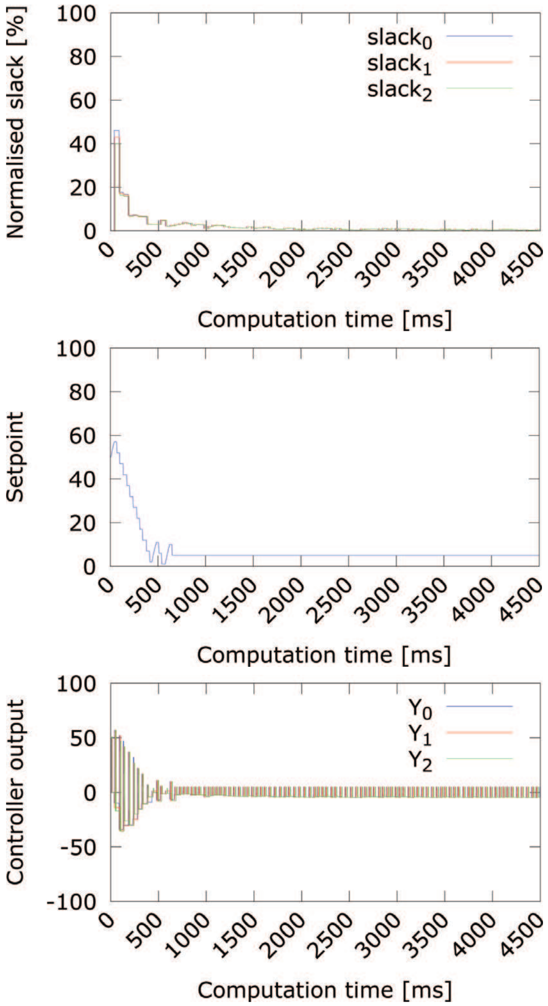


Figure 4.7 Dynamic slack (*top*), setpoint (*centre*) and controller output (*bottom*) during the simulation for the periodic task workload simulation scenario executed by a 3 core system.

(plotted with three different colors), normalised to task deadlines, equal to 46%, 42.8%, and 39.9% (the difference is caused with the random ET). The slack values decrease fast and after 530 ms none of them is higher than 5% of the task deadlines. This implies that tasks are executed relatively close to their deadlines, but never miss them. This behavior is obtained due to the exact schedulability test performed in the Design Space Exploration block and also indicates minimization of the steady-state errors by controllers. Taking into account the initial high values of the setpoint, the time of reaching the low normalised slack time can be treated as rather short. However, in random and industrial workloads reaching any steady state is very rare due to the lack of strict periodic behaviour of the workloads, as shown later in this chapter.

The early estimation based on controller outputs does not admit too many unschedulable tasks (in this experiment only 19 such tasks have been detected by the schedulability test). It is visible in Figure 4.7 (centre), where the value of setpoint decreases from the initial value to the minimum (by the functionality of the 2nd part of the algorithm presented in Algorithm 3.9), and after 650 ms no increase is observed. It means that after this time not a single unschedulable task has been wrongly identified as schedulable by the early estimation. The initial high values of normalised slack and setpoint are also reflected in Controllers' output values (Figure 4.7 (bottom)). Every time the value of an appropriate controller output is negative, a released task cannot be executed on the corresponding processing core. Despite only a sign of the controller output is important for the task admittance, relative large values of the controller outputs denote significant variance over observed normalised slack, which may be caused with not yet stabilised value of the controller setpoint. After about 750 ms the absolute value of the controller outputs are rather low, which means that the task slacks observed in the corresponding cores are low and the workload is rather predictable as compared to the random workload (next experiment).

4.3.2.2 Light workload

In Figure 4.8, the observed run-time metrics of the closed-loop 3-core system simulation of one selected light workload (with $Param = 7.89$), taken from [23], is presented. From this particular workload, 53 tasks have been executed, 569 tasks rejected by the early estimation, and 293 tasks rejected by the exact schedulability test. In comparison with the periodic workload run-time characteristics, presented in Figure 4.7, more false positive early estimations can be observed, which is reflected in higher values in the curve depicting the setpoint value (Figure 4.8 (centre)). Since the execution time of the tasks assigned to the cores vary significantly (from 1 ms to 95 ms), the normalised slack times and consequently controller outputs differ considerably for each system core (Figure 4.8 (top, bottom)), but overall decrease trend in the normalised slack time can be noticed.

4.3.3 Core Utilization

For the periodic workload, Figure 4.9 (top) presents the total utilisation of the three cores (100% core utilisation means that all cores are busy at a particular instant). Except for the system initialisation, there is no situation that all three

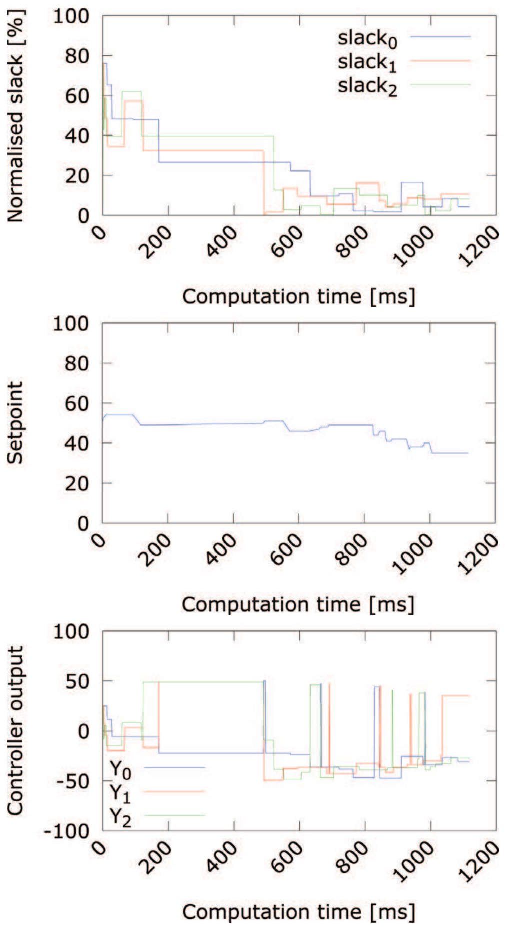


Figure 4.8 Dynamic slack (*top*), setpoint (*centre*) and controller output (*bottom*) during the simulation for the selected light workload simulation scenario executed by a 3 core system.

cores are idle. On average, the core utilisation for this simulation is equal to 83%. All three cores are balanced as the difference in their utilisations does not exceed more than 2 per mile. Similar utilisation and balance have been observed for other system configurations.

For the light workload, used also as an example in the previous subsection, a relatively long idle period of all cores can be observed (Figure 4.9 (bottom)). It is caused with the lack of task release between 10 ms and 490 ms in this particular workload. Except for this interval, it is rather difficult to observe any controller steady state, which is due to the changeable nature of the workload.

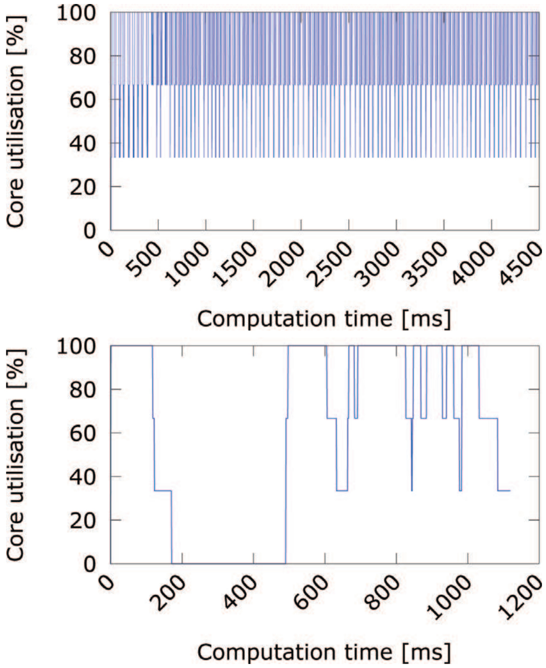


Figure 4.9 Core utilisation during the simulation for the periodic (*top*) and light (*bottom*) workload simulation scenario with 3 core system.

4.3.4 Case Study: Industrial Workload Having Dependent Jobs

To analyse industrial workloads, 90 workloads have been generated based on the grid workload of an engineering design department of a large aircraft manufacturer, as described in [23]. These workloads include 100 tasks of 827 to 962 jobs in total. The job execution time varies from 1ms to 99 ms. Since the original workloads have no deadlines provided explicitly, relative deadline of each task has been set to its WCET increased by a certain constant (100 ms).

In these workloads all jobs of any task are submitted at the same time, thus it is possible at the first stage to identify the critical path of each task and admit the task if there exists a core that is capable of executing the jobs belonging to the critical path before their deadlines. At the second stage, the remaining jobs of the task can be assigned to other cores so that the deadline of the critical path is not violated. The outputs from controllers can be used for choosing the core for the critical path jobs (during the first stage) or the cores for the remaining jobs (during the second stage). Four configurations, summarised in Table 4.2, can be then applied.

Table 4.2 Four configuration possibilities with respect to controllers' output usage (OL and CL stands for open-loop and closed-loop, respectively)

Core Selection for Critical Path Tasks	Core Selection for Tasks Outside the Critical Path	Configuration Abbreviation
without controllers	without controllers	OLOL (baseline)
without controllers	with controllers	OLCL
with controllers	without controllers	CLOL
with controllers	with controllers	CLCL

Figure 4.10 (*top*) shows the number of jobs executed before their deadlines. The OLOL configuration can be treated as the baseline, since no control theory elements have been applied (only exact schedulability tests are used to select a core for a job execution). The cores are scanned in a lexicographical

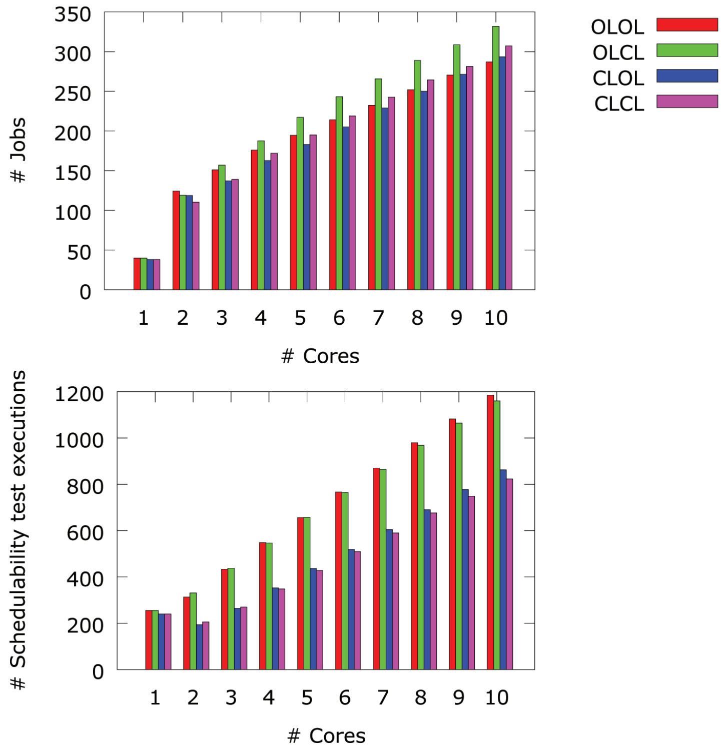


Figure 4.10 Number of executed jobs (*top*) and number of schedulability test executions (*bottom*) for systems configured in four different ways for the industrial workloads simulation scenario.

order as long as the first one capable of executing the job satisfying its timing constraints is not found, whereas in the closed-loop configurations the tasks are checked with regards to the decreasing value of the corresponding controller outputs.

The OLOL configuration approach seems to be particularly beneficial in the systems with lower number of cores (heavier loaded with tasks). However, in the systems with more than two cores, the OLCL configuration leads to the best results. Its superiority in comparison with CLCL stems from the fact that an over-pessimistic rejection of critical path jobs leads to fast rejection of the whole task. Thus the cost of a false negative estimation is rather high. Wrong estimation at the second stage usually results in choosing an idler core. The OLCL configuration admits 11% more jobs than OLOL, whereas CLCL is only slightly (about 1.5%) better than the baseline OLOL.

The main reason for introducing the control-theory based admittance is, however, decreasing the number of costly exact schedulability testing. The number of the exact test executions is presented in Figure 4.10 (bottom). Not surprisingly, the wider the usage of controller outputs, the lower is the cost of schedulability testing. The difference between OLOL and OLCL is almost unnoticeable, but the configurations with control-theory-aided selection of a core for the critical path jobs leads to significant, over 30% reduction.

From the results it follows that two configurations OLCL and CLCL dominate the others: the former in terms of number of executed jobs, the latter in terms of number of schedulability tests. Depending upon which goal is more important, one of them is advised to be selected. Interestingly, only in case of low number of processing cores, the baseline OLOL approach is slightly better than the remaining ones. For larger systems, applying PID controllers for task admissions seems to be quite beneficial.

4.4 Related Work

A majority of works that apply techniques originated from control-theory to map tasks to cores offers soft real-time guarantees only, which cannot be applied to time-critical systems [82]. Relatively little work is related to the hard real-time systems, where the task dispatching should ensure admission control and guaranteed resource provisions, i.e., start a task's job, only when the system can allocate a necessary resource budget to meet its timing requirements and guarantee that no access of a job being executed to its allocated resources is denied or blocked by any other jobs [95]. Providing such kind of guarantee facilitates to fulfill the requirements of time critical systems, e.g., avionic and automotive systems, where timing constraints must be satisfied [56, 75].

Usually hard real-time scheduling requires a priori knowledge of the worst-case execution time (WCET) of each task to guarantee the schedulability of the whole system [42]. However, according to a number of experimental results [48], the difference between WCET and observed execution time (ET) can be rather substantial. Consequently, underutilization of resources can often be observed during hard real-time system run-time. The emerging dynamic slack can be used for various purposes, including energy conservation by means of dynamic voltage and frequency scaling (DVFS) or switching off the unused cores with clock or power gating and slack reclamation protocol [140].

In [162], the authors claim that numerous existing hard real-time schemes are not capable of adapting to dynamically changing workloads in a satisfactory manner and thus do not scale well in the average case, whereas substantial energy dissipation savings are possible. An idea of splitting each task's WCET into two intervals is presented, where the length of the first interval is equal to the predicted execution time and the remaining part is the second interval. The entire dynamic slack, accumulated from previously executed tasks, is meant to be consumed during the first interval, by executing the task with lower voltage and frequency and, consequently, lower performance. The length of this interval is determined with a proportional-integral-derivative (PID) controller. Similar approaches have been applied in [3] and [140].

In [45], a response time analysis (RTA) has been used to check the schedulability of real-time tasksets. This ensures meeting all hard deadlines despite assigning various execution frequencies to all real-time tasks to minimise energy consumption. In the approach proposed in this chapter, RTA is also performed, but it is executed far less frequent due to the fast schedulability estimation based on controllers and thus is characterised with shorter total execution time.

Some researchers highlight the role of a real-time manager (RTM) in scheduling hard real-time systems. In [74], an RTM is used together with computing resources monitoring, while schedule information are precomputed from an SDF graph statically to help guaranteeing the real-time constraints. We have extended basic ideas from their scheme to work with dynamic workloads using information gathered by the monitor. The role of RTM is also highlighted in [72]. They described that after receiving a new allocation request, it checks the resource availability using a simple predictor. Then the manager periodically monitors the progress of all running tasks and allocates more resources to the tasks with endangered deadlines. However, it is rather difficult to guarantee hard real-time requirements when no proper schedulability test is applied. In [131], an RTM exploits information about the probability of task execution time to predict the slack available for power management. It is assumed that the execution time of a task in terms of its worst-case execution time is given by a known cumulative distribution function. The stochastic nature of this approach prevents it from application in hard real-time systems if even tiny probability (e.g., 10^{-12} [16]) of missing any deadline is not allowed.

From the literature survey it follows that applying feedback-based controllers in hard real-time systems has been limited to determining the appropriate frequency benefiting from DVFS. According to the authors' knowledge, the feedback controller has not been yet used by an RTM to perform hard real-time task allocation under dynamic workload on many-core systems.

4.5 Summary

In this chapter we presented a novel scheme for dynamic workload task allocation to many-cores using a control-theory-based approach. Unlike the majority of similar existing approaches, we deal with workloads having hard real-time constraints that is desired in time critical systems. Thus, we are forced to perform exact schedulability tests, whereas PID controllers are used for early estimation of schedulability.

We achieved an improved performance due to reduced number of costly scheduling test executions, slightly limiting the number of admitted tasks in the majority of cases. The controllers observe dynamic slack of executed tasks and aim to select the core with the lowest load.

For heavy workloads the proposed scheme achieves a better performance than using the typical open-loop approach. Up to 65% lower number of schedulability tests are to be performed, whereas the number of admitted tasks is almost equal for the heavy-loaded system and lower up to 19% with lightweight scenarios, for which the proposed scheme is less appropriate. For industrial workloads with dependent jobs executed on larger systems, the number of executed tasks using the proposed approach was even higher than the open-loop baseline system due to the selection of more idle cores for computing jobs belonging to the critical path.

Since schedulability analysis requires relatively long computation time, decreasing the number of its executions should lead to considerable computation time and energy reduction. The exact gain depends on a particular system configuration and will be evaluated in our future work. We also plan to consider heterogeneous many-core system and extend the proposed approach for mixed criticality workloads.

5

Search-Based Heuristics for Modal Application

Due to the growing number of electronic control units (ECUs) in contemporary cars, sometimes reaching even 100, the automotive industry gradually resigns from their paradigm of using a separate unit for each functionality [99]. The requirement of placing a number of ever more sophisticated functionalities in one chip resulted in appearance of multi-core ECUs [158]. The AUTOSAR (AUTomotive Open System ARchitecture) standard [1] assumes a static (i.e., compile-time) mapping of atomic software components, named *runnables*, into cores since it is less complex and more predictable than dynamic resource allocation [94].

Due to the hard real-time constraint in automotive systems, the cores have to execute all the tasks on time even for their worst-case execution behavior, where they take worst-case execution time (WCET), which is usually much higher than the average execution time [152]. One possibility of decreasing the difference between the worst and average task execution times stems from the modal nature of such applications, i.e., from the fact that they can behave in a limited, known at design-time, number of ways, named *modes*. If each mode is analysed independently, the average execution time may be closer to the WCET determined for that mode [118]. In [104], six modes have been identified in a 4 cylinder gasoline torque based system, for example Cranking, Idle and Wide Open Throttle. It has been stressed there that execution times of particular runnables differ significantly for various modes of an ECU and thus applying different mappings for each operating mode may be beneficial. This way a lower number of cores could be needed than that of the corresponding system design not considering operating modes. However, introducing different mapping for modes imposes significant design complications, which have not been analysed in [104].

The contexts of runnables that are executed on different cores in different modes have to be migrated from one core to another, setting additional requirements for the available communication bandwidth. The process of mode switching usually incurs overhead (both in execution time and energy), which is to be taken into account at run-time to decide whether to switch to a different mode or not. In hard real-time systems, it is essential to satisfy all the timing constraints even during the mode switching process, i.e., the migration time of tasks must be time bounded [146]. Therefore, the worst case switching time has to be assumed to provide the timing guarantees.

During the migration process, the taskset schedulability must not be violated. To guarantee this property, we propose to treat a migration process as any other asynchronous process in schedulability analysis, i.e., to use so-called *periodic servers*, which are periodic tasks executing aperiodic jobs. When a periodic server is executed, it processes pending task migration. If there is no pending migration, the server simply holds its capacity. To reduce the migration time, a recursive greedy algorithm for reducing the amount of data transferred during a mode change is proposed. It aims to decrease the number of periodic server instances used during a single mode switching. The proposed approach can be applied to any hard real-time systems, where different operating modes can be identified, and automotive systems in particular.

As an example, throughout this chapter we will analyse an engine ECU code named *DemoCar*. We will identify its operating modes and apply clustering to decrease their number and to eliminate task migration between neighbouring mode pairs (i.e., two modes from which at least one mode can be directly transferred to the second one) if the mode change is to be finished rapidly. The mappings for each mode will be determined using a genetic algorithm. This algorithm applies two optimizing criteria: runnable schedulability in terms of a number of deadline violations and migration cost in terms of the context length of the transmitted runnables. The typical schedulability analysis is used to determine the necessary network bandwidth to guarantee that the mode switching migration finishes in the required time.

In the next section, the state-of-the-art solutions are described followed by the proposed approach and a discussion on providing performance guarantees during mode changes.

5.1 System Model and Problem Formulation

5.1.1 Application Model

In this work we assume application model is consistent with the AUTOSAR standard [1]. A taskset Γ is comprised of an arbitrary number of periodic runnables, $\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots\}$, grouped in tasks with hard real-time constraints. The j-th occurrence (j-th job) of runnable τ_i is denoted with $\tau_{i,j}$. The taskset is known in advance, including the WCET of each runnable, C_i, its period T_i, priority P_i and its relative deadline D_i equal to this period. Runnables are atomic schedulable units communicating each other with so called *labels*,

$N = \{\nu_1, \ldots, \nu_r\}$, which are memory locations of a particular length. The order of read and write operations to labels denotes the runnable dependencies, as the write operation to a particular label should be completed before its reading. Deadlines for mode changing time between each neighbouring pair of modes are also provided. We assume that the labels are stored in the same node that the runnable that reads these labels. If more than one runnable mapped to different cores read from the same label, its content is to be replicated to all the reading nodes and the writer should update the label value at all the locations. It means that the writer is aware of all its readers and knows their locations in all the possible modes.

Example 1 *Throughout this chapter, we consider a lightweight engine control system named DemoCar as an example application. The flow graph of this application is depicted in Figure 5.1. It consists of 18 runnables and 61 labels. All runnables are periodic: 8 runnables (highlighted in green) are to be executed every 10 ms, whereas period of 6 runnables (red, blue and yellow) equals 5 ms, two (violet) runnables are executed every 20 ms and the period of two (orange) runnables is 100 ms. In Figure 5.2 (upper part), 11 identified modes of this application are presented. These modes have been identified by inspecting the code of the runnable named* OperatingModeSWC*, which computes values of transaction and output functions of the Finite State Machine steering this engine. For example, label* FuelEnabled *is read by two runnables:* TransFuelMassSWC *and* ThrottleChangeSWC*. If these runnables are mapped to different cores, the label is to be replicated and kept in both the cores where these runnables were mapped to. It is a role of the writer,* OperatingModeSWC*, to update these values coherently not violating any timing constraints.*

5.1.2 Platform Model

The hardware platform assumed in this chapter is a mesh Network on Chip (NoC) with a certain number of cores $\pi \in \Pi$ and routers $\psi \in \Psi$, as shown in Figure 5.3. Each link is modelled as a single resource, so, for example, to transfer a portion of data from $\pi_{0,1}$ to appropriate sink $\pi_{2,0}$ we need such resources allocated simultaneously: $\pi_{0,1} - \psi_{0,1}$, $\psi_{0,1} - \psi_{1,1}$, $\psi_{1,1} - \psi_{2,1}$, $\psi_{2,1} - \psi_{2,0}$, $\psi_{2,0} - \pi_{2,0}$. In every mode, each runnable is mapped to one core and a label is stored in local memories of the cores requesting that label. Data transfer overhead is taken into consideration, assuming constant time for transferring a single flit (Flow control digIT, a piece of a network package whose length usually equals the data width of a single link) between two neighbouring cores if no contentions are present. Timing constants for packet

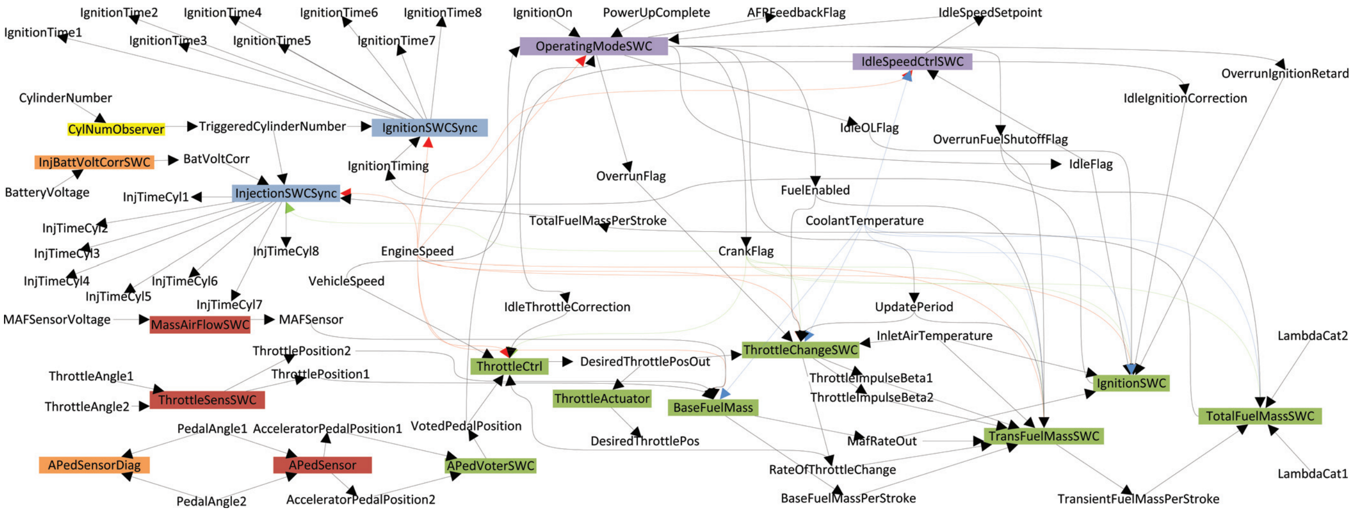


Figure 5.1 Flow graph of the DemoCar example; the runnables belonging to the same task are highlighted with the same colour, labels are not highlighted. Some flows are drawn in different colours for readability.

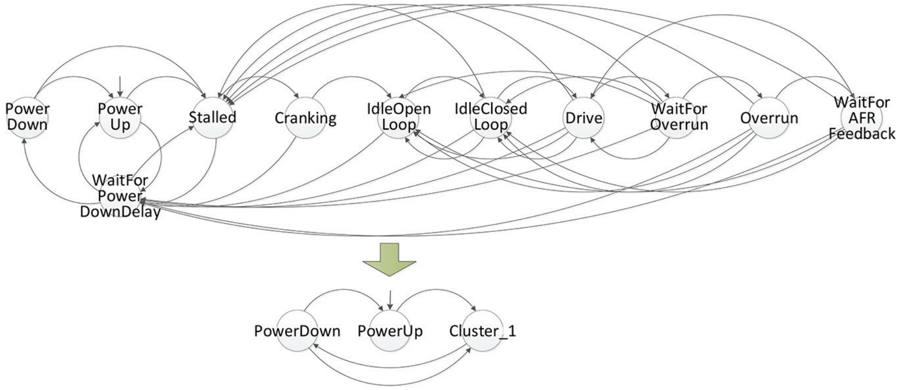


Figure 5.2 Finite State Machine describing mode changes in DemoCar use case: before (*upper part*) and after (*lower part*) the clustering step.

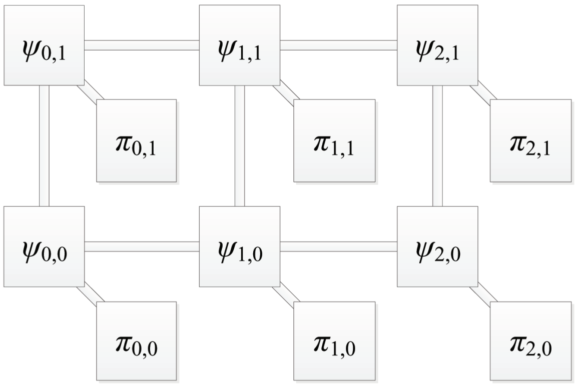


Figure 5.3 An example many-core system platform.

latencies while traversing one router and one link are denoted as d_R and d_L, respectively. The priority of data transfer packets are assumed to be equal to the priority of the runnable sending them.

5.1.3 Problem Formulation

Given a platform and an application model with a defined set of operating modes, the problem is to determine schedulable mappings for each mode so that the amount of data to be migrated during the allowed mode changes is minimized. During mode changing, the taskset should be still schedulable despite the additional network traffic generated by the task migrations. The neighbouring modes (i.e., the modes connected with a link in the FSM describing the allowed mode transitions) with similar runnables' execution time can be clustered to decrease the frequency of task migrations. The deadlines for mode changing time between each neighbouring pair of modes must not be violated.

5.2 Proposed Approach

In this section, steps of the proposed design flow are described. Since it has been assumed that the tasksets of the considered application are known in advance, it is possible to perform some preliminary computations statically. Consequently, the mapping problem can be split into two stages: off-line (static) and on-line (dynamic), as shown in Figure 5.4. The computation time of the off-line part is not crucial and thus heuristics with even high complexity may be used for runnable and label mappings. It seems promising to combine the most effective approaches, such as multi-objective simulating annealing or genetic algorithms. The possibility of extending genetic operators benefiting from the full knowledge of the system domain, such as mutation in a way similar to [109], makes the genetic approach the first choice at this step.

During run time, detection of the current mode is assumed to be done by observing certain variable. (In DemoCar such variable is named *–sm* and is stored in runnable *OperatingModeSWC*.) When a value of this variable has been changed, the current runnable and label mapping might have to be switched. The mappings have been chosen during the design time with respect to minimize the amount of data to be migrated. Schedulability analysis guarantees that even the worst case switching time does not violate the deadline required for mode changes. If such violation is unavoidable, either the states can be clustered, or the network bandwidth is to be increased.

The off-line part of the proposed approach is comprised of five steps, which are covered in the following subsections.

5.2.1 Mode Detection/Clustering

The reasons for introducing the *mode detection & clustering* step are twofold. Firstly, some neighbouring modes can be characterized with similar runtime

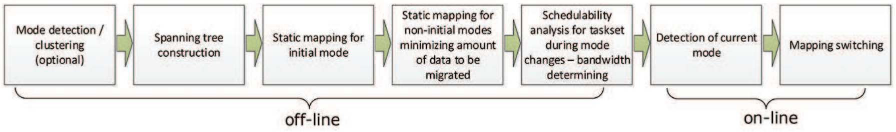

Mode detection / clustering (optional)
Spanning tree construction
Static mapping for initial mode
Static mapping for non-initial modes minimizing amount of data to be migrated
Schedulability analysis for taskset during mode changes – bandwidth determining
Detection of current mode
Mapping switching
off-line
on-line

Figure 5.4 Steps of dynamic resource allocation method benefiting from modal nature of applications.

and resource consumption. Then there is little benefit in preparing different mappings for such modes and migrating the runnables when a transition between these neighbouring modes is made. Moreover, some transitions are required to be done immediately, whereas others can be less time tight. If a runnable migration is to be performed quickly, for example between two consecutive runnable occurrences, the bandwidth needed to transfer the appropriate amount of data in that time may be unreasonably high. Therefore, it may be more sensible to merge two modes with such rapid task switching time and generate only one mapping for them.

Example 2 *(continuation of Example 1) In DemoCar, transitions between modes:* Stalled*,* Cranking*,* IdleOpenLoop*,* IdleClosedLoop*,* Drive*,* WaitForOverrun*,* Overrun*,* WaitForAFRFeedback *and* WaitForPowerDownDelay *are to be performed between two consecutive executions of their runnable occurrences, which is upperbounded with 5 ms for 9 runnables. Since performing task migration during such short time window would require a bandwidth of considerable size, these modes have been clustered into* Cluster_1*. Finally, three modes can be identified after the clustering step:* PowerDown*,* PowerUp *and* Cluster_1*, as presented in Figure 5.2 (lower part).*

5.2.2 Spanning Tree Construction

To minimize the amount of data to be migrated between two consecutive modes with the technique proposed in this chapter, the FSM describing mode changes should include weights denoting state transition probabilities. Since probabilities of staying in the current mode are not relevant at this step, they can be omitted for simplicity. The probabilities can be given or determined during long simulation of the modal system. The FSM has also to have all its cycles removed to guarantee halting of the *Static mapping for non-initial modes* step. In this regard, for an FSM treated as a weighted connected graph $G(V, E)$, where V is the set of vertices and E denotes the set of edges, a maximum spanning tree can be constructed. We recollect that a spanning tree of a graph G is its subgraph $T(V, E')$, which is connected and whose number of edges is equal to the number of vertices minus 1, $|E'| = |V|-1$. If $\mathcal{T}$ denotes the set of all spanning trees of G, a maximum spanning tree $T_{max}(V, E_{max})$ of G is a spanning tree iff:

$$\underset{T(V,E')\in\mathcal{T}}{\forall} \sum_{(v,z)\in E_{max}} w(v,z) \geq \sum_{(v,z)\in E'} w(v,z),$$

where $w(v, z)$ is the weight value assigned to the edge from a vertex v to z. A maximum spanning tree can be constructed in time $O(|E|\log|V|)$, e.g., by the classic Prim's algorithm [108].

The operation performed in this step neither influences the application behaviour nor limit the possible mode transitions. It only makes the least frequent transitions not optimized during stage *Static mapping for not initial modes minimizing amount of data to be migrated* (Figure 5.4).

Example 3 *(continuation of Example 2) For DemoCar, probabilities of mode changing have been shown in Figure 5.5 (left). The maximum spanning tree, constructed with the Prim's algorithm, is presented in Figure 5.5 (right).*

5.2.3 Static Mapping for Initial Mode

Since mapping for each mode is performed off-line, even heuristics known from their high computational cost, such as genetic algorithms, can be applied. A genetic algorithm used for hard real-time systems shall guarantee that under the chosen schedule all timing constraints are satisfied. This can be performed in several ways. For example, each missed deadline can impose a certain penalty to the fitness function value, and thus each schedule with unsatisfied constraints should be eliminated during the evolutionary process. A particular mapping is portrayed as a chromosome, stored as a bit string, representing on each gene the processing core where the task would be mapped to, similarly to [61]. The bit string one-point crossover operator and flip bit mutation have been applied together with the tournament selection of the individuals.

Below, an algorithm encompassing the aforementioned properties is described.

In Algorithm 5.1, it is presented a pseudo-code of a genetic algorithm that can be used during *Static mapping for initial mode* step, the third off-line step of the proposed approach, as depicted schematically in Figure 5.4. We propose to use two fitness functions – measuring (i) the number of deadline violations and (ii) makespan (also known as response time). Both these functions apply the interval algebra described in Chapter 2. The first fitness function value is of primary importance, as in a hard real-time system no deadline violation is allowed. But among fully schedulable mappings, the one leading to a

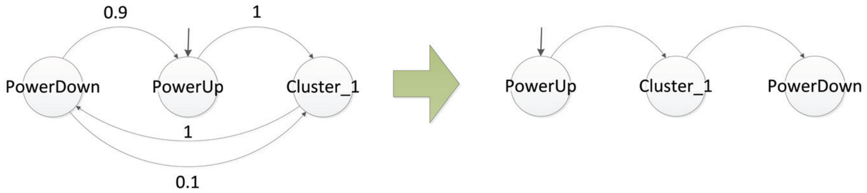


Figure 5.5 Spanning tree construction for DemoCar.

Algorithm 5.1 Pseudo-code of no deadline violation with makespan minimisation algorithm for the initial mode mapping

```
inputs   : Workload Γ;
           Resource set Π;
outputs  : Task mapping;

1  Choose an initial random population of task mappings
2  while not termination condition do
3      Evaluate the number of deadline violations using IA; //criterion (i)
4      Evaluate the makespan using IA; //criterion (ii)
5      Create clusters of individuals with the same number of deadline violations;
6      Sort the clusters by increasing number of deadline violations;
7      Sort individuals in each cluster wrt their makespan;
8      Perform tournament selection; //criterion (i) has higher priority than criterion
         (ii)
9      Generate individuals using crossover and mutation;
10     Create a new population with the best found mappings;
11 end
```

lower makespan is chosen, since idle intervals can be used to decrease energy consumption or execute tasks of lower criticality levels.

In the algorithm, the following steps can be singled out.

Step 1. Initial population initialisation (line 1). An arbitrary number of random task mappings (individuals) is created.

Step 2. Creating a new population (lines 3–10). For each individual, values of two fitness functions are computed - the number of deadline violations and the makespan (lines 3–4). Individuals with the same number of deadline misses are grouped together (line 5). The groups are then sorted with respect to the number of deadline violations in the ascending order (line 6). Inside each group, individuals are sorted according to their growing makespan (line 7). The tournament selection is then performed – individuals from a group with lower number of deadline violations are always preferred, whereas among individuals from one group the one with the lowest makespan is to be chosen (line 8). The individuals winning the tournament are then combined using a typical crossover operation and mutated (line 9). A new population is created (line 10). Step 2 is repeated in a loop as long as a termination condition is not fulfilled, which can be a maximal number of generated populations or lack of improvement in a number of subsequent generations.

Example 4 *(continuation of Example 3) For the* PowerUp *(initial) mode of DemoCar to be executed on a multi-core embedded system, we evaluate makespan and number of violated deadlines during one hyperperiod (i.e.,* *the least common multiple of all runnables' periods) by allocating runnables and labels to different cores.*

The size of the NoC mesh has been initially configured as 2x2 with no idle cores, since this size has been earlier checked (also using Algorithm 5.1) and is large enough to execute DemoCar in the most computational intensive mode, Cluster_1*, not violating any of its timing constraints. The genetic algorithm is executed again to perform assignment of tasks to cores with timing characteristics for the initial* PowerUp *mode. The genetic algorithm has been configured to generate 100 generations of 20 individuals each. The first fully schedulable allocation has been found in the 1st generation, which suggests that it might be possible to allocate the taskset to a lower number of cores.*

After performing further search it has appeared that the taskset in the initial mode is schedulable even when mapped to one (out of four) active core. The lowest makespan for the NoC with three idle cores is equal to 8622 μ*s.*

5.2.4 Static Mapping for Non-Initial Modes

It is of primary importance to migrate as little data as possible during mode changes to minimise the migration time.

Each application A includes a set of tasks and can be represented with a vector comprised of p runnables and r shared memory locations (labels) of these tasks, A $= [\tau_1, \ldots, \tau_p, \nu_1, \ldots, \nu_r]$, and platform Π is composed of s processing cores, $\Pi = \{\pi_1, \ldots, \pi_s\}$. A mapping M is a vector of p core locations, M $= [\pi_{\tau_1}, \ldots, \pi_{\tau_p}]$, where each element corresponds with the appropriate element of A and can be substituted with any element of set Π. Each element of weight vector W, W $= [w_{\tau_1}, \ldots, w_{\tau_p}]$, is equal to the amount of data that has to be transferred when a particular runnable is migrated, including the labels to be read.

Let M_α and M_β be sets of mappings that are fully schedulable in a given system in state α and β, respectively. The elements of the difference vector $D_{m_\alpha, m_\beta} = [d_{\tau_1}, \ldots, d_{\tau_p}]$ indicate which runnables are to be migrated when the mode is changed from α to β. Each element d_δ, $\delta \in \{\tau_1, \ldots, \tau_p\}$, takes value 1 if the particular runnable/label is allocated to different cores in mappings $m_\alpha \in M_\alpha$ and $m_\beta \in M_\beta$, and 0 otherwise:

$$d_\delta = \begin{cases} 0, & \text{if } m_{\alpha,\delta} = m_{\beta,\delta}, \\ 1, & \text{otherwise} \end{cases} \tag{5.1}$$

where $m_{\alpha,\delta}$ and $m_{\beta,\delta}$ denote the δ-th element of vectors m_α and m_β, respectively. The migration cost c between two states α and β is then computed in the following way:

$$c_{m_\alpha,m_\beta} = \mathrm{D}_{m_\alpha,m_\beta} \cdot \mathrm{W}^{\mathrm{T}}. \tag{5.2}$$

A recursive greedy algorithm for reducing an amount of data transferred during mode changes is presented in Algorithm 5.2.

Since some cycles are likely to occur in a graph representing the Finite State Machine describing transitions between modes, a spanning tree (ST) is to be built, as described in the previous subsection. Then the mode corresponding to the initial state of the FSM is selected as the current mode (line 1). For this mode, a set of schedulable mappings is generated, e.g., with Algorithm 5.2 (line 2). If more than one schedulable mapping is found, an additional criterion $crit$ (e.g., minimum makespan value) is used to select one of them (line 3). Then for each direct successor of the ST node corresponding to FSM initial state, the *FindMappingMin* procedure is executed (lines 4 and 5).

In the *FindMappingMin* procedure, a set of schedulable mappings for that successor node is found using minimal migration cost criterion (5.2) (line 8). If more than one schedulable mapping is equally evaluated by this criterion, an additional criterion, $crit$, is used (line 9). The *FindMappingMin* procedure is then recursively run for each direct successor of the ST node provided as the function parameter (lines 10 and 11). More mappings could be delivered to the *FindMappingMin* procedure to browse a larger search space by skipping lines 4 and 9 in the algorithm and providing all elements of M_α instead of just one.

Algorithm 5.2 Pseudo-code of a migration data transfer minimisation algorithm

```
inputs  : A spanning tree ST based on Finite State Machine (FSM) describing the
          system modes with transaction probabilities;
          W - size of each runnable memory footprint;
          $crit$ - mapping optimality criterion (e.g., minimum makespan value);
outputs : Runnable and label mapping for each mode;
1  Select the initial state of ST and assign it to $\alpha$;
2  Find a set of schedulable mappings $\mathrm{M}_\alpha$;
3  Select $m_\alpha \in \mathrm{M}_\alpha$ wrt criterion $crit$;
4  forall $\beta$ being a direct successor of $\alpha$ in ST do
5      FindMappingMin($\alpha$, $\beta$, $m_\alpha$);
6  end
7  FindMappingMin($\alpha$, $\beta$, $m_\alpha$)
8  Find a set of schedulable mappings $\mathrm{M}_\beta$ minimizing criterion (5.2) using W
9  Select $m_\beta \in M_\beta$ wrt criterion $crit$
10 forall $q$ being a direct successor of $\beta$ in ST do
11     FindMappingMin($\beta$, $q$, $m_\beta$)
12 end
```

Example 5 *(continuation of Example 4) Regardless of the mode, the application has been mapped in a 2 × 2 mesh Network on Chip without deadline violations. For the* PowerUp *mode, schedulable mappings have been found even if three of the four NoC cores remains idle, as shown in Example 4. It means that in this mode three cores can be switched off, leading to considerable energy savings. Similarly, two cores can remain idle in the* PowerDown *mode.* (PowerDown *requires more computations than* PowerUp *since some maintenance procedures are to be consistently performed.*) *However, despite intensive search using a genetic algorithm, all four cores are needed in the* Cluster_1 *mode to have the taskset fully schedulable. Thus, when the current mode changes from* PowerUp *to* Cluster_1*, three cores have to be activated, whereas two cores can be switched off after leaving the* Cluster_1 *mode.*

Let us focus on the transition between the PowerUp *and* Cluster_1 *modes. For* PowerUp*, only one core is active and thus all runnables are to be mapped to the only active core. However, in other cases a larger set of mappings that are fully schedulable on active NoC cores would have been identified. From these mappings, the one with the lowest makespan (an additional criterion) is chosen. This mapping has been used as a parameter of the* FindMappingMin *procedure (from the algorithm presented in Algorithm 5.2). The set of schedulable mappings following the minimum criterion (Equation (5.2)) is identified using a genetic algorithm. By the applied criterion ($min(c_{m_\alpha,m_\beta})$), a significantly lower amount of data has to be migrated during the mode change. In the best found case, 3 runnables have to be migrated, whose total* $c_{PowerUp,Cluster\_1} = 261968$ *bytes. However, by not using this criterion, but the minimal makespan instead, the lowest number of runnables to be migrated equals 13, which results in* $c_{PowerUp_2,Cluster\_1} = 890162$ *bytes. In the second case, the amount of data to be transferred using a periodic server is about 240% higher than in the first mapping pair. Since periodic servers offer equal throughput during the system execution, the mode change between the latter mappings would last more than three times as long as between the former pair.*

During mode change from Cluster_1 *to* PowerDown*, 2 runnables have to be migrated and* $c_{Cluster\_1,PowerDown} = 113568$ *bytes. Although the transition between modes* PowerDown *and* PowerUp *are not optimized, in this case only 2 runnables have to be migrated with* $c_{PowerDown,PowerUp} = 113536$ *bytes.*

5.2.5 Schedulability Analysis for Taskset During Mode Changes

Since some runnables and labels are expected to be located at different cores in two different modes, their migration is to be performed during mode changes. A runnable migration process is schematically depicted in Figure 5.6. Two mappings m_α and m_β of runnables τ_i, τ_j, τ_k into nodes π_a and π_b are

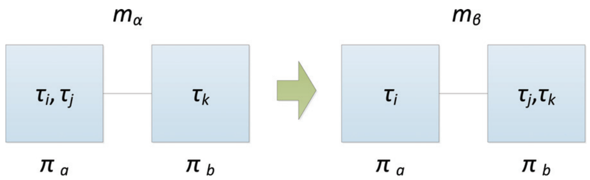


Figure 5.6 Example of two different mappings (m_α, m_β) of runnables τ_i, τ_j, τ_k into cores π_a and π_b.

used in two different system modes: α and β, respectively. The difference between these mappings is the assignment of runnable τ_j. We assume no deadline violations for both mappings m_α and m_β. During hyperperiods involved in the migration process between α and β, the schedulability analysis for communication resources should take into consideration not only all the transfers between π_a and π_b described in the workload, but also an additional periodic job, i.e., the periodic server of certain policy (polling, sporadic, deferrable, etc.) with a certain execution time in each period. A technique for determining this time is presented in this subsection.

When the mode changes from α to β, runnable τ_j is to be copied from π_a to π_b. Since the precopy strategy is applied, τ_j is still executed on core π_a during the migration. To migrate runnable τ_j, the periodic server is used. The whole context of the runnable is transferred during a number of subsequent hyperperiods. It is worth stressing that the maximal migration time can be computed statically, since the runnable context size and the periodic server time slot length and period are known. After this time, it is safe to start executing τ_j on π_b and remove its copy in π_a.

To guarantee schedulability of runnables, one of the schedulability tests, described for example in [42], shall be applied. It is possible to calculate the longest possible time interval between the release of runnable τ_i and its termination, which is referred as τ_i's worst case response time (WCRT) and is represented by R_i. The schedulability analysis is performed in the way described in [9], i.e., by checking whether WCRTs of all runnables do not exceed their deadlines. WCRT of runnable τ_i can be computed using equation:

$$R_i = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i + R_j - C_j}{T_j} \right\rceil C_j, \tag{5.3}$$

where $hp(\tau_i)$ denotes the set of all runnables that can preempt τ_i, C_i and C_j are the worst case execution time of τ_i and τ_j, respectively, and T_j denotes the period of τ_j. Similarly, the worst case latency r_k of packet φ_k transmitted over a link in a mesh NoC with wormhole switching, issued periodically every t_k,

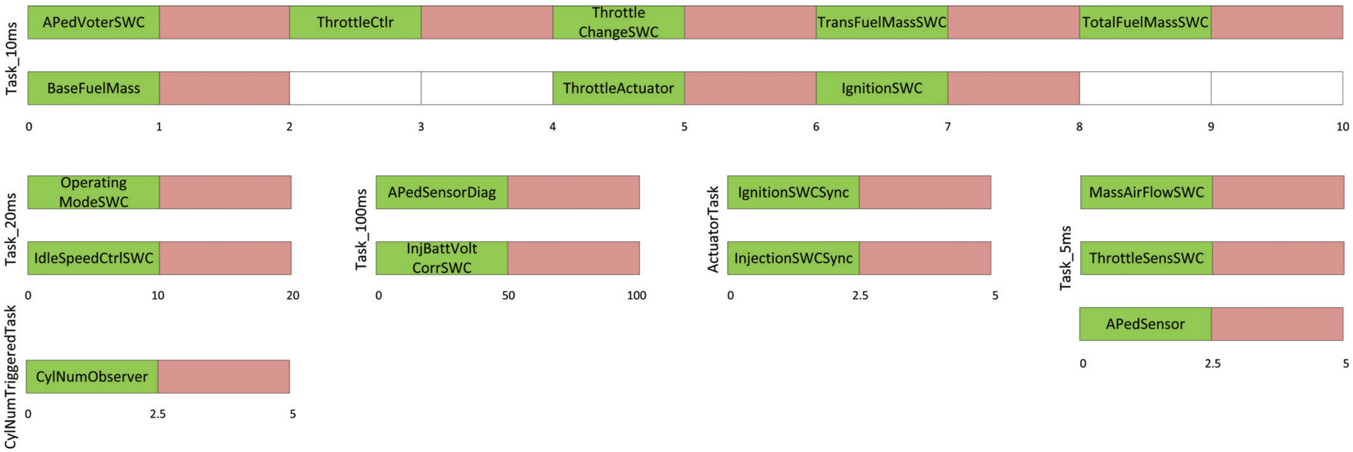


Figure 5.7 Tasks’ stages in DemoCar: green – runnable execution, red – write to labels; release times and deadlines are provided in ms.

can be formulated in a similar manner as that of [61, 100, 122]:

$$r_k = c_k + b_k + l_k, \tag{5.4}$$

where c_k is a basic network latency, b_k is the maximal blocking time from lower-priority packages, and l_k is the maximal blocking time due to interference with higher-priority packets. The basic network latency can be computed with the following equation [61, 100]:

$$c_k = H \cdot (d_R + d_L) + \left\lceil \frac{PS}{FS} \right\rceil \cdot d_L, \tag{5.5}$$

where d_R and d_L denote the constant packet latencies while traversing one router and one link, respectively, PS is the number of bits in the package, and FS is the flit length in bits. H is the hop number between source and destination cores. The remaining terms of Equation (5.4) can be computed with equations [61, 100]:

$$b_k = H \cdot (d_R + d_L), \tag{5.6}$$

$$l_k = \sum_{\forall \varphi_l \in interf(\varphi_k)} \left\lceil \frac{r_k + (r_l - c_l) + R_i}{t_l} \right\rceil (c_l + b_l), \tag{5.7}$$

where $interf(\varphi_k)$ denotes the direct interference set of φ_k, which is the set of all packets that can preempt φ_k, i.e., have a higher priority and share at least one link with φ_k. The response time of task τ_i that releases φ_k, R_i, has been substituted as a maximum release jitter. The term $(r_l - c_l)$ is an upper bound of indirect interference [61].

By applying Equations (5.3) and (5.4), both schedulabilities of independent runnables executed on processing cores and packet transmissions can be verified. However, jobs in the considered applications, possibly executed on different cores, are characterised with various dependency patterns. Typically, to start a job execution it is required to have all its parent jobs executed (which contributes to so-called *computation latency*) and all the necessary data transferred to the core where this job is assigned to (*communication latency*).

The goal is thus to establish whether all task-chains of an application have their end-to-end deadlines met in a particular platform, and this assessment is referred as *end-to-end schedulability test*. Such test must consider the end-to-end latency of each task of a *task-chain*. To check schedulability of a task chain, it is sufficient and necessary to test the individual end-to-end response times of all tasks belonging to that chain [73]. In [73], a technique for end-to-end schedulability analysis is proposed, but it assumes a pipelined task execution pattern, where multiple jobs of the same task chain are executed simultaneously over different cores, but the simultaneous execution of more than one job of the same task is not allowed. When the execution pattern does not follow this scheme, meeting end-to-end deadlines can be checked by assigning an appropriate local deadline for each job in every chain. These local deadlines shall be chosen in a way that all the jobs on every core are schedulable and the local deadlines at the chain last stage do not exceed the respective end-to-end deadline [59].

Example 6 *(continuation of Example 5) In DemoCar, each task is composed with series of three subsequent stages: read from labels, runnable execution, write to labels. Since the labels are always located in the same core as that of runnable reading these labels, the read stage can be omitted and two remaining stages are presented in Figure 5.7 for all tasks, highlighted in green and red. Runnables belonging to one task and drawn one above the other can be executed in parallel, whereas the execution order of the runnables follows dependencies defined by label write and read operations so that each label is to be written by a runnable prior to be read by another runnable. The end-to-end deadline has been divided into number of stages in each task and in that way deadlines for each stage have been determined* (*in Figure 5.7 these deadlines are written beneath the end point of each stage*)*.*

For example, the release time of runnable ThrottleCtrl *is 2ms. By this time, all the packets with label values required by this runnable are assumed to arrive at the node executing* ThrottleCtrl*. The deadline for this runnable execution is 3 ms, so the WCRT (R_i) must not be higher than this value. The packages with data are then issued between 2 ms (the runnable release time) and R_i. They have to reach their destination nodes earlier than 4 ms.*

To check schedulability of DemoCar, it is then sufficient to check schedulability for runnables executed in all (green) stages and also data transfers to and from labels performed in the appropriate (red) stages.

Since the earliest execution starting time of each runnable is limited by the starting time of the stage including particular runnable, this stage starting time can be treated as an offset O_i as described in [143]:

$$R_i = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i + (R_j - C_j - O_j) - O_i}{T_j} \right\rceil C_j + O_i. \tag{5.8}$$

Using similar rationale, Equations (5.4) and (5.7) can be rewritten in the following way:

$$r_k = c_k + b_k + l_k + R_i, \tag{5.9}$$

$$l_k = \sum_{\forall \varphi_l \in interf(\varphi_k)} \left\lceil \frac{(r_k - O_i) + (r_l - c_l - O_{i'})}{t_l} \right\rceil (c_l + b_l), \qquad (5.10)$$

where term $(r_k - O_i)$ reflects an additional jitter imposed by the response time of task R_i that initiates this transfer and $O_{i'}$ denotes an offset of the task that releases φ_l. One more requirement has to be added to set $interf(\varphi_k)$. It includes not only packets having a higher priority and transferred via a path sharing at least one link, but also timing boundaries of both their sender executions or traffic stages have to overlap.

As mentioned earlier, the proposed task mapping technique aims to benefit from a modal nature of applications, but it also possess new challenges. If the modes are treated independently from each other, the end-to-end schedulability of runnables and packet transmission in each mode can be analysed using Equations (5.8) and (5.9). It is the instant of transition between these modes that requires special attention. The task migration time can be computed with Equations (5.5), (5.6), (5.9), (5.10), where the packet size, PS, is equal to the sum of the header length and the size of the payload including the whole context of runnables and labels to be migrated. If a relatively large runnable is to be migrated in a highly utilised platform, performing the migration when the next job of the runnable is due to start could require rather high bandwidth in order not to violate any deadlines. Thus we assume to use the precopy strategy, as described in [109]. The job is executed in its current (source) location during the mode switching, until all the runnables have been migrated to their new (destination) locations. Then the migrated runnables are removed from the source location, and their next execution will be performed in their destination locations. If a runnable is of combinational nature (its outputs depend solely on input values; all DemoCar runnables have this property), only the runnable code section is to be migrated. In case of a sequential nature of a runnable, the whole context is to be migrated.

Similarly to [98], we split a runnable context intro two parts: invariant, which is not modified at runtime, and dynamic, including all volatile memory locations. We assume that an upper bound of the dynamic part size of all runnables is known in advance. This part shall be migrated at once using the last instance of the periodic server. It means that the local memory locations that can be modified by the runnable must not be precopied, but migrated after the last execution of the runnable in the old location. This requirement can influence the minimum periodic server size and, consequently, the network bandwidth, as it must be then wide enough to guarantee migration of dynamic part before the next runnable execution (in the new location). This property shall be checked using (5.9).

In the proposed approach, any kind of periodic servers can be used, however, the trade-off between implementation complexity and ability to guarantee the deadlines of hard real-time tasks, as described for example in [40], shall be considered.

The number of the hyperperiods required for performing task migration depends on the size of runnables and labels to be transferred, mappings, and network bandwidth, in particular flit size FS and timing constants for packet latencies while traversing one router and one link d_R and d_L.

Example 7 *(continuation of Example 6) The flit size, FS, has been fixed to 16 bits. A few examples of the number of hyperperiods required to migrate tasks from* PowerUp *to* Cluster_1*, depending on constants d_R and d_L are presented in Table 5.1. The hyperperiod length for DemoCar equals 100 ms and this time is enough to migrate all data when the router and link latencies are equal to 50 and 100 ns, respectively.*

5.2.6 On-Line Steps

In the proposed approach, only two steps are performed on-line: *Detection of current mode* and *Mapping switching*. Both of these steps are characterised with low computational complexity and thus they impose low overhead for the system during run time.

We assume that the system states are defined explicitly and there is a possibility of determining the current state by observing some system model variables, similarly to [104]. Otherwise, the most efficient multi-choice knapsack problem (MMKP) heuristics, listed in the brief survey earlier, have to be applied to identify the current mode on-line, as proposed in [73].

When the mode change is requested, an agent residing in each core prepares a set of packages with runnables to be migrated via the network. This agent is configured statically and is equipped with a table with information which runnables have to be migrated during a particular mode change. Then the precopy of these runnables is performed. In the following hyperperiods, runnables are transported using periodic servers of the length determined statically in step *Schedulability analysis for taskset during mode change*. The agent is aware of the number of periodic server instances that have to be used during the whole migration process (as in example in Table 5.1), and have the

Table 5.1 Number of hyperperiods (100 ms) required for switching between states *PowerUp* to *Cluster_1* in DemoCar depending on router (d_R) and one link latencies (d_L)

d_R [ns]	d_L [ns]	No. of Hyperperiods
50	100	1
100	200	2
100	400	3
200	500	4
400	800	6
500	1000	7

volatile portion of the context identified. If this instance number elapses, the runnables that have been migrated are killed.

Simultaneously, the same agent can receive migration data from other agents in the network. After the appropriate number of hyperperiods, the contexts of these runnables are fully migrated and are ready to be executed by the operating system.

The details of the agent depend on the underlying operating system.

5.3 Related Works

Systems with distinguishable operating modes are increasingly popular in research. A number of research activity aims at developing design-time (off-line) heuristics to reduce the number of operating points, since the amount of possible scenarios is typically prohibitively high [89]. This Design Space Exploration (DSE) process can be carried out using classic heuristic techniques (including genetic algorithms [73, 89, 116], tabu search [115], simulated annealing [160], particle swarm optimization [103], etc.), or with techniques for pruning the design space [107], performing statistical analysis for identifying potentially benefiting operating points, or use a priori knowledge of the target platform [14]. Then during run-time of that system, a run-time manager (RTM) chooses an appropriate operating point according to the available platform resources by solving an instance of multi-dimensional multi-choice knapsack problem (MMKP). Despite MMKP belongs to the NP-hard complexity class, there exists a number of light-weight greedy heuristics facilitating finding a quasi-optimal mapping during run-time [73]. Alternatively, in [104], there is a possibility of determining the current mode out of explicitly given set by observing some variables of the model. In our work, the current mode is determined in a similar way.

Two different mapping approaches are proposed in [119]. In the first one, named global static power-aware mapping, each task is assigned to one particular processing element independently from the actual scenario. This approach reduces the amount of memory required for storing the configurations and increases the efficiency of run-time management. However, it results in increased power consumption in comparison with the second approach, dynamic power-aware scenario-mapping, where this assumption is relaxed and different mappings for scenarios are stored. These approaches do not allow task migration – once a task is assigned to a processing element, it remains there until finishing its computation. In contrast, Benini et al. [15] allowed tasks to migrate between processing elements when the envisaged performance increase is higher than the precomputed migration cost. This analysis is performed at each instance of configuration change.

In order to analyse the worst case switching time between two modes, it is helpful to show the possible modes and transition between them in a formal way, using for example Finite State Machines (FSMs), as proposed in [118]. In this way it is possible to enumerate all allowed modes and transitions, and to check the cost of mode switchings. In [55], an average switching time overhead for H.264 decoder has been measured to be equal to 0.2% of the total system time. This sligh value has been caused by a low number of existing modes, obtained due to the clustering, and thus relatively rare switches. In hard real-time systems such decrease of modes by clustering is even more crucial and thus it is incorporated in the proposed design flow. In [137], the authors suggest to map as many tasks as possible to the same core in various modes to avoid the data or code items to be moved between different resources when switching between modes. In the proposed approach, we use a genetic algorithm to minimize the amount of data to be migrated.

To perform a schedulability analysis during mode changes, the data migration work is performed during time slots allocated to a periodic server. There exist different kinds of such servers, including *polling servers*, *sporadic servers* and *deferrable servers*, with different replenish policies of server execution time [40]. Despite these differences, their period and maximum execution time during one period are selected in a way that the chosen end-to-end scheduling test proves that no deadline is violated. To decrease the timing of best-effort (i.e., migration) task execution, the *best-effort bandwidth server* includes a slack reclaiming procedure and an algorithm for determining appropriate server parameters [11]. An example of a direct application of *synchronized deferrable servers* for multi-core systems has been demonstrated in [159], but its authors assumed the migration cost to be either negligible, or added to the worst-case execution time of each task, which is difficult to be applied in systems with network architecture prone to contentions. In the proposed approach, more realistic migration time is evaluated, taking into account network parameters and interference from other flows.

In [20], it was experimentally shown that even a total freezing task migration strategy, i.e., where the migrated task is stalled while all its code and data are transferred through links to the target core, can be used in a NoC-based environment and still improve the fulfilment of task deadlines in soft real-time systems. To guarantee hard timing constraints, freeze time should be bounded and possibly short. One of the possible techniques is a precopy strategy, where code and data of some tasks are copied before the actual switching. However, this technique is more complicated than total freezing and has higher migration time, as some data portions may be required to be copied more than once due to their modifications [93]. Storing task code in a few cores and transferring only the necessary data is another possibility. However, in doing so, the storage overhead at each core can increase by a large amount.

A method to guarantee hard real-time for task migration is proposed in [98]. However, a costly schedulability analysis is performed during runtime. No experiments supporting their proposed approach is provided, but one may predict that the overload of that dynamics could be considerable.

The research described in [104] is the closest to the approach proposed in this chapter. Its authors have identified mode transition points in an engine management system, and shown that a load distribution by mode-dependent task allocation is better balanced in comparison with a static task allocation. The performance has been evaluated by simulation, but, contrary to our approach, the task migration costs have not been considered.

From the literature survey it follows that designing real-time systems with distinguishable operating modes has been mainly limited to soft timing constraints. According to the authors' knowledge, there is no proposal of any method guaranteing no hard deadline violation during task migrations. In particular, schedulability analysis has not been applied to check the feasibility of task migration process or to determine the worst case switching time between two operating modes except of the positional paper [98].

5.4 Summary

An approach of task migration in a multi-core network-based embedded system has been proposed as a way to decrease the number of cores needed for guaranteing safe execution of a hard real-time software. The steps to be performed statically have been described in details using illustrative examples based on a lightweight engine control unit. A Finite State Machine describing mode changes has been extracted from the software code and transaction probabilities have been identified during simulation. The closely related modes have been merged into clusters. The most frequent transactions have been identified with the classic Prim's algorithm, and a genetic algorithm has been used to determine the runnable-to-core mapping for the initial mode. Similarly, a genetic algorithm minimizing the number of migrated data has been used for selecting the runnables to be migrated when a change of the current mode is requested. The migration time has been evaluated using schedulability analysis depending on the network bandwidth.

The proposed approach requires development of an agent realising the migration process. Since its architecture details depend on the underlying operating system, its implementation and evaluation in real embedded environments are planned as a future work.

6

Swarm Intelligence Algorithms for Dynamic Task Reallocation

The resource allocation mechanisms discussed so far in this book have some features in common: they have (some) a priori knowledge about the application load they are managing, and they are executed in a centralised or hierarchical manner. In this chapter, we explore an approach that does not require explicit information about application load, and that is able to make decisions about resource allocation in a distributed fashion. To better motivate such an approach, let us consider a concrete case study.

Multimedia applications such as video and audio processing are among the most communication and computation-intensive tasks performed by current embedded, high-performance and cloud systems. Hardware platforms with hundreds of cores are now becoming a preferred target for such applications [138], as the computational load in video decoding can be parallelised and distributed across the different processing elements on the chip to minimise metrics such as overall execution time, power or even temperature.

An important design constraint in such systems is performance predictability. There is plenty of evidence showing that inconsistent performance in video decoding applications can lead to reduced user engagement [46]. However, the task of predictably manage multimedia load is not trivial. Video decoding execution times vary greatly depending on the spatial and temporal attributes of the video [63]. Furthermore, when decoding multiple streams of live video (e.g., multipoint video conferencing, multi-camera real-time video surveillance, multiple view object detection), the workload characteristics became increasingly dynamic and thus difficult to model. Thus, efficient resource allocation is critical in achieving load balance, power/energy minimisation and latency reduction [43].

Centralised resource management with a master-slave approach, for instance, is a straightforward way to approach this problem, and is probably good enough for small systems, but there are many issues that appear as one scales up the amount of load to be handled [4, 127]. Cluster based resource management techniques have been introduced (e.g., [69]) to overcome the limitations of centralised systems by partitioning the system resources and employing multiple cluster managers. Despite those efforts, the complexity of dynamic applications and the avaliability of distributed large-scale multiprocessor systems motivate the investigation of fully-distributed, autonomous self-organising/optimising mechanisms [21, 97, 145]. Such systems should be able to adapt or optimize itself to changing workload and internal conditions and to recover from faults. Many of these systems implement self-management features by autonomously controlling and adapting task allocation and resource management at runtime.

As a basis for a distributed resource allocation approach, this chapter focuses on the behaviour of biological systems. More specifically, we study the swarm intelligence phenomenon, where the individual decisions made by the members of a swarm in a distributed manner can result in a global behaviour that is beneficial to the whole group. By applying such approach to the management of multi-stream video processing load, we hope to show its potential and to hint on its applicability to similar kinds of resource management problems.

6.1 System Model and Problem Formulation

6.1.1 Load Model

We consider a load model (Figure 6.1) consisting of workflows which resemble a container for parallel video stream decoding requests that may arrive at arbitrary times, but respecting a specified inter-arrival time. Such load model is general enough to describe systems handling a time-varying number of parallel video decoding streams. A video stream consists of an arbitrary number of *N* independent jobs. Each job (J_i) represents a MPEG group of pictures (GoP), and is modelled via a fixed dependency task-graph, and takes the structure defined in Figure 6.1. Each node in the task-graph is a MPEG-2 frame-level decoder task, and has fixed precedence constraint and communication flow shown via the graph edges. A decoder task can only start execution *iff* its predecessor(s) have completed execution and their output data is available. A decoder task τ_i is characterised by the following tuple: (p_i, t_i, x_i, c_i, a_i); where p_i is the fixed priority, t_i is the period, x_i is the actual execution time, c_i is the worst-case computation cost and a_i is the arrival time of the decoder task τ_i. Decoder tasks are preemptive and have a fixed priority. Tasks within a job are assigned fixed mapping and priorities at the start of the video stream; these exact assignments are used for all tasks of all subsequent jobs in a video stream. Tasks of low resolution video streams are given higher priority over high-resolution video streams, to ensure low-resolution video streams have a lower response-time.

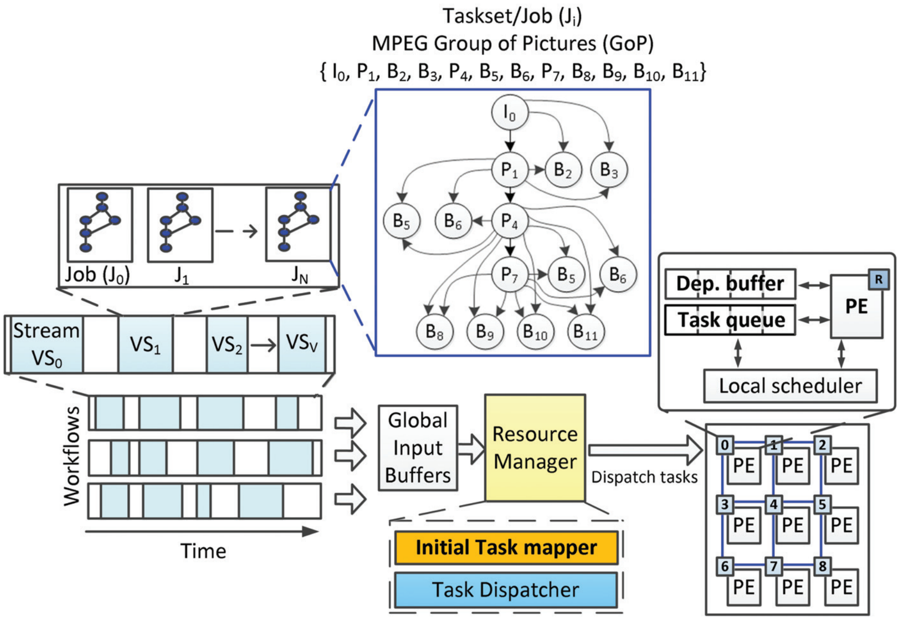


Figure 6.1 System overview diagram.

The spatial resolution of a video stream will correspond directly to the computation cost of the task and the payload of the message flows generated by those tasks. The exact execution time of the tasks are unknown in advance; however, it is assumed that the worst-case computation cost can be estimated. Subtask deadlines are unknown but each job is considered schedulable if it completes execution within its end-to-end deadline ($J_i^r \leq D_{e2e}$). The response-time of a job (denoted J_i^r) is the arrival time of the job to the point in time which all of its subtasks have completed execution. A job is considered late when $(J_i^r - D_{e2e}) > 0$ and late jobs impact the viewing quality of experience (QoE) of the real-time video stream. We assume that the arrival rate of jobs are sporadic, and the arrival pattern of new video decoding streams are aperiodic.

Once a task has completed execution, its output (i.e., the decoded frame data) is immediately sent as a message flow to the processing element executing its successor child tasks, as well as to a buffer in main memory. Message flows inherit the priority of their source tasks, with an added offset to maintain unique message flow priorities. A message flow, denoted by Msg_i is characterised by the following tuple: (P_i, T_i, PL_i, C_i); where P_i is the priority, T_i is the period, PL_i is the size of the message payload and C_i is the maximum no-load latency of message flow Msg_i, which can be calculated a priori and usually depends on the topology of the multiprocessor interconnect and on the total size of the message (i.e., payload plus headers and other overheads).

6.1.2 Platform Model

The multiprocessor platform we target in this chapter is composed of P homogeneous processing elements (PEs) connected by a Network-on-Chip (NoC) interconnect. Each PE has a local scheduler that handles a task-queue which is contained within its local memory. The PEs are directly connected to the NoC switches which route data packets towards any destination PE. We assume the NoC in our platform model uses fixed priority preemptive arbitration, has a 2D mesh topology and uses a deterministic routing algorithm such as in [19].

In such a platform, the no-load latency C_i of a message flow as given in Equation (6.1) includes the hop-distance and the number of data units (i.e., header and payload flits).

$$C_i = (numHops \times arbitrationCost) + (numFlits) \tag{6.1}$$

We assume that the NoC link arbiters can preempt packets when higher-priority packets request the output link they are using. This makes it easier to predict the outcome of network contention for specific scenarios. We assume all inter-PE communication occurs via the NoC by passing messages. Once a task is released from a global input buffer, it is sent to the task queue of the assigned PE. The PE upon completing a tasks execution, transmits its output to the appropriate PEs dependency buffer. Once a task has completed, the local scheduler picks the next task with the highest priority with dependencies fulfilled, to be executed next. The resource manager (RM) of the system (Figure 6.1), performs *initial task mapping* and priority assignment and *task dispatching* to the PEs. It also maintains a task-to-PE mapping table of the jobs of every admitted and active video stream in the system. The mapping table is essentially a hash-table where keys are task-identifiers and values are node-identifiers. In this chapter, the terms RM and $dispatcher$ are interchangeable as task dispatching is a functionality of the RM. The main responsibility of the RM is to make initial mapping decisions for new video streams, and to dispatch tasks to the mapped PEs according to the task-to-PE mapping table. Most importantly, the system is open-loop as the RM does not gather monitoring information from the PEs.

6.1.3 Problem Statement

In a centralised closed-loop system, PEs would continuously feedback the state of the tasks they were allocated (e.g., their completion time) to a central manager via status message flows. The central manager would then have global knowledge of the system in order to make efficient resource management decisions for future workloads. As discussed in [4, 127], these advantages come at the price of higher communication traffic, congestion hot-spots, higher probability of failure, and bottlenecks around the centralised manager. Furthermore, such issues are made worse as the NoC size and workload increase.

Cluster-based distributed management approaches can offer a certain degree of redundancy and scalability by varying the number of clusters and respectively local cluster managers. However, appropriate cluster size selection is vital to balance communication-overhead/performance; for example, cluster monitoring message flow routes and the cluster manager processing overhead will increase as the cluster size increases. Furthermore, the local cluster managers are still points of failure in the system, where if one of them fails the respective cluster of nodes will severely degrade in performance.

Fully distributed approaches offer higher levels of redundancy and scalability over cluster based approaches for large scale systems, due to not having any central management nodes. However, due to the lack of global knowledge and no monitoring being performed by a centralised authority, the system may be load-unbalanced, and cause jobs to miss their deadlines and become late. To reduce this job lateness of the admitted dynamic varying workload, we follow a bio-inspired distributed task-remapping technique with self-organising properties. This technique builds upon existing bio-inspired load balancing approaches by Caliskanelli et al. [30] and Mendis et al. [91].

6.2 Swarm Intelligence for Resource Management

6.2.1 PS – Pheromone Signalling Algorithm

A distributed load-balancing algorithm based on the pheromone signalling mechanism used by honey bees has been introduced in [30]. That algorithm, henceforth referred to as the *PS algorithm*, was originally developed to improve availability in wireless sensor networks but has features that make it attractive as a general load balancing approach. It is based on the concept of queen nodes (QN) and regular nodes (WN) in a network, drawing inspiration from queen bees and worker bees. The algorithm mimics the process of pheromone propagation by queen bees, which by doing so prevent the birth of new queens. If a queen dies or leaves the hive, the pheromone levels decay and worker bees are then triggered to feed larvae with royal jelly and thus differentiate one of them into becoming the new queen. Perhaps counter-intuitively, the PS algorithm uses the pheromone signalling process to select queen nodes that will be allocated workload (there can be many queens in a system), while worker nodes are not allocated any load unless they become queens themselves (which is a completely different behaviour from the biological system that inspired the approach). QNs are dynamically differentiated from other nodes to indicate they are ready to handle a workload, and the aim of the algorithm is to produces sufficient QNs to handle all the required system functionality.

The algorithm is based on the periodic transmission of pheromone by QNs, and its retransmission by receipients to their neighbours. The pheromone level at each node decays with time and with distance to the source. All nodes accumulate pheromone received from QNs, and if at a particular time the pheromone level of a node is below a given threshold this node will differentiate itself into a QN. This typically happens when this node is too far from other QNs. The PS algorithm consists of three phases, which are executed asynchronously on every node of the network: two of them are time-triggered (differentiation cycle and decay of pheromone) and one of them is event-triggered (propagation of received pheromone).

The first time-triggered phase, referred to as the differentiation cycle (Algorithm 6.1), is executed by every node of the network every T_{QN} time units. On each execution, the node checks its current pheromone level h_i against a predefined level Q_{TH}. The node will differentiate itself into QN (or maintain its QN status) if $h_i < Q_{TH}$, otherwise it will become a WN. If the node is a QN, it then transmits pheromone to its network neighbourhood to make its presence felt. Each pheromone dose hd is represented as a two-position vector. The first element of the vector denotes the distance in hops to the QN that has produced it (and therefore is initialised as 0 in line 4 of Algorithm 6.1). The second element is the actual dosage of the pheromone that will be absorbed by the neighbours.

Algorithm 6.1 PS Differentiation Cycle

```
Input: Differentiation period $T_{QN}$, local pheromone level $h_i$, local threshold
       $Q_{TH}$, initial pheromone dosage $h_{QN}$
Output: Pheromone dose $hd$, Queen Node status $QN_i$
1 while true do
2     if $h_i < Q_{TH}$ then
3         $QN_i$ = true;
4         broadcast $hd$ = $\{0, h_{QN}\}$;
5     else
6         $QN_i$ = false;
7     end
8     wait for $T_{QN}$;
9 end
```

The event-triggered phase of PS deals with the propagation of the pheromone released by QNs (as described above in the differentiation cycle) and received at neighbouring nodes. The purpose of propagation is to extend the influence of QNs to nodes other than their directly connected neighbours. Propagation is not a periodic activity, and happens every time a node receives a pheromone dose. Its pseudocode appears in Algorithm 6.2. Upon receiving a pheromone dose, a node checks whether the QN that has produced it is sufficiently near for the pheromone to be effective. It does that by comparing the first element of the vector hd with a predefined $threshold_{hopcount}$. If the hd has travelled more hops than the threshold, the node simply discards it. If not, it adds the received dosage of the pheromone to its own pheromone level h_i and propagates the pheromone to its neighbourhood. Before forwarding it, the node updates the hd vector element by incrementing the hop count, and by multiplying the dosage by a decay factor $0 < K_{hopdecay} < 1$. This represents pheromone transmission decaying with distance from the source. Figure 6.2 shows four WNs connected to a QN and retransmitting a lower dose of pheromone to their neighbours.

The second time-triggered phase of the algorithm, shown in Algorithm 6.3 is a simple periodic decay of the local pheromone level of each node. Every T_{decay} time units, h_i is multiplied by a decay factor $0 < K_{timedecay} < 1$. It can be easily inferred from the PS differentiation cycle that each node makes its own decision on whether and when it becomes a QN by referring to local information only: its own pheromone level h_i. This follows the principles of swarm intelligence, where decisions are based on local information and interactions within a small neighbourhood.

The computational complexity of the PS algorithm is very low, as each of the phases is a short sequence of simple ALU operations. The communication complexity, which in turn determines how often the PS propagation step

Algorithm 6.2 PS Propagation Cycle

```
Input: Propagation threshold $threshold_{hopcount}$, decay factor $K_{hopdecay}$,
       pheromone dose $hd$
Output: Updated pheromone dose $hd$
if $hd$ received then
    if $hd[1] < threshold_{hopcount}$ then
        $h_i = h_i + hd[2]$;
        broadcast $hd = \{hd[1] + 1, hd[2] \times K_{hopdecay}\}$;
    else
        drop $hd$;
    end
end
```

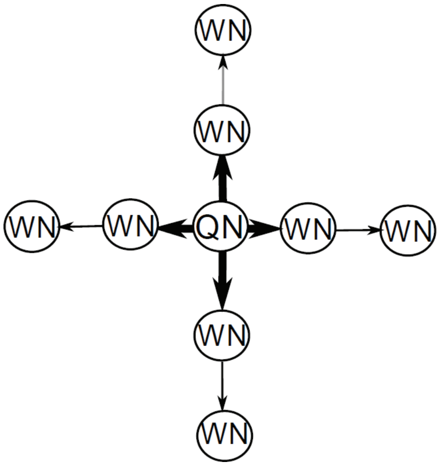


Figure 6.2 PS pheromone propagation

Algorithm 6.3 PS Decay Cycle

```
Input: Decay period $T_{DECAY}$, local pheromone level $h_i$,
decay factor $K_{timedecay}$
Output: Updated local pheromone level $h_i$
while true do
    $h_i = h_i \times K_{timedecay}$;
    wait for $T_{DECAY}$;
end
```

is executed, depends on the connectivity of the network and on the T_{decay} parameter. The protocol also provides a stability property, in that a lone node with no peers will become and always remain a queen node after a given delay, unlike in other distributed resource management approaches where nodes may be probabilistically deactivated for some intervals.

6.2.2 PSRM – Pheromone Signalling Supporting Load Remapping

The PS algorithm described in the previous subsection can be used as a general-purpose load balancing approach. Given a set of parameters, it will converge to a set of QNs that will then serve the system workload as it arrives. Once a QN becomes fully utilized, it can simply decide not to be a QN anymore. By stopping pheromone propagation, its neighbours' pheromone levels will reduce over time due to the decay phase of the algorithm, until one or more will have their levels below the threshold and will differentiate themselves into QNs. The new QNs will then be ready to handle new workload as it arrives.

In this subsection, we explore another possibility: using the PS algorithm to handle load remapping. In this case, it enables distributed resource allocation to optimise a centralised allocation mechanism which may not be fully aware of the state of each resource. Such variation of the PS algorithm has been applied to the problem of dynamically allocating video streams to a NoC-based platform, as described in Section 6.1.

Algorithm 6.4 shows extensions made to the original PS differentiation cycle, aiming to support the remapping functionality. In the original algorithm, Q_{TH} is fixed as a parameter of the algorithm. In this case, Q_{TH} is dynamically adjusted depending on the workload mapped on the resource (namely, the PE connected to the NoC). The *cumulative slack* of the tasks mapped on the PE is used to vary the QN threshold Q_{TH}, such that a node will differentiate itself into a QN if it has enough slack to accommodate additional tasks (line 4). The slack of a task is calculated as the difference between the relative deadline (d_i) and the observed response-time of the task r_i. A negative cumulative slack value indicates the PE does not have any spare capacity to take additional

Algorithm 6.4 PSRM Differentiation Cycle

```
   Input: Differentiation period $T_{QN}$, local pheromone level $h_i$,
   local threshold $Q_{TH}$, initial pheromone dosage $h_{QN}$
   Output: Pheromone dose $hd$, Queen Node status $QN_i$
1  while true do
       /* calc. normalised cumulative TQ slack */
2      $TQ_{Slack} = \frac{\sum_{\forall \tau_i \in PE_{MPT}} (d_i - r_i)}{\sum_{\forall \tau_i \in PE_{MPT}} (d_i)}$;
       /* calc. QN threshold */
3      if $TQ_{Slack} > 0$ then
4          $Q_{TH} = Q_{TH} \times (1 + (TQ_{Slack} \times Q_{TH}^{\alpha}))$;
5      else
6          $Q_{TH} = h_i \times Q_{TH}^{\beta}$;
7      end
       /* determine queen status */
8      if $h_i < Q_{TH}$ then
9          $QN_i$ = true;
10         broadcast $hd = \{0, h_{QN}, QN_{xy}, PE_{MPTinfo}\}$;
11     else
12         $QN_i$ = false;
13     end
14     wait for $T_{QN}$;
15 end
```

tasks, and hence the node is converted or remains a worker node. Line 2 in Algorithm 6.4 shows the calculation of the task queue (TQ) cumulative slack (TQ_{Slack}) of the mapped tasks. If TQ_{Slack} is positive, Q_{TH} is incremented by a ratio defined by $(TQ_{Slack} + Q_{TH}^{\alpha})$; where $\{Q_{TH}^{\alpha} \in \Re \mid 0 \leq Q_{TH}^{\alpha} \leq 1\}$ is a parameter of the algorithm. If TQ_{Slack} is negative, then the algorithm ensures the node does not become a QN in this differentiation cycle, by setting Q_{TH} as a proportion of h_i as given in Line 6; here $\{Q_{TH}^{\beta} \in \Re \mid 0 \leq Q_{TH}^{\beta} \leq 1\}$ is also a parameter of the algorithm. The self-organising behaviour of the distributed algorithm (specifically the Differentiation cycle in Algorithm 6.4), stabilises the number and position of the QNs in the NoC, as time progresses and depending on the workload. A node propagates pheromones immediately after it becomes a queen (line 10). We represent the pheromone dose (hd) as a four position vector containing the distance from the QN, the initial dosage (h_{QN}), the position of the QN in the network (QN_{xy}) and a data structure ($PE_{MPTinfo}$) containing the p_i and c_i of the tasks mapped on the QN. The worker nodes will receive and store this information as the pheromones traverse through the network.

Now that the extension to PS has been introduced, let us focus on its integration to a centralised mapping of video stream tasks. We assume the centralised mapping is performed according to a lowest worst-case utilisation heuristic. Once mapped, a task within a job may be late due to the PE or network route being over-utilised and/or due to the heavy blocking incurred by higher-priority tasks and flows. We then aim to change the task-to-PE mapping of late tasks, such that these causes of lateness can be mitigated. The task-remapping procedure (Algorithm 6.5) is executed by each PE periodically, using only its local knowledge gathered via the pheromone doses.

Algorithm 6.5 illustrates the proposed remapping procedure that utilises the adapted PS algorithm, denoted as $PSRM$. The following steps occur at each remapping cycle (seen in Figure 6.3). Firstly the task with the maximum lateness τ_L^{MAX} from the PE task queue, is selected as the task that needs to be remapped to a different PE (line 1). The deadline of a task (d_i) is calculated as a ratio of the end-to-end job deadline (D_{e2e}), as given in [65]. Each node is aware of the nearest QNs (Q_{List}), and their mapped tasks, by storing the information received from each pheromone dose hd. In Lines 3–10, the algorithm evaluates the worse-case blocking that will be experienced for the target task τ_L^{MAX} and the number of lower priority tasks that will be blocked, by mapping it onto each $Q_i \in Q_{List}$. Once a list of QNs with lower blocking than the current blocking is obtained (lines 7–9), they are requested (RQ) for their $availability$ (line 11) via a low payload, high priority message flow. The QNs reply (REP) with its availability (i.e., if other worker nodes have been remapped to a QN in that remapping cycle, then the QNs' availability is set to $false$). This avoids unnecessary overloading of QNs. τ_L^{MAX} will then be remapped to the

Algorithm 6.5 PSRM Remapping

```
1  while true do
       /* find most late task from task queue */
2      $\tau_i^{MAX\_L} = MAX(\{\tau_i \in TQ \mid (a_i + d_i) \leq t_c\})$;
       /* get current blocking for late task */
3      $B(\tau_i^{MAX\_L}) = getCurrentBlocking(hp(\tau_i^{MAX\_L}))$;
       /* find suitable QNs which offer lower blocking, than
          current blocking */
4      $Q_{List}^{B} = \{\,\}$;
5      foreach $Q_i \in Q_{List}$ do
           /* get target task blocking factor */
6          $Self\_B_Q = \sum_{\forall \tau_j \in hp(\tau_i^{MAX\_L})} c_j$;
           /* get number of lower priority tasks */
7          $LP_{size} = \left|lp(\tau_i^{MAX\_L})\right|$;
8          if $Self\_B_Q < B(\tau_i^{MAX\_L})$ then
9              Insert $\{Q_i, LP_{size}\}$ to $Q_{List}^{B}$;
10         end
11     end
       /* request for QN availability */
12     $Avlb\_Q_{List}^{B} = requestAvailability(Q_{List}^{B})$;
       /* get available QN that has least amount of lower
          priority tasks */
13     $\{Q_i^{MIN\_LP}, LP_Q^{MIN}\} = MIN(Avlb\_Q_{List}^{B})$;
       /* Update dispatcher task-mapping table */
14     Notify dispatcher: $\tau_i^{MAX\_L} \rightarrow PE(Q_i^{MIN\_LP})$;
15     wait for $T_{RM}$;
16 end
```

QN with the least number of lower priority tasks (denoted $Q_i^{MIN\_LP}$) from the available QN list ($Avlb\_Q_{List}^{B}$) (line 12). Finally, the task dispatcher is notified via message flow to update the task mapping table; the dispatcher looks up the task-id in the table and updates the corresponding node-id with the new remapped node-id. When the tasks of the next job of the video stream arrives into the system they will be dispatched to the node-id indicated by the updated mapping table. Therefore, remapping will only take effect from the subsequent arrival of the next job in the video stream. Even though there is an update message sent to the dispatcher at a remapping event, the remapping decision is achieved purely using local information at each PE, based on the PSRM algorithm.

Figure 6.4 illustrates an example of the remapping procedure in a 4 × 4 NoC. The (x, y) coordinates refer to the processing node in column x and row y. In step 1 of Figure 6.4, at each remapping interval (T_{RM}) each PE

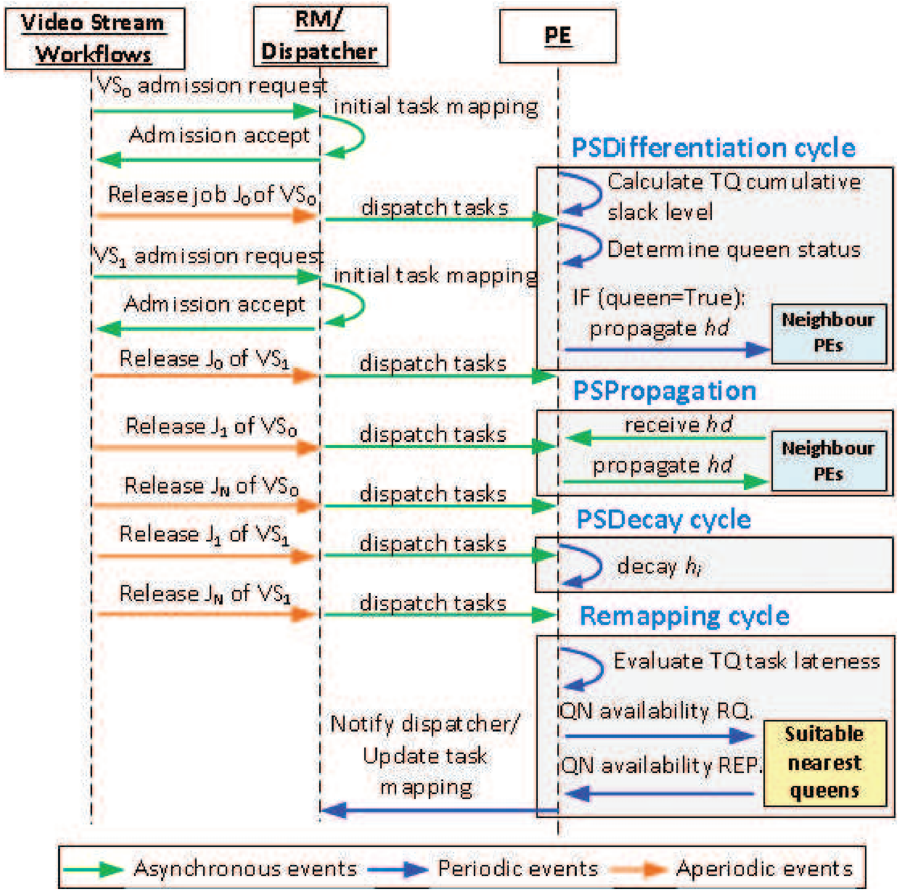


Figure 6.3 Sequence diagram of PSRM algorithm related events. Time triggered (periodic): *PSDifferentiation*, *PSDecay* and *Remapping* cycles; Event triggered: *PSPropagation*.

identifies the late tasks in their task queues; they are also aware of the position of any nearby QNs due to the pheromone signals. τ_1 and τ_2 on PE(1,0) and PE(2,2) are tasks that are late, at that time instant. In step 2, they determine the suitability of each QN to remap the late tasks to. τ_1 can either be remapped to Q(1,1) or Q(3,0); and τ_2 can be remapped on to either Q(3,2) or Q(1,1) but Q(3,2) is not suitable due to the task blocking behaviour and Q(0,3) is not in the Q_{List} due to distance. In step 2, the nodes request for the suitable QNs' availability; in this instance PE(1,0) obtained a lock on Q(1,1) first. Hence, τ_1 will be remapped onto Q(1,1) and τ_2 will be remapped to Q(2,3). In step 3

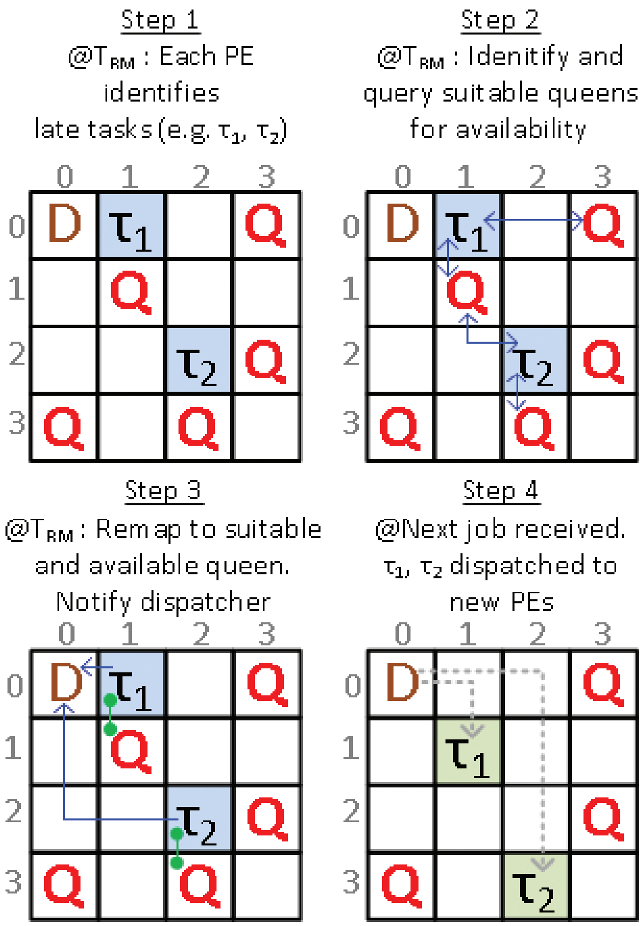

Step 1
@T_RM : Each PE identifies late tasks (e.g. τ1, τ2)
Step 2
@T_RM : Idenitify and query suitable queens for availability
Step 3
@T_RM : Remap to suitable and available queen. Notify dispatcher
Step 4
@Next job received. τ1, τ2 dispatched to new PEs
0 1 2 3
D
Q
τ1
τ2

Figure 6.4 Task remapping example. (Q = queen nodes; D = Dispatcher; $[\tau_1, \tau_2]$ are late tasks; Blue lines represent communication.

the PEs notify the dispatcher via a message flow regarding the remapping. In step 4 the next job arrives and τ_2 and τ_3 are now dispatched to the new processing elements – PE(1,1) and PE(2,3) respectively.

The performance of adaptive algorithms such as PSRM is highly dependent on the selection of a good set of parameters. Manual selection of parameters is not feasible due to the size of the search space. Table 6.1 shows several important parameters obtained via a search-based parameter selection method inspired by [29]. The parameters T_{QN}, T_{DECAY} and T_{RM} and their ratios

Table 6.1 PSRM algorithm parameters

Differenciation cycle (T_{QN})	0.22
Decay cycle (T_{DECAY})	0.055
Remapping period (T_{RM})	6.9
Default QN threshold (Q_{TH})	20
QN threshold inc./dec. factors (Q^{α}_{TH}, Q^{β}_{TH})	0.107, 0.01
Pheromone time and hop decay factors	0.3, 0.15
Pheromone propagation range	3

play a key role in obtaining a good performance from the algorithm. The experimental results during the parameter search process show that the remapping frequency has a significant impact in accuracy and communication overhead. The relationship between these parameters have been investigated extensively in previous work [29, 30]. As a general guideline, to keep the communication overhead low, the event cycles (T_{QN} and T_{RM}) and the QN hormone propagation range must be kept relatively low.

The platform model used in this case study has fixed priority preemptive NoC arbiters and local schedulers. Hence, tasks and flows can be blocked by higher priority tasks and flows. The remapping heuristic takes into account the new tasks' blocking incurred by a possible remapping (lines 4–10 of Algorithm 6.5). However, since the processing nodes lack a global view of the communication flows, the remapping heuristic cannot take into account the change in the overall network *communication interference pattern* caused by the reallocation of the tasks. Therefore, there are situations where remapping a task can result in an actual lateness increase. As shown in Figure 6.3 and Algorithm. 6.4, every T_{QN} time units the worker nodes get updates from all QNs in close proximity to them. However, between subsequent *PSDifferentiation* events, the workload of the QN can change rapidly when the system is heavily utilised, which may lead to inaccurate local knowledge regarding the nearby QNs. Furthermore, late tasks should be remapped ideally before the next job invocation. However, the remapping event is periodic (i.e., every T_{RM} seconds) which allows the remapping overhead to be kept at a minimum, but does not guarantee synchronisation with the workload arrival pattern. Longer periodic events may lead to inconsistency in data and states, but are used to keep the communication overhead at a minimum.

6.3 Evaluation

6.3.1 Experiment Design

To evaluate the resource allocation approach described in this chapter, we performed a number of simulations of realistic load patterns allocated over a 100-core Network-on-Chip platform. A discrete-event, abstract simulator described in [92] has been adopted. The volume of load was configured such that there would be an upper limit of 103 parallel video streams at any given time in the simulation. Experiments were performed under 30 unique workload situations, where the number of videos per workflow, their resolutions and arrival patterns vary based on the randomiser seed used in each simulation run. The computation to communication ratio of the workload was approximately 2:1. The resolution of the video streams were selected at random from a list of low to high resolutions (e.g., from 144p to 720p). The inter-arrival time of jobs in a video stream were set to be between 1 to 1.5 times the D_{e2e}. Tasks were initially mapped to the lowest utilised PE (according to worst-case utilisation) and priority assignment of the tasks followed a scheme were the lowest-resolution tasks get the highest priority. This initial mapping and assignment scheme were constant variables for all evaluations.

6.3.1.1 Metrics

The experiments have multiple dependent variables as described below:

- *Total number of fully schedulable video streams* is the number of all admitted video streams that have no late jobs (i.e., $J_i^r \leq D_{e2e}$).
- *Cumulative job lateness* (C_L^{Jobs}) is calculated as the summation of lateness of all the late jobs from every video stream (v_i) admitted to the system (Equation (6.2)). In Equation (6.2), J_i^L is a late job and VS denotes all the video streams admitted to the system. We measure the job lateness with remapping enabled/disabled, hence a reduced C_L^{Jobs}, when remapping is enabled is considered an improvement to the system. This metric gives us a notion of how the remapping technique reduced the lateness of the unschedulable video streams, which directly affects the QoE of the video stream.
- *Communication overhead* is calculated as the sum of the basic latencies (C_i) of every control signal in the respective remapping technique. In the PSRM algorithm these are the pheromone broadcast and QN availability request signals. In the cluster-based technique the PE status update traffic and the inter-cluster communication traffic contributes to the overhead. Furthermore, the task dispatcher notification messages in all the remapping techniques are included in the overhead. Lower communication overheads lead to less congested networks as well as lower communication energy consumption [39].
- *Distribution of PE utilisation* is calculated by the measured total busy time for every PE on the network during a simulation run. PE utilisation gives a notion of the workload and a lower variation in workload distribution is desirable. Overloading a single resource and/or having

a high number of idle PEs, are undesirable properties which may lead to reduced reliability and increased wear-and-tear.

$$C_L^{Jobs} = \sum_{\forall v_i \in VS} \left[\sum_{\forall J_i^L \in v_i} (J_i^r - D_{e2e}) \right] \tag{6.2}$$

$$\text{Comms. overhead} = \sum_{\forall msg_i \in ControlMsgs} C_i \tag{6.3}$$

6.3.1.2 Baseline Remapping Techniques

The PSRM resource manager was evaluated against the following baselines:

- $CCPRM_{V2}$ – is a cluster-based management proposed in [91] as an improvement of the original work by Castilhos et al. [33]. It is configured with a cluster size of 2 × 5 (i.e., 10 clusters).
- *Centralised management* – is essentially CCPRM$_{V2}$ with only one 10 × 10 cluster. A single centralised resource manager receives status updated from every slave PE in the network and performs periodic remapping as described in [91]. The manager notifies the task dispatcher of any remapping decisions.
- A *random remapper* – is a remapping scheme where, every remapping interval each PE selects the most late task in its task queue and randomly selects another node on the network to remap to. The task dispatcher is notified of the remapping event.

6.3.2 Experimental Results

6.3.2.1 Comparison between clustered approaches

The comparison of CCPRM$_{V1}$ and CCPRM$_{V2}$ for the C_L^{Jobs} metric is shown in Figure 6.5(a). In this plot a positive improvement indicates that task remapping has helped to reduce the cumulative job lateness in the admitted video streams. A negative improvement indicates that the remapping has instead worsened the lateness of the jobs. Each sample in the distribution corresponds to a simulation run with a unique workload. It is clear that the modifications made to the original CCPRM$_{V1}$ technique has resulted in an improvement in reducing job lateness. In CCPRM$_{V1}$ a majority of the data shows negative improvement, while CCPRM$_{V2}$ shows more positive job lateness improvement. However, this improvement has costed a 4% increase in communication cost. Certain constraints in the local remapping decisions in CCPRM$_{V2}$ would result in more communication with neighbouring clusters which might explain the increased overhead.

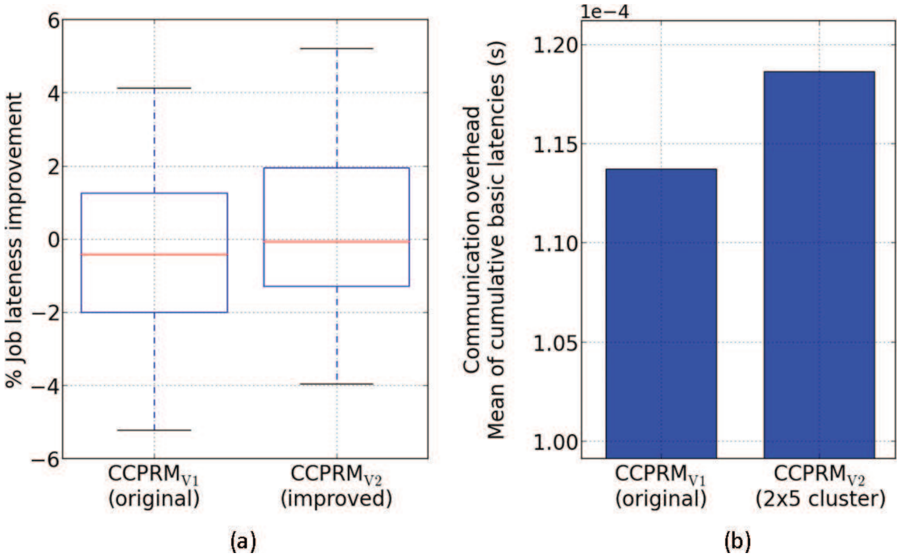


Figure 6.5 Comparison of CCPRM$_{V1}$ (original) and CCPRM$_{V2}$ (improved). (a) Cumulative job lateness improvement. (b) Communication overhead.

6.3.2.2 Comparison regarding video processing performance

Figure 6.6 shows the distribution of cumulative job lateness improvement for each of the remapping techniques. Firstly, all the techniques show both negative and positive improvements; hence, under certain workload situations the remapping techniques have failed to improve the lateness of the jobs. However, a majority of the distribution in both PSRM and CCPRM$_{V2}$ are in the positive improvement region. PSRM has a smaller spread in lateness compared to the baselines. The upper quartile and a significantly large proportion of the inter-quartile range (IQR) falls in the positive improvement area, which is not seen in any of the baselines. In over 60% of the workload scenarios PSRM will produce positive improvement to the job lateness of the video streams but the actual improvement is small (up to 3%–4%). Futhermore, in Figure 6.7, we can see PSRM is marginally better than the CCPRM$_{V2}$ in the number of fully schedulable video streams. CCPRM$_{V2}$ shows a better job lateness improvement over the centralised management, because the monitoring traffic is shorter in route-length and hence is less disruptive to the data communication. We can see that the centralised management has the highest number of schedulable video streams out of the evaluated remappers. This could indicate that CCPRM$_{V2}$ and PSRM gave significant job lateness improvements only to a few video streams while the centralised management was able to make minor improvements to multiple video streams. The random remapper shows

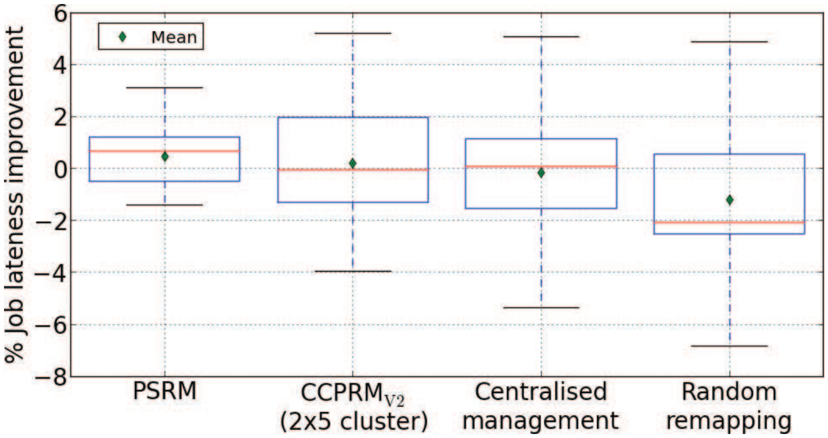


Figure 6.6 Distribution of cumulative job lateness improvement after applying remapping.

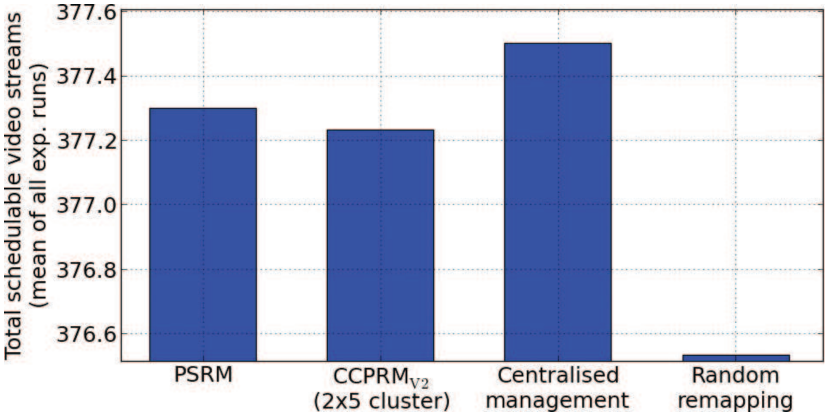


Figure 6.7 Comparison of fully schedulable video streams for each remapping technique.

the worst results with a majority of the experiments resulting in negative improvements and produces the lowest number of fully schedulable video streams. It was interesting to note that there were a few scenarios where random remapping produced significant job lateness improvements, which is seen by the high upper whisker in the box plot (Figure 6.6).

6.3.2.3 Comparison regarding communication overhead

PSRM shows a significant communication overhead reduction when compared to the baselines (Figure 6.8). The mean and IQR of PSRM communication overhead is lower than the baselines but the larger variance of the

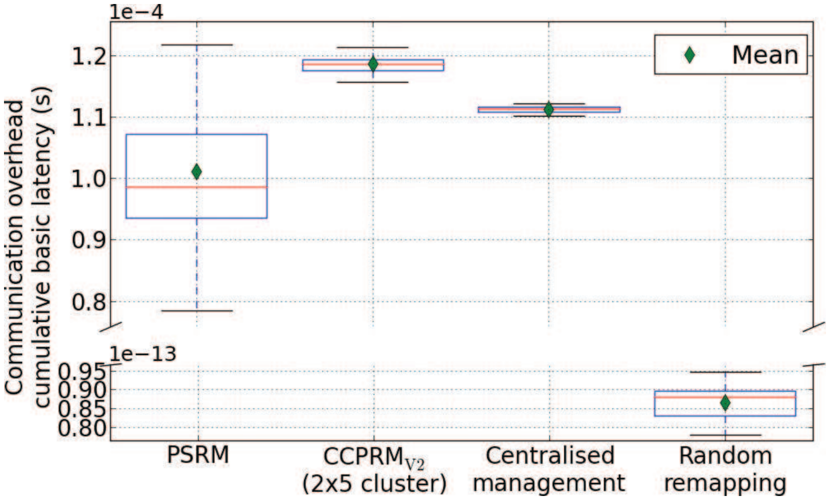


Figure 6.8 Communication overhead of the remapping approaches.

results is due to the different range of workloads and their effect on the QN differentiation cycle in each experimental run. The maximum overhead is comparable to that of CCPRM$_{V2}$. Both the CCPRM$_{V2}$ and centralised management show a higher and narrower distribution of communication overhead than PSRM. A higher upper whisker in PSRM shows that under certain workload scenarios the overhead can be costly and similar to the CCPRM$_{V2}$ baseline. The lower communication overhead distribution of the centralised manager when compared with CCPRM$_{V2}$, is due to the lack of inter-cluster communication. In the centralised management scheme communicating tasks mapped at the middle of the NoC will suffer due to the network congestion caused by the incoming monitoring traffic. Furthermore, these traffic flows will occupy longer routes than CCPRM$_{V2}$. Furthermore, we are shown in [69], that the centralised managers' communication overhead issues become severe after the NoC size exceeds 12 × 12. The random mappers' communication overhead is many orders of magnitude lower than the others as it only incurs overhead when notifying the task dispatcher regarding remapping decisions.

6.3.2.4 Comparison regarding processor utilisation

The PE utilisation distribution shown in Figure 6.9(a), indicates the PEs with higher utilisation using lighter shade, while the darker shades show PEs with low utilisation levels. The data shown in this plot are normalised such that each remapping technique is relative to each other. PSRM shows a slightly similar variation in the workload distribution to CCPRM$_{V2}$ with only a single

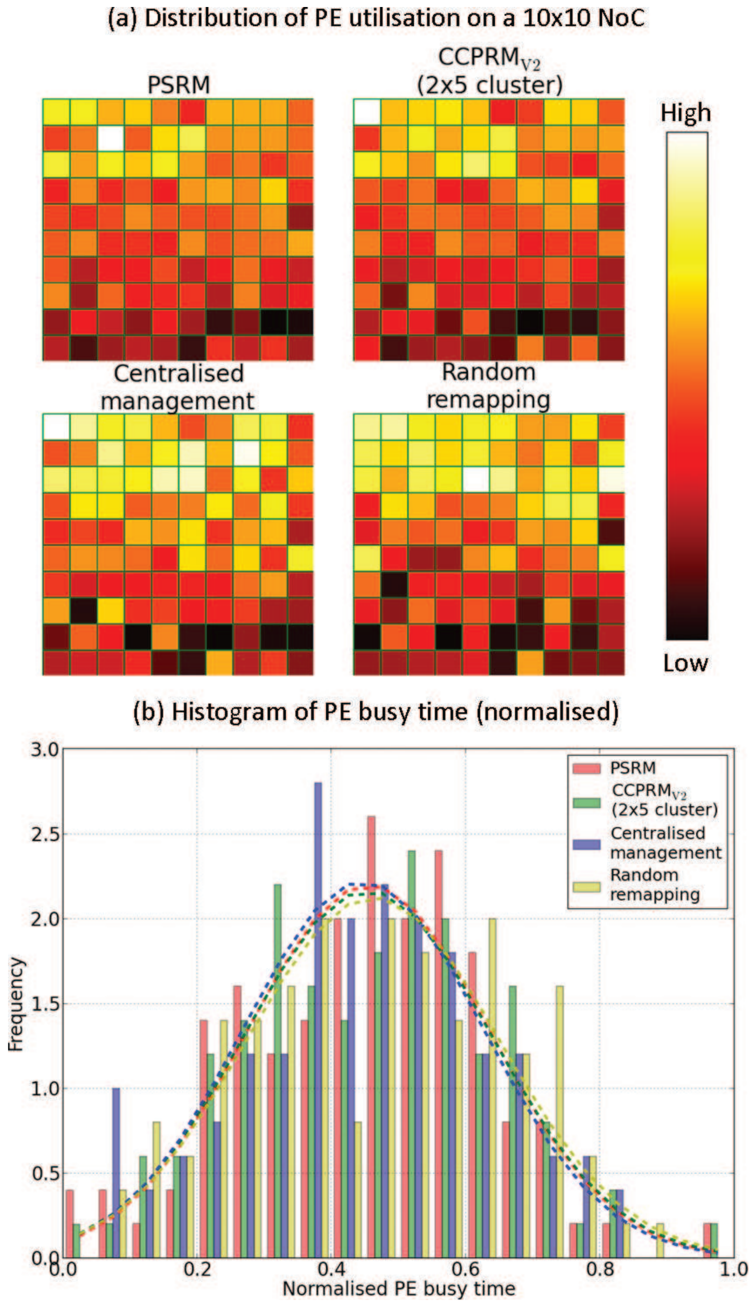


Figure 6.9 Comparison of PE utilisation for all remapping techniques. (a) Distribution of PE utilisation across a 10 × 10 NoC. (b) Histogram of PE busy time (normalised; 20 bins).

PE with extremely high utilisation and a few with very low utilisation. The curves fitted to the histogram data shown in Figure 6.9(b) indicates that all four remapping techniques have a similar spread of workload distribution. However, closer examination to the statistical properties of the distributions (given in Table 6.2), indicate that centralised management has the lowest variance and the mean utilisation. The random remapper has the highest variance and mean utilisation. The frequency spikes of the centralised and random remappers in Figure 6.9(b) at 0.8, 0.4 and 0.7 probably give rise to these statistical properties. PSRM shows a higher mean utilisation and lower distribution variance when compared with CCPRM$_{V2}$.

6.3.3 Outlook

Overall the results indicate the PSRM technique helps to reduce lateness in the video stream jobs and to increase the number of schedulable video streams when compared with the CCPRM$_{V2}$ remapper. It is important to note that this improvement, even though is marginal, comes at a much lower communication overhead (up to 30% lower than the cluster-based and centralised approaches). A higher maximum lateness improvement can be obtained using CCPRM$_{V2}$, but only in 40%–50% of the workload scenarios. Communication overhead of CCPRM$_{V2}$ may grow as the cluster sizes increase, however in the PSRM technique this overhead will vary depending on the distribution of QNs in the network. Also, unlike in the baseline approaches, in PSRM the pheromone signalling message paths are usually short (only a few hops) regardless of the NoC size increases. A centralised resource manager can help to evenly distribute the workload much better than PSRM, because of its global knowledge of the PE status and the mapped tasks. However, PSRM shows better workload distribution when compared with a cluster-based approach. Unlike in the cluster-based management, in PSRM, there are no resource managers in the network; each node executes a simple set of rules using only local knowledge to collectively improve the performance. The execution cost of the remapping event (Algorithm 6.5) is bounded by the number of QNs in the local vicinity and the number of tasks mapped on the node. In the cluster

Table 6.2 PE utilisation distribution statistics. Lower variance (var.) = better workload distribution

	mean	var.
PSRM	0.455	0.033
CCPRM$_{V2}$	0.454	0.034
Centralised management	0.447	0.032
Random remapper	0.462	0.035

based approach each local processor's execution overhead for management functions (such as remapping, inter-core communication, monitoring etc.) would increase as the cluster size increases. Unlike in the centralised approach, PSRM is decentralised hence has no single point of failure or an isolated communication congestion area. Each PE has the capability of performing remapping and becoming a QN, hence the level of redundancy in the system is greater than in the baseline remappers.

One of the identified limitations in PSRM is the sensitivity of the parameters. The parameters need to be tuned for a specific network size and can produce varying results based on the nature of the workload. Parameters that are suitable for a smaller NoC size may not necessarily produce favourable results for a larger NoC. The experimental results during the tuning of the parameters for the PSRM and CCPRM$_{V2}$ techniques show that the remapping frequency has a significant impact on the performance and communication overhead. Furthermore, depending on the value of T_{QN} and T_{DECAY}, PSRM will vary in the time it takes to identify suitable QNs in the network, and hence better performance results can be seen after longer runs of the algorithm.

6.4 Summary

This chapter presented extensions to a fully distributed resource management technique based on swarm intelligence. We have shown how such an approach can be applied to a multiple video stream decoding application with several unknown dynamic workload characteristics, on a NoC-based multicore. With low communication overhead, it relies on task-remapping strategies to progressively distribute the workload in the network and to reduce the overall job lateness.

The experimental results have shown that the bio-inspired remapper gives a marginal (2%–4%) improvement in lateness reduction but incurs 10%–30% lower communication overhead and minor improvement to workload distribution than the baseline cluster-based and centralised management schemes. The centralised management allows the system to increase the number of total schedulable video streams, but the improvement to the cumulative job lateness of the late video streams is poor and the communication overhead is higher than in the proposed technique. The benefits of the centralised management degrade as the scale of the network and workload increase [4]. Results show that the proposed PSRM approach give a marginal benefit in reducing the cumulative job lateness of the video streams when compared against the CCPRM$_{V2}$ cluster based resource management approach; however it is important to note that the improvement is obtained using a significantly less (up to 30% lower) communication overhead than the cluster based approach.

A reduced communication overhead may lead to lower energy consumption [39] and less congested communication network, making PSRM more efficient than the cluster-based approach. Furthermore, unlike in the centralised or cluster-based approaches the proposed PSRM remapping technique does not depend on a single or group of management entities. Each node is independent and capable of relocating late tasks to improve the overall job latency, hence adopting this technique introduces a high degree of redundancy for NoC-based multi/many-cores that require reliable and timely operation.

7

Value-Based Allocation

In a many-core HPC data centers, jobs arrive at different moments of time and they need to be serviced by allocating on the available system cores at run-time. In doing so, the value (utility) achieved by servicing the jobs should be maximized while trying to minimize the overall energy consumption during system operation as mentioned earlier. A job may contain a number of dependent/independent tasks or processes to be allocated on the system cores. The allocation results for each job determine the value to be achieved and also energy consumption, and thus allocation process needs to optimize both the metrics (value and energy). Optimizing of energy of large scale HPC data centers is of paramount importance as there is a huge concern about the energy required to operate such systems [112]. The reports indicate the energy consumption of data centers to be between 1.1% and 1.5% of the worldwide electricity consumption [70]. Thus, both the value and energy consumption need to be optimized during resource allocation process.

Previous researchers have introduced notion of values (economic or otherwise) of the jobs to define their importance level [66]. In overload situations where demand for available resources is higher than the supply, such a notion facilitates in deciding to hold the low value jobs for late allocation and allocating limited resources to the high value jobs. The value of a job can change over time to reflect the impact of the computation over the business processes, which adds complexity to the allocation process.

Existing dynamic resource allocation approaches allocate dynamically arriving jobs to the platform resources by employing light-weight heuristics that can find an allocation quickly. There have also been efforts to utilize design-time profiled results to facilitate efficient resource allocation and reduce the computations at run-time [125]. These efforts seem promising to design job-specific-clouds, where the clients (or customers) and their jobs to be submitted for execution are pre-defined, which can be realized from the historical data. However, they optimize only for value. Further, existing approaches optimizing for both value and energy cannot be applied to dependent tasks. Since an HPC job may contain a set of dependent tasks, there is a need to devise resource allocation approaches to be applied on dependent tasks while optimizing both value and energy.

7.1 System Model and Problem Formulation

Figure 7.1 shows our target system model, which is based on typical industrial HPC scenario. The system contains a *many-core HPC platform* that executes a set of *jobs* submitted by various *users* at different moments of time. The jobs are submitted to the *platform resource manager* that allocates resources to them. This section provides a brief overview of the platform and workload model along with the problem formulation.

7.1.1 Many-Core HPC Platform Model

The HPC platform HP contains a set of nodes ($PG_1, \ldots, PG_N$), where each node (server) contains a set of homogeneous cores, referred to as processing elements (PEs), as shown in the bottom part of Figure 7.1. Similar to a typical data center, each node represents a physical server. A node n is represented as a set of cores C_n, which communicate via an interconnect. Each core is assumed to support DVFS (as briefly described in Chapter 3) and thus its voltage and

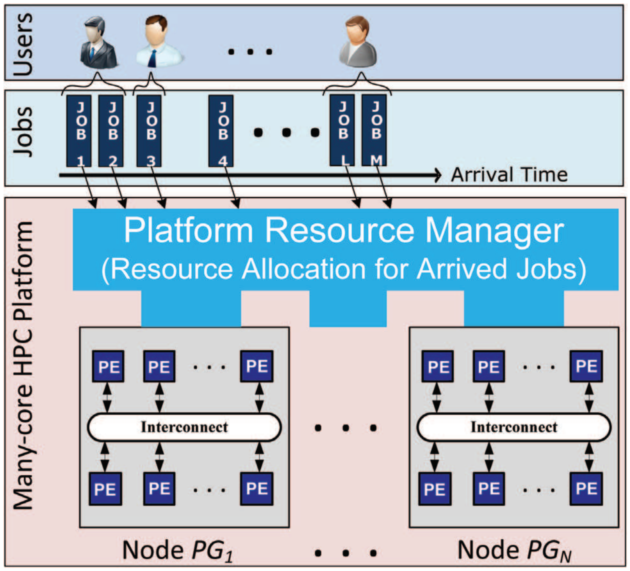


Figure 7.1 System model adopted in this chapter. A cloud data center containing different nodes (servers) with dedicated cores (PEs) to execute jobs submitted by multiple users.

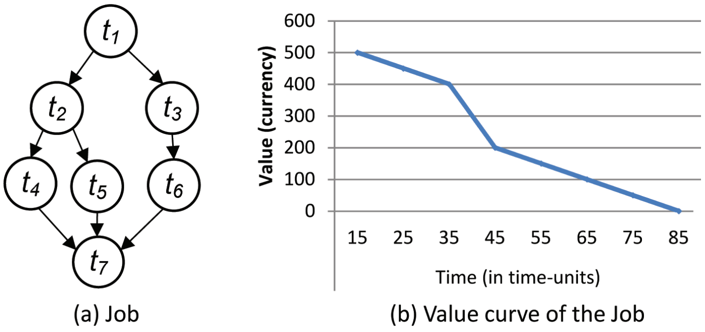
frequency can be independently adjusted in order to achieve a balance between energy consumption and job execution time. A *platform resource manager* controls access of platform resources and coordinates the execution of jobs submitted by the users, which facilitates efficient management of resources and incoming requests.

7.1.2 Job Model

Each job j in the HPC workload is modelled as a directed graph $TG = (T, E)$, where T is the set of tasks of the job and E is the set of directed edges representing dependencies amongst the tasks. Figure 7.2(a) shows an example job that contains 7 tasks (t_1, . . ., t_7) connected by a set of edges. Each task $t \in T$ is associated with its execution time (*ExecTime*, measured as worst-case execution time (WCET)), when allocated on a core operating at a particular voltage level. Such information can be obtained from previous executions of the tasks in the job from historical data. Each edge $e \in E$ represents data that is communicated between the dependent tasks. A job j is also associated with its arrival time AT_j.

7.1.3 Value Curve of a Job

For each job j, the value curve VC_j is a function of the value of the job to the user depending on the completion time of the job [66]. The value curve is usually a monotonically-decreasing function and trends towards zero with the increasing completion time, as shown in Figure 7.2(b). We assume a value curve is given for each job, as this reflects its business importance as assessed by the end user (i.e., domain specific economic model). The description of the economic model is orthogonal to our approach and out of scope of this chapter.



(a) Job (b) Value curve of the Job

Figure 7.2 An example job model and its value curve.

Each job is considered to have a soft deadline [28]. This implies that the violation of deadline does not make the computation irrelevant, but reduces its value for the user [34, 62, 66]. The reduction in value due to delay can be determined by observing the value in the value curve at the delayed completion time. Deadlines missed by large margins may result in zero value and thus the computation becomes useless for the user. Further, the energy spent on such computation can be considered as wasted. Therefore, the job request should be rejected if no (zero) value can be obtained by executing it.

7.1.4 Energy Consumption of a Job

The total energy consumption (E_{total}) of a job is computed as the sum of dynamic and static energy as follows.

$$E_{total} = E_{dynamic} + E_{static} \tag{7.1}$$

The dynamic energy consumption for all the tasks in the job is estimated from Equation (7.2).

$$E_{dynamic} = \sum_{\forall t \in T} (ExecTime[t] \rightarrow c_v) \cdot (pow \rightarrow c_v)] \tag{7.2}$$

where $ExecTime[t] \rightarrow c_v$ and $pow \rightarrow c_v$ are the execution time of task t mapped on core c operating at voltage v, and respective power consumption, respectively. The $ExecTime$ measures are provided in the job model. It is assumed that the power consumption at different operating voltages is known in advance and taken from chip manufacturer's data sheet.

The E_{static} for each core is computed as the product of overall execution time of the job and static power consumption of the used cores. For p used cores, total static energy is computed as $p \cdot E_{static}$, and unused cores are considered as power gated so that they do not contribute to the overall energy consumption.

7.1.5 Problem Formulation

In an HPC system (e.g., Figure 7.1), jobs ($j_1, \ldots, j_M$) arriving at different moments of time submitted by various users need to be efficiently allocated on the resources (cores) of the platform nodes ($PG_1, \ldots, PG_N$). The resource allocation problem targeted in this paper is to jointly optimize value and energy while servicing arrived jobs. It is assumed that the tasks of a job are allocated to only one node (server) in order to avoid huge communication delay between different nodes. To summarize, the targeted problem considers the following set of input, constraints and objective.

- **Input**: Workload, i.e., Job set ($j_1, \ldots, j_M$), Value curve of each job VC_j, Arrival time of each job AT_j ($j \in 1, \ldots, M$), Cores of the HPC platform nodes ($PG_1, \ldots, PG_N$), Voltage levels ($v_1, \ldots, v_l$) supported by each core.
- **Constraints**: Limited resources (cores) on each node of HP.
- **Objective**: Maximize overall value Val_{total} and minimize energy consumption E_{total}.

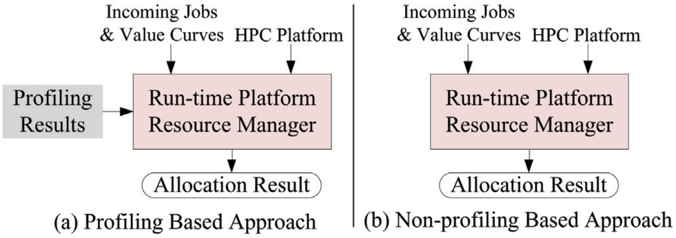
For an arrived job, the allocation process followed by the global resource manager needs to identify the node to execute the job, tasks to cores allocation inside the node, and the voltage/frequency levels of the cores executing tasks of the job. We assume negligible time for switching between voltage/frequency levels of a core as it is in the order of nanoseconds while tasks execution is in the order of minutes or hours [49]. Since there are several possible allocations (tasks to cores assignment) for a job and several voltage scaling (VS) options for each allocation, exploring the complete design space to identify the optimal design in terms of value and energy might not be feasible within acceptable time. Therefore, only efficient allocations and appropriate VS options need to be evaluated. Further, for dependent tasks, applying VS on a core is rather challenging as one needs to capture the VS effect on the execution of dependent tasks allocated on other cores.

7.2 The Solution

This section describes solutions in order to address the aforementioned problem. In order to allocate platform cores to the incoming jobs at run-time, the platform resource manager is invoked to find allocations. The manager follows profiling or non-profiling based approach, as shown in Figure 7.3. The details of these approaches are as follows.

7.2.1 Profiling Based Approach (PBA)

This approach uses design-time profiling results of the jobs in the historical data to perform run-time resource allocation for the incoming jobs, as shown



(a) Profiling Based Approach (b) Non-profiling Based Approach

Figure 7.3 Profiling and non-profiling based approaches.

in Figure 7.3(a). For each job, the profiling process identifies the allocation and voltage/frequency levels leading to optimized response time (determines value) and energy consumption when utilizing different amount of computing power in terms of number of cores. The response time is calculated as the difference between the end and start time of the job execution after allocating resources to it and should be minimized to optimize value. To jointly optimize value and energy, we consider to minimize the product of response time and energy consumption. At different number of cores, the allocation and voltage/frequency levels leading to minimum product value are identified by employing a genetic algorithm (GA) based evaluation, similarly as in [117]. The number of cores is varied from one to the number of tasks in the job. Such variation can exploit all the potential parallelism present in the job as each task can occupy only one core. For each job, the allocation, voltage/frequency levels, value corresponding to the response time and energy consumption at different number of cores are stored as the profiling results.

To perform resource allocation by using the profiling results, the manager follows Algorithm 7.1. The algorithm takes profiling results of the jobs from the storage along with their value curves and arrival times, and the HPC Platform *HP* as input and identifies the value and energy optimizing allocation for each job based on the number of available cores at different nodes in the platform. The algorithm checks mainly for two events as follows: 1) *any already allocated job(s) finish execution* to update the platform resources (lines 1–3), and 2) *any job(s) arrive into the platform* to put into a job queue (lines 4–6). If any of the two events or both of them occurs, the algorithm tries to perform resource allocation for the queues job(s) having non-zero values (lines 7–17).

To perform resource allocation for all valuable queued jobs (i.e., jobs having positive values), all of them (*count* = 0 to *JobQueue.size()*, line 8) are tried to be allocated on the platform resources as along as any core is available. It is ensured that a queued job having zero value at the allocation time is dropped from the queue as no value can be made out of it. The allocation process continues until all the arrived jobs are allocated or dropped due to having zero value while waiting in the job queue. First, bids (in terms of number of available cores) from different platform nodes are collected, then the maximum bid ($maxBid$) and the corresponding node is selected (line 9). Choosing such a node to use its cores helps to achieve better load balancing amongst nodes and thus better resource utilization. In case more than one nodes have the same amount of bid, any of them is chosen. If the estimate of $maxBid$ is greater than zero ($maxBid > 0$, line 10), i.e., at least one core is available in the platform, the value/energy estimates of jobs utilizing $maxBid$ cores are computed and the job leading to maximum value per energy consumption ($maxValuePerEnergyJob$) is selected to

Algorithm 7.1 Profiling Based Resource Allocation

```
Input: Incoming Jobs with arrival times, Jobs' profiling results and value
    curves, HPC Platform HP.
Output: Resource Allocation for Incoming Jobs.
if allocated_job(s) finish execution then
    Update platform resources;
end
if job(s) arrive then
    Put the job(s) in JobQueue;
end
if JobQueue contains job(s) having positive values then
    for count = 0 to JobQueue.size() do
        Collect $bids$ from all nodes and select $maxBid$;
        if $maxBid > 0$ then
            Compute $value/energy$ estimates of unscheduled jobs when
              utilizing $maxBid$ cores;
            Select $maxValuePerEnergyJob$ and its $value$, $energy$,
              $allocation$, and $voltage/frequency\ levels$ from profiling
              results;
            Schedule $maxValuePerEnergyJob$ on node having
              $maxBid$ cores by following the $allocation$ to perform
              execution at $voltage/frequency\ levels$;
            Update platform resources;
        end
    end
end
```

be scheduled to the node having $maxBid$ cores by following the allocation and voltage/frequency levels leading to the optimized value and energy. The computation of value/energy for each job considers its value at the allocation time and the exact number of cores to be used by the job computed as minimum between $maxBid$ and the number of cores to be used to achieve maximum value/energy. The platform resources are updated after scheduling each job to have up to date resources' availability information for the next allocation instance. This helps to achieve an accurate and efficient allocation. Similar process is repeated for all the arrived jobs.

For each job, this approach selects (from the profiling results) allocation and voltage/frequency levels leading to maximum value/energy, and thus both the value and energy consumption are optimized.

7.2.2 Non-profiling Based Approach (NBA)

The NBA approach does not use profiling results as no historical pattern of jobs is available to perform advance profiling. Rather, all the computations are performed at run-time. This approach is suitable to the scenarios when the jobs to be executed are unknown in advance, i.e., no historical pattern of jobs is available.

The steps followed by the NBA are similar to PBA and sketched in Algorithm 7.2. Here, if $maxBid$ is greater than zero ($maxBid > 0$), the following two main steps are employed: *i)* Compute $values$ of unscheduled jobs by finding allocations on $maxBid$ cores (line 6), and *ii)* Identify voltage/frequency levels of used cores to execute allocated tasks to maximize value over energy (line 8), which are described subsequently.

In step *i)*, firstly, an appropriate allocation for each job is identified by allocating on $maxBid$ cores. The allocation considers the exact number of cores to be used, which is the minimum between $maxBid$ cores and the number of cores equivalent to the number of tasks in the job. The exact number of cores could be higher than that of PBA as no profiling information is available to identify it exploiting the maximum parallelism. To find an efficient allocation, we try to balance load across the used cores. Every task of the job is allocated to a core such that the processing load is balanced over the cores. In case the number of tasks in the job is higher than the number of cores, the approach allocates highly communicating tasks on the same core to reduce the

Algorithm 7.2 Non-profiling Based Resource Allocation

```
   Input: Incoming Jobs with arrival times, Value curves of Jobs,
   HPC Platform $HP$.
   Output: Resource Allocation for Incoming Jobs.
1  Steps 1 to 6 of Algorithm 7.1;
2  if JobQueue contains job(s) having positive values then
3      for count = 0 to JobQueue.size() do
4          Collect $bids$ from all nodes and select $maxBid$;
5          if $maxBid > 0$ then
6              Compute $values$ of unscheduled jobs by finding allocations on
                 $maxBid$ cores;
7              Select $maxValuableJob$, its $allocation$ and respective
                 $value$;
8              Identify voltage/frequency levels of used cores in the
                 $allocation$ to execute allocated tasks to optimize value and
                 energy;
9              Schedule $maxValuableJob$ on node having $maxBid$ cores
                 by following the $allocation$ to perform execution at found
                 voltage/frequency levels;
10             Update platform resources;
11         end
12     end
13 end
```

communication overhead. These considerations can lead to minimal response time and thus completion time of the job, resulting in maximum value. After finding the allocation, the value is computed as the value in the corresponding value curve at the completion time by taking the arrival time into account. Similarly, value achieved by each job is computed.

From all the jobs, the one leading to the maximum value, i.e., $maxValuableJob$, corresponding $allocation$ and $value$ is selected (line 7). Then, voltage/frequency levels are identified in step *ii)* as described subsequently.

Step *ii)* follows Algorithm 7.3, which takes the set of voltage scaling (VS) levels V available for cores as input and identifies the VS levels to be applied on cores to execute allocated tasks. For each task t, available VS levels are applied, and response time and value of the job at its completion is computed. From here onwards, applying voltage scaling on a task implies applying voltage scaling on the allocated core for the task. Similarly, VS level of a task implies VS level of the allocated core to execute the task. The value at completion is estimated by looking into the corresponding value curve while taking the arrival time of the job into account. If an applied VS on a task is valuable ($value_{job\_completition} > 0$), then total energy consumption of the job is calculated from Equation (7.1). Next, value at per unit of energy

Algorithm 7.3 Voltage/frequency Identification

```
Input: $V = \{v_i | \forall i \in [1, \cdots, n]\}$.
Output: VS levels of tasks.
repeat
    for each task t whose VS level is not fixed do
        for each VS level $v_i$ do
            Apply VS $v_i$ on $t$, and compute $response\_time$
            and $value_{job\_completition}$;
            if $value_{job\_completition} > 0$ then
                Calculate total energy consumption $E_{total}$
                (by Equation 7.1) when applying $v_i$ on $t$;
                $ValPerUnitEnerg = \frac{value_{job\_completition}}{E_{total}}$;
            end
        end
    end
    Find task $t_f$ & VS level $v_f$ corresponding to maximum
      $ValPerUnitEnerg$;
    Fix voltage of $t_f$ to $v_f$;
until VS levels of all tasks are not fixed;
```

consumption ($ValPerUnitEnerg$) is computed. Thereafter, the task and its VS level corresponding to maximum $ValPerUnitEnerg$ is found to fix the voltage level to execute the task. The same process is repeated to find VS levels of other tasks. Once voltage/frequency levels are identified, the $maxValuableJob$ is scheduled on the node having $maxBid$ cores based on the $allocation$ to perform execution at the identified voltage/frequency levels (Algorithm 7.2).

7.3 Evaluations

The proposed value and energy optimizing resource allocation approaches have been implemented in a C++ prototype and integrated with a SystemC functional simulator. As a workload, job models from historical data of an industrial HPC system at High Performance Computing Center Stuttgart (HLRS) are considered. The jobs in the workload have varying arrival time. It is considered that higher numbers of jobs arrive in peak times as compared to off-peak times. To sufficiently stress the platform, we consider all the jobs arriving over a day, i.e., 24-hour period. Each job contains a set of tasks having predefined connections (edges) amongst them that determines dependencies. For each task, the worst-case execution time (WCET) is known a priori and specified in the job model. The number of tasks in the jobs varies from 5 to 10. Further, it is assumed that the value curve of each job is given.

To evaluate our approaches under different load conditions, we conducted experiments with varied arrival rates of jobs while keeping higher number of arrivals during peak times over off-peak times. We have considered low, moderate and high arrival rates, where jobs arrive in the orders of a few seconds, dozens of seconds and minutes, respectively. It is assured that the total number of jobs for different arrival rates remains the same as the number of jobs considered for 24 hours.

To evaluate our approaches for different number of available servers (nodes), varying number of nodes are considered in the HPC platform. Further, the number of cores at each node is also varied to evaluate the approaches for assorted chip manufacturing technologies, where different number of cores can be integrated within a physical chip. The number of cores is varied such that it covers a broad spectrum of technologies including advanced servers to be available in future. The platform cores are assumed as the cores of Intel Core M processor, which supports 6 voltage/frequency levels of operation. However, any other type of core and higher number of voltage/frequency levels can be considered.

The main evaluated performance metrics are value and energy consumption, which are overall value achieved by executing the arrived jobs and energy consumed by the platform cores to execute the jobs, respectively. We also evaluate the percentage of rejected jobs that are removed from the job queue as their value becomes zero before the resources become available to allocate them. The rejected jobs also include jobs achieving zero value after their execution, which can be prevented by employing proper admission control and schedulability analysis.

7.3.1 Experimental Baselines

There are algorithms reported in the literature that apply DVFS to execute jobs. However, most of them optimize either only for [141] or energy [124], and both value and energy optimizing approaches do not consider jobs containing dependent tasks [66].

We compare results obtained from our approaches (PBA and NBA) to those of [141] and [124]. These approaches are considered for comparison as they can be applied to jobs containing dependent tasks and DVFS can be applied. In [141], the optimization is performed to optimize only value, i.e., no DVFS is applied, and the cores are assumed to operate at the highest supported voltage level. This approach chooses the maximum value job first to optimize the overall value and has been referred to as *ValOpt*. It helps to recognize energy savings by all the approaches applying DVFS. To employ this approach, the voltage/frequency identification step (line 8, in Algorithm 7.2) has been removed.

The approach of [124] identifies voltage/frequency levels of cores to execute the tasks scheduled on them in order to optimize only energy consumption. Therefore, it has been extended to optimize both the value and energy for a fair comparison. To employ this approach, the greedy algorithm of [124] is called for voltage/frequency identification in Algorithm 7.2 (line 8). In this algorithm, all the tasks scheduled on a core execute on a fixed identified voltage/frequency level, referred to as fixing cores power states (FCPS), as shown in example Figure 7.4. The voltage/frequency identification follows a greedy heuristic, where voltages of cores are fixed one by one during consecutive iterations. When employing voltage/frequency identification of [124], the approach is referred to as NBA-FCPS. Our approach identifies voltage/frequency levels of tasks in the similar manner, where tasks scheduled on a core can be executed on different voltages, referred to as fixing tasks power states (FTPS), as shown in example Figure 7.4. In this case, our NBA approach has been referred to as NBA-FTPS. It should also be noted that the run-time computation overhead of NBA approach has been considered to capture accurate achieved value after the job completion.

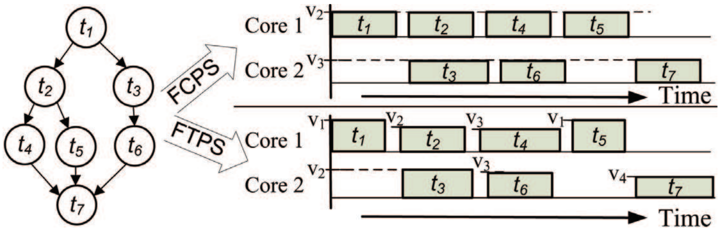


Figure 7.4 Voltage/frequency identification by FCPS and FTPS.

7.3.2 Value and Energy Consumption at Different Arrival Rates

Figure 7.5 shows the overall value and energy consumption when various approaches are employed for different arrival rates of jobs. A high arrival rate indicates that the jobs arrive quite frequently, whereas less frequently in low arrival rate. The value and energy estimates are normalized with respect to (w.r.t.) the value and energy by ValOpt approach at high arrival rate. The shown results have been computed for 3 nodes, where each node contains 8 cores. A couple of observations can be made from the figure. *1)* The value obtained by all the approaches increases from high to low arrival rates as more jobs are processed before their value becomes zero due to late availability of cores. *2)* The value obtained by PBA approach is always higher than that of other approaches due to joint optimization effect. On an average, PBA achieves 5.6% higher value than that of ValOpt. However, the joint optimization also leads to higher energy consumption when jobs arrival rate is not high. For the sake of both value and energy optimization, PBA is recommended to be employed.

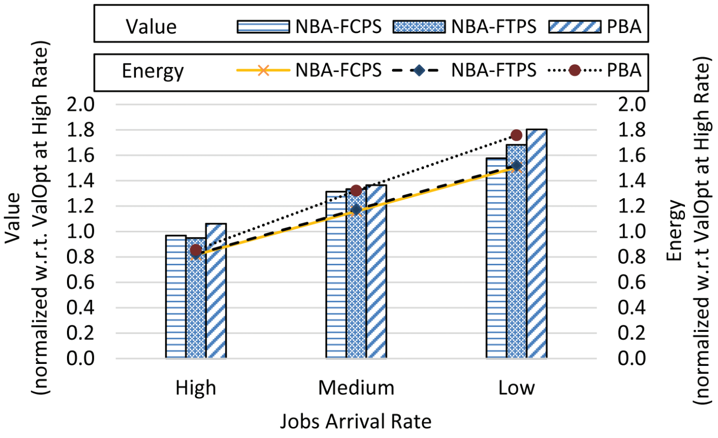


Figure 7.5 Value and energy at different arrival rates.

3) The energy consumption by NBA-FCTS and NB-FTPS is close to each other and lower than that of ValOpt. On an average, NBA-FCTS and PBA reduce energy consumption by 15.8% and 5.8%, respectively, when compared to ValOpt. Therefore, for the sake of both value and energy optimization, PBA is recommended to be employed.

7.3.3 Value and Energy Consumption with Varying Number of Nodes

Figure 7.6 shows the influence of the number of nodes (servers) on the overall value and energy consumption. At each node, a total of 8 cores are considered. The shown results are for high arrival rate of the jobs. The value and energy results are normalized w.r.t. the value and energy by ValOpt approach at 2 nodes. It can be observed that the overall value by all the approaches increases with the number of nodes due to increased processing capability leading to completion of higher number of jobs before their value becomes zero. It can also be observed that PBA achieves higher overall value than other approaches. Further, on an average, PBA performs better than other approaches if both the value and energy metrics are jointly evaluated as value divided by energy.

7.3.4 Value and Energy Consumption with Varying Number of Cores in Each Node

Figure 7.7 shows the overall value and energy consumption when number of cores at each node are varied for a total of 3 considered nodes. The jobs arriving at high rate are considered. The value and energy results are normalized w.r.t.

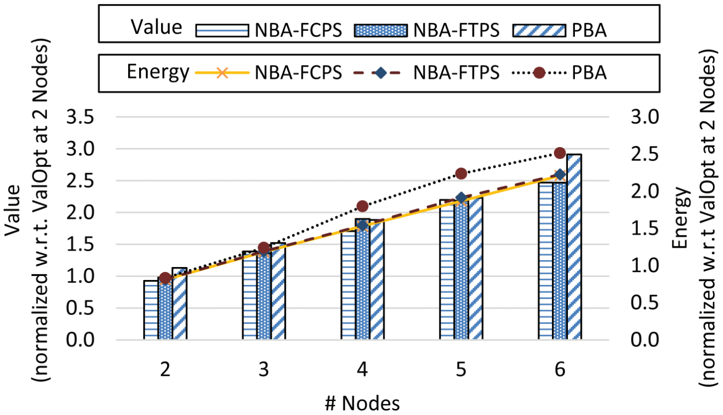


Figure 7.6 Value and energy with varying number of nodes.

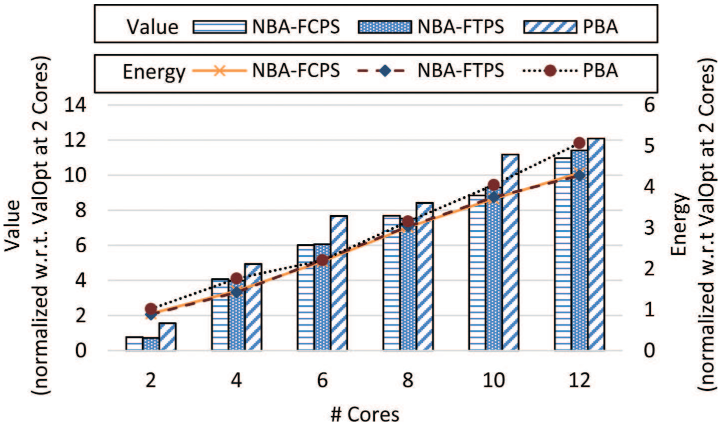


Figure 7.7 Value and energy with varying number of cores at each node.

the value and energy by ValOpt approach at 2 cores. A couple of observations can be made from the figure. First, the value by all the approaches increases with the number of cores due to increased processing capability leading to completion of higher number of jobs before their value becomes zero. Second, PBA achieves higher overall value than other approaches and better results when both the value and energy need to be considered. Third, in case profiling of jobs is not possible, i.e., PBA cannot be applied, NBA-FTPS can be employed to achieve a better trade-off between value and energy over other approaches.

7.3.5 Percentage of Rejected Jobs

Table 7.1 shows the rejected jobs (%) at different arrival rates when various approaches are employed. The average over different arrival rates is also shown for all the approaches. The tabulated results have been computed by considering 3 nodes, where each node contains 8 cores. It can be observed that, on an average, our proposed approaches NBA-FTPS and PBA reject lesser number of jobs as compared to baseline approaches. The PBA has the lowest rejection of jobs as each job is allocated on the exact number of cores

Table 7.1 Percentage of rejected jobs at different arrival rates

	ValOpt	NBA-FCPS	NBA-FTPS	PBA
High	49.0%	49.4%	48.8%	46.8%
Medium	29.2%	30.8%	30.0%	22.0%
Low	13.0%	12.8%	12.2%	00.0%
Average	30.4%	31.0%	30.3%	22.9%

exploiting all the potential parallelism with the help of design-time profiled results. This result in cores availability for higher number of jobs before their value become zero and thus lowers rejections. It should be noted that rejection rate by PBA for low arrival rate is not always zero and varies with number of cores/nodes.

7.4 Related Works

The dynamic resource allocation process usually employs a heuristic following some fundamental optimization procedure (e.g., incremental dynamic allocation) to identify an efficient allocation for each job at run-time. Several heuristics have been proposed to accomplish this aim [126]. These heuristics optimize one or several performance metrics, e.g., response time and energy consumption. In overload situation, these heuristics can lead to starvation, missed deadlines, and reduced throughput. Further, these heuristics do not take into account any notion of values of jobs to users and thus they do not optimize the overall value achieved by executing different jobs.

Market-inspired resource allocation heuristics are proven to provide promising results in the overload situation that is normally encountered in HPC system [156]. The heuristics employ notion of values of jobs, where values represent importance levels. Some researchers assume fixed value of a job [141], whereas others consider values that can change with time, described with so-called value curve of the job [25, 66]. In such curve, the value of a job normally decreases with computation time and reflects the importance level over the business process.

Market-inspired heuristics allocate jobs in several ways. For example, the highest value job is chosen first [141]. This approach might lead to small amount of available resources if a high value job requires a large amount of resources. To overcome above problem, the job having maximum value density can be chosen first [79], where the value density is computed as value divided by the amount of required computational resources Another heuristic to choose the job having minimum remaining value first is also proposed [24]. The remaining value is calculated as the area under the value curve from the current time to the time when its value is zero. These heuristics try to optimize overall value, but they do not consider energy consumption optimization. Further, they do not consider DVFS capable cores, which provide opportunities to reduce energy consumption.

Energy optimization approaches for HPC data centers have focused mainly on virtual machines (VMs) consolidation and DVFS. In consolidation, VMs with low utilization are placed together on a single host so that other used hosts can be freed to shut them down [13, 132, 148]. DVFS based approaches have been explored to reduce energy consumption is several areas, e.g., clusters [114, 149], web servers [142] and HPC data centers [28]. The approaches for HPC data centers (e.g., [28]) do not consider jobs containing dependent tasks. For other application domains, DVFS techniques for dependent tasks are explored (e.g., [124]), but optimization is not performed for value.

Some heuristics considering DVFS and optimizing both the value and energy consumption are reported in [66]. However, they consider independent tasks or jobs containing independent tasks. There are some additional multi-criteria optimization approaches, but they perform static resource allocation [50, 105]. Further, in dynamic resource allocation process, they do not use design-time profiling results, which can provide optimized value and energy. In contrast, the reported profiling and non-profiling based dynamic resource allocation approaches in this chapter consider jobs containing dependent tasks and jointly optimizes for both value and energy while applying DVFS.

7.5 Summary

This chapter proposed value and energy optimizing resource allocation approaches for HPC data centers. It has been shown that the approaches combine identification of efficient allocation and appropriate voltage/frequency levels to jointly optimize value and energy consumption. Whilst existing approaches focus on methods like server consolidation and DVFS, they do not consider jobs containing dependent tasks. It has been shown that the proposed approach is able to significantly reduce energy consumption and improve value while applying DVFS for jobs containing dependent tasks.

References

[1] Autosar: Automotive open system architecture. 2015.

[2] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems*, 23(3): 74–90, June 2003.

[3] A. Sharif Ahmadian, M. Hosseingholi, and A. Ejlali. Discrete feedback-based dynamic voltage scaling for safety critical real-time systems. *Scientia Iranica*, 20(3):647–656, 2013.

[4] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. In *Design Automation Conference*, 2008.

[5] Karl-Erik Arzen, Anders Robertsson, Dan Henriksson, Mikael Johansson, Håkan Hjalmarsson, and Karl Henrik Johansson. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Rev.*, 3(3):11–20, July 2006.

[6] Giuseppe Ascia, Vicenzo Catania, and Maurizio Palesi. A multi-objective genetic approach to mapping problem on network-on-chip. *Journal of Universal Computer Science*, 12(4):370–394, 2006.

[7] K. Astrom and T. Hagglund. *PID Controllers: Theory, Design and Tuning*. EDS Publications Ltd., 2nd edition, 1995.

[8] K. J. Astrom and T. Hagglund. Revisiting the Ziegler Nichols step response method for PID control. *Journal of Process Control*, 14(6): 635–650, 2004.

[9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993.

[10] R. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. Muller. Performance prediction in a grid environment. In *Grid Computing*, pp. 257–264, 2004.

[11] S. Banachowski, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pp. 139–150, Dec 2004.

[12] Anton Beloglazov and Rajkumar Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *2010 10th IEEE/ACM*

International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 577–578, May 2010.

[13] Anton Beloglazov and Rajkumar Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *Concurr. Comput.: Pract. Exper.*, 24(13):1397–1420, 2012.

[14] G. Beltrame, L. Fossati, and D. Sciuto. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(7): 1083–1095, July 2010.

[15] Luca Benini, Davide Bertozzi, and Michela Milano. Resource management policy handling multiple use-cases in mpsoc platforms using constraint programming. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP'08, pp. 470–484, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS'02, p. 279–, Washington, DC, USA, 2002. IEEE Computer Society.

[17] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *10th IFIP/IEEE International Symposium on Integrated Network Management, 2007. IM'07*, pp. 119–128, 2007.

[18] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, 30(3):207–214, 1981.

[19] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Sys. Arch.*, 50:105–128, 2004.

[20] E. W. Briao, D. Barcelos, F. Wronski, and F. R. Wagner. Impact of task migration in NoC-based mpsocs for soft real-time applications. In *2007 IFIP International Conference on Very Large Scale Integration*, pp. 296–299, Oct 2007.

[21] Uwe Brinkschulte, Mathias Pacher, and Alexander Von Renteln. Towards an artificial hormone system for self-organizing real-time task allocation. In *Software Technologies for Embedded and Ubiquitous Sys.*, pp. 339–347. Springer, 2007.

[22] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. *Future Gener. Comput. Syst.*, 29(8):2009–2025, October 2013.

[23] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A characterisation of the workload on an engineering design grid. In *Proceedings of the High Performance Computing Symposium*, HPC' 14, pp. 8:1–8: 8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[24] Andrew Marc Burkimsher. *Fair, Responsive Scheduling of Engineering Workflows on Computing Grids*. PhD thesis, UK, 2014.

[25] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The Meaning and Role of Value in Scheduling Flexible Real-time Systems. *J. Syst. Archit.*, 46(4):305–325, 2000.

[26] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1):7–24, 2002.

[27] Rajkumar Buyya and Manzur Murshed. A deadline and budget constrained cost-time optimisation algorithm for scheduling task farming applications on global grids. *arXiv:cs/0203020*, March 2002. Technical Report, Monash University, March 2002.

[28] Rodrigo N. Calheiros and Rajkumar Buyya. Energy-Efficient Scheduling of Urgent Bag-of-Tasks Applications in Clouds Through DVFS. In *Proceedings of IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM)*, pp. 342–349, 2014.

[29] I. Caliskanelli and L. S. Indrusiak. Search-Based Parameter Tuning on Application-Level Load Balancing for Distributed Embedded Systems. In *IEEE Int. Conf. on High Perf. Comp. and Comms. on Embedded and Ubiquitous Computing (HPCC_EUC)*, 2013.

[30] Ipek Caliskanelli, Leandro Soares Indrusiak, Fiona Polack, James Harbin, Paul Mitchell, and David Chesmore. Bio-inspired load balancing in large-scale WSNs using pheromone signalling. *Int. Journal of Distributed Sensor Networks*, 2013.

[31] Salvatore Carta, Andrea Alimonda, Alessandro Pisano, Andrea Acquaviva, and Luca Benini. A control theoretic approach to energy-efficient pipelined computation in mpsocs. *ACM Trans. Embed. Comput. Syst.*, 6(4), September 2007.

[32] E. Carvalho, C. Marcon, N. Calazans, and F. Moraes. Evaluation of static and dynamic task mapping algorithms in NoC-based mpsocs. pp. 087–090, Oct. 2009.

[33] G. Castilhos, M. Mandelli, G. Madalozzo, and F. Moraes. Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *IEEE Computer Society Annual Symp. on VLSI*, 2013.

[34] Ken Chen and Paul Muhlethaler. A Scheduling Algorithm for Tasks Described by Time Value Function. *Real-Time Syst.*, 10(3):293–312, 1996.

[35] Razvan Cheveresan, Matt Ramsay, Chris Feucht, and Ilya Sharapov. Characteristics of workloads used in high performance and technical computing. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS'07, pp. 73–82, New York, NY, USA, 2007. ACM.

[36] C.-L. Chou and R. Marculescu. Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(1):78–91, Jan. 2010.

[37] Chen-Ling Chou and R. Marculescu. Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels. pp. 161–166, 30 2007-Oct. 3 2007.

[38] Chen-Ling Chou and R. Marculescu. User-aware dynamic task allocation in networks-on-chip. pp. 1232–1237, March 2008.

[39] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Energy-Aware Communication and Remapping of Tasks for Reliable Multimedia Multiprocessor Systems. In *ICPADS*, 2012.

[40] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pp. 10 pp.–398, Dec 2005.

[41] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[42] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[43] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes. Dynamic task mapping for MPSoCs. *IEEE Design Test of Computers*, 27:26–35, 2010.

[44] Yixin Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pp. 219–234, 2002.

[45] S. Djosic and M. Jevtic. Dynamic voltage scaling for real-time systems under fault tolerance constraints. In *Microelectronics (MIEL), 2012 28th International Conference on*, pp. 375–378, May 2012.

[46] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the Impact of Video Quality on User Engagement. In *SIGCOMM Conf.*, SIGCOMM'11. ACM, 2011.

[47] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[48] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4(4):437–455, 2003.

[49] Stijn Eyerman and Lieven Eeckhout. Fine-grained DVFS Using On-chip Regulators. *ACM Trans. Archit. Code Optim.*, 8(1):1:1–1:24, 2011.

[50] Hamid Mohammadi Fard, Radu Prodan, Juan Jose Durillo Barrionuevo, and Thomas Fahringer. A multi-objective approach for workflow scheduling in heterogeneous environments. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 300–309, 2012.

[51] Dror Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. 2013.

[52] J. D. Gelas. Dynamic power management: A quantitative approach. http://www.anandtech.com/show/2919, 2010. Accessed: 2016-06-05.

[53] H. A. Ghazzawi, I. Bate, and L. S. Indrusiak. A control theoretic approach for workflow management. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pp. 280–289, July 2012.

[54] H. A. Ghazzawi, I. Bate, and L. S. Indrusiak. A control theoretic approach for workflow management. In *2012 17th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 280 –289, July 2012.

[55] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[56] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT'13, pp. 17:1–17:15, Piscataway, NJ, USA, 2013. IEEE Press.

[57] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[58] Philip K. F. Hölzenspies, Johann L. Hurink, Jan Kuper, and Gerard J. M. Smit. Run-time spatial mapping of streaming applications to a

heterogeneous multi-processor system-on-chip (mpsoc). pp. 212–217, 2008.

[59] S. Hong, T. Chantem, and X. S. Hu. Meeting end-to-end deadlines through distributed local deadline assignments. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 183–192, Nov 2011.

[60] Jingcao Hu and R. Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. pp. 233–239, jan. 2003.

[61] Leandro Soares Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture*, 60(7): 553–561, 2014.

[62] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing Risk and Reward in a Market-Based Task Service. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 160–169, 2004.

[63] D. Isovic, G. Fohler, and L. Steffens. Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions. In *Euromicro Conference on Real-Time Sys.*, 2003.

[64] P. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, 2013.

[65] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Trans. on Parallel and Distributed Sys.*, 8:1268–1274, 1997.

[66] Bhavesh Khemka, Ryan Friese, Sudeep Pasricha, Anthony A Maciejewski, Howard Jay Siegel, Gregory A Koenig, Sarah Powers, Marcia Hilton, Rajendra Rambharos, and Steve Poole. Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system. *Sustainable Computing: Informatics and Systems*, 5:14–30, 2015.

[67] Abbas Eslami Kiasari, Axel Jantsch, and Zhonghai Lu. Mathematical formalisms for performance evaluation of networks-on-chip. *ACM Comput. Surv.*, 45(3):38:1–38:41, July 2013.

[68] Wonyoung Kim, M. S. Gupta, G. Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 123–134, Feb 2008.

[69] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: distributed resource management for on-chip many-core systems. In *Int. Conf. on Hardware/software codesign and sys. synthesis (CODES+ISSS)*, 2011.

[70] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[71] H. Kopetz. The time-triggered model of computation. In, *The 19th IEEE Real-Time Systems Symposium, 1998. Proceedings*, pp. 168 –177, December 1998.

[72] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, ESTMED'06, pp. 33–38, Washington, DC, USA, 2006. IEEE Computer Society.

[73] Yu-Kwong Kwok, Anthony A. Maciejewski, Howard Jay Siegel, Ishfaq Ahmad, and Arif Ghafoor. A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 66(1):77–98, January 2006.

[74] C. Lee, S. Kim, and S. Ha. Efficient run-time resource management of a manycore accelerator for stream-based applications. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pp. 51–60, Oct 2013.

[75] Yann-Hang Lee, Daeyoung Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pp. 14–23, 2000.

[76] C. Li and L. Li. Multi-level scheduling for global optimization in grid computing. *Computers & Electrical Engineering*, 34(3):202–221, 2008.

[77] Bin Lin, Arindam Mallik, Peter A. Dinda, Gokhan Memik, and Robert P. Dick. Power reduction through measurement and modeling of users and cpus: Summary. *SIGMETRICS Perform. Eval. Rev.*, 35(1): 363–364, June 2007.

[78] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[79] Carey Douglass Locke. *Best-effort Decision-making for Real-time Scheduling*. PhD thesis, Pittsburgh, PA, USA, 1986. AAI8702895.

[80] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pp. 56–67, 1999.

[81] Chenyang Lu, John A. Stankovic, Tarek F. Abdelzaher, Gang Tao, Sang H. Son, and Michael Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS'10, pp. 13–23, Washington, DC, USA, 2000. IEEE Computer Society.

[82] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Syst.*, 23(1/2):85–126, July 2002.

[83] Chenyang Lu, Xiaorui Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 16(6):550–561, June 2005.

[84] Chenyang Lu, Xiaorui Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, June 2005.

[85] Shih-Shen Lu, Chun-Hsien Lu, and Pao-Ann Hsiung. Congestion- and energy-aware run-time mapping for tile-based network-on-chip architecture. pp. 300–305, Aug. 2010.

[86] Bertram Ludscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[87] M. Mandelli, L. Ost, E. Carara, G. Guindani, T. Gouvea, G. Medeiros, and F. G. Moraes. Energy-aware dynamic task mapping for NoC-based MPSoCs. In *2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1676–1679, May 2011.

[88] C. A. M. Marcon, E. I. Moreno, N. L. V. Calazans, and F. G. Moraes. Comparison of network-on-chip mapping algorithms targeting low energy consumption. *Computers Digital Techniques, IET*, 2(6):471–482, November 2008.

[89] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 196–201, March 2010.

[90] Andrew Stephen McGough, Ali Afzal, John Darlington, Nathalie Furmento, Anthony Mayer, and Laurie Young. Making the grid predictable through reservations and performance modelling. *Comput. J.*, 48(3):358–368, May 2005.

[91] H. R. Mendis, L. S. Indrusiak, and N. C. Audsley. Bio-inspired distributed task remapping for multiple video stream decoding on homogeneous NoCs. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pp. 1–10, October 2015.

[92] Hashan R. Mendis, Leandro Soares Indrusiak, and Neil C. Audsley. Predictability and utilisation trade-off in the dynamic management

of multiple video stream decoding on network-on-chip based homogeneous embedded multi-cores. In *Proc. of the 22nd Int. Conf. on Real-Time Networks and Sys.*, 2014.

[93] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3): 241–299, September 2000.

[94] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource software on multicore automotive ecus - combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, Oct 2012.

[95] O. Moreira, J. D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, pp. 332–341, March 2005.

[96] Orlando Moreira, Frederico Valente, and Marco Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT'07, pp. 57–66, New York, NY, USA, 2007. ACM.

[97] Pierre-André Mudry and Gianluca Tempesti. Self-scaling stream processing: A bio-inspired approach to resource allocation through dynamic task replication. In *Adaptive Hardware and Systems, NASA/ ESA Conference on*. IEEE, 2009.

[98] P. Munk, B. Saballus, J. Richling, and H. U. Heiss. Position paper: Real-time task migration on many-core processors. In *Architecture of Computing Systems. Proceedings, ARCS 2015 – The 28th International Conference on*, pp. 1–4, March 2015.

[99] M. Di Natale and A. L. Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, April 2010.

[100] Borislav Nikolić, Hazem Ismail Ali, Stefan M. Petters, and Luís Miguel Pinho. Are virtual channels the bottleneck of priority-aware wormhole-switched NoC-based many-cores? In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pp. 13–22, New York, NY, USA, 2013. ACM.

[101] A.-C. Orgerie, L. Lefevre, and J.-P. Gelas. Chasing gaps between bursts: Towards energy efficient large scale experimental grids. In *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008*, pp. 381–389, December 2008.

[102] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present, and future. *Linux Symposium*, 2:223–238, 2006.

[103] S. Pandey, L. Wu, S. M. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 400–407, April 2010.

[104] J. Park, J. Harnisch, M. Deubzer, and K. Jeong et al. Mode-dynamic task allocation and scheduling for an engine management real-time system using a multicore microcontroller. *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.*, 7(1):133–140, 2014.

[105] Ilia Pietri, Maciej Malawski, Gideon Juve, Ewa Deelman, Jarek Nabrzyski, and Rizos Sakellariou. Energy-constrained provisioning for scientific workflow ensembles. In *IEEE International Conference on Cloud and Green Computing (CGC)*, pp. 34–41, 2013.

[106] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 35(5):89–102, October 2001.

[107] R. Piscitelli and A. D. Pimentel. Design space pruning through hybrid analysis in system-level design space exploration. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 781–786, March 2012.

[108] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[109] Wei Quan and Andy D. Pimentel. Exploring task mappings on heterogeneous mpsocs using a bias-elitist genetic algorithm. *CoRR*,abs/14 06.7539, 2014.

[110] A. Racu and L. S. Indrusiak. Using genetic algorithms to map hard real-time NoC-based systems. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2012.

[111] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, January 2012.

[112] I. Rodero, J. Jaramillo, A. Quiroz, M. Parashar, F. Guim, and S. Poole. Energy-efficient application-aware online provisioning for virtualized clouds and data centers. In *International Green Computing Conference (IGCC)*, pp. 31–45, 2010.

[113] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *, The 18th IEEE Real-Time Systems Symposium, 1997. Proceedings*, pp. 320–329, December 1997.

[114] Xiaojun Ruan, Xiao Qin, Ziliang Zong, K. Bellam, and M. Nijim. An Energy-Efficient Scheduling Algorithm Using Dynamic Voltage Scaling for Parallel Applications on Clusters. In *Proceedings of International Conference on Computer Communications and Networks (ICCCN)*, pp. 735–740, 2007.

[115] P. K. Saraswat, P. Pop, and J. Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 89–98, April 2010.

[116] M. N. S. M. Sayuti and L. S. Indrusiak. Real-time low-power task mapping in networks-on-chip. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 14–19, Aug 2013.

[117] M. N. S. M. Sayuti and L. S. Indrusiak. Real-time low-power task mapping in Networks-on-Chip. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 14–19, 2013.

[118] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'12, pp. 71–80, New York, NY, USA, 2012. ACM.

[119] A. Schranzhofer, J. J. Chen, and L. Thiele. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *IEEE Transactions on Industrial Informatics*, 6(4):692–707, Nov 2010.

[120] A. Schranzhofer, Jian-Jian Chen, and L. Thiele. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *Industrial Informatics, IEEE Transactions on*, 6(4):692–707, Nov. 2010.

[121] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2–3):101–155, November 2004.

[122] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pp. 161–170, April 2008.

[123] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–10, May 2013.

[124] Amit Kumar Singh, Anup Das, and Akash Kumar. Energy Optimization by Exploiting Execution Slacks in Streaming Applications on Multiprocessor Systems. In *Proceedings of ACM Design Automation Conference (DAC)*, pp. 115:1–115:7, 2013.

[125] Amit Kumar Singh, Piotr Dziurzanski, and Leandro Soares Indrusiak. Market-inspired Dynamic Resource Allocation in Many-core High Performance Computing Systems. In *IEEE International Conference on High Performance Computing & Simulation (HPCS)*, pp. 413–420, 2015.

[126] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of ACM Design Automation Conference (DAC)*, pp. 1:1–1:10, 2013.

[127] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, Jörg Henkel, Anup Das, Wu Jigang, Thambipillai Srikanthan, Samarth Kaushik, Yajun Ha, and Alok Prakash. Mapping on multi/many-core systems: survey of current and emerging trends. In *Design Automation Conference*, 2013.

[128] Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. Communication-aware heuristics for run-time task mapping on NoC-based mpsoc platforms. *J. Syst. Archit.*, 56(7):242–255, July 2010.

[129] L. T. Smit, J. L. Hurink, and G. J. M. Smit. Run-time mapping of applications to a heterogeneous soc. pp. 78 –81, Nov. 2005.

[130] O. O. Sonmez and A. Gursoy. A novel economic-based scheduling heuristic for computational grids. *International Journal of High Performance Computing Applications*, 21(1):21–29, 2007.

[131] Ranjani Sridharan and Rabi Mahapatra. Reliability aware power management for dual-processor real-time embedded systems. In *Proceedings of the 47th Design Automation Conference*, DAC'10, pp. 819–824, New York, NY, USA, 2010. ACM.

[132] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of Conference on Power Aware Computing and Systems (HotPower)*, pp. 10–10, 2008.

[133] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2009.

[134] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2009.

[135] J. A. Stankovic, Chenyang Lu, S. H. Son, and Gang Tao. The case for feedback control real-time scheduling. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pp. 11–20, 1999.

[136] S. Stuijk, T. Basten, M.C.W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th ACM/IEEE Design Automation Conference, 2007. DAC'07*, pp. 777–782, June 2007.

[137] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 404–411, July 2011.

[138] Y. Takeuchi, Y. Nakata, H. Kawaguchi, and M. Yoshimoto. Scalable parallel processing for H.264 encoding application to multi/many-core processor. In *Int. Conf. on Intelligent Control and Information Processing (ICICIP)*, August 2010.

[139] Y. Tao and X. Yu. Classified optimization scheduling algorithm driven by multi-qos attributes in economical grid. In *International Conference on Computer Science and Software Engineering*, volume 3, pp. 70–73. IEEE, 2008.

[140] M. K. Tavana, M. Salehi, and A. Ejlali. Feedback-based energy management in a standby-sparing scheme for hard real-time systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 349–356, Nov 2011.

[141] Theocharis Theocharides, Maria K. Michael, Marios Polycarpou, and Ajit Dingankar. Hardware-enabled Dynamic Resource Allocation for Manycore Systems Using Bidding-based System Feedback. *EURASIP J. Embedded Syst.*, 2010:3:1–3:21, 2010.

[142] Y. Tian, C. Lin, Z. Chen, J. Wan, and X. Peng. Performance evaluation and dynamic optimization of speed scaling on web servers in cloud computing. *Tsinghua Science and Technology*, pp. 298–307, 2013.

[143] Ken Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, Dept. of Computer Science, University of York, January 1994.

[144] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, March 2002.

[145] Wolfgang Trumler, Tobias Thiemann, and Theo Ungerer. An artificial hormone system for self-organization of networked nodes. In *Biologically Inspired Cooperative Computing*. Springer, 2006.

[146] J. R. van Kampenhout. Deterministic task transfer in network-on-chip based multi-core processors. *Computer Engineering*, (18), 2011.

[147] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Bruce Jacob. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the 2003*

International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES'03, pp. 255–266, New York, NY, USA, 2003. ACM.

[148] G. von Laszewski, Lizhe Wang, A. J. Younge, and Xi He. Power-aware scheduling of virtual machines in DVFS-enabled clusters. In *Proceedings of IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*, pp. 1–10, 2009.

[149] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 368–377, 2010.

[150] M. Wieczorek, M. Siddiqui, A. Villazon, R. Prodan, and T. Fahringer. Applying advance reservation to increase predictability of workflow execution on the grid. In *Second IEEE International Conference on e-Science and Grid Computing, 2006. e-Science'06*, p. 82, December 2006.

[151] S. Wildermann, T. Ziermann, and J. Teich. Run time mapping of adaptive applications onto homogeneous NoC-based reconfigurable architectures. In *International Conference on Field-Programmable Technology, 2009. FPT 2009*, pp. 514–517, December 2009.

[152] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem & mdash; overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[153] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGOPS Oper. Syst. Rev.*, 38(5):248–259, October 2004.

[154] L. Xiao, Y. Zhu, L. M. Ni, and Z. Xu. Incentive-based scheduling for market-like computational grids. *Parallel and Distributed Systems, IEEE Transactions on*, 19(7):903–913, 2008.

[155] T. T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. pp. 524–529, 2002.

[156] Chee Shin Yeo and Rajkumar Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Softw. Pract. Exper.*, 36(13):1381–1419, 2006.

[157] Chee Shin Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software: Practice and Experience*, 36(13):1381–1419, 2006.

[158] D. Zhu and C. Qian. Challenges in future automobile control systems with multicore processors. In *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*, 2011.

[159] H. Zhu, S. Goddard, and M. B. Dwyer. Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 239–248, Nov 2011.

[160] Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Trans. Embed. Comput. Syst.*, 11(4):85:1–85:30, January 2013.

[161] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS'12, pp. 109–118, Washington, DC, USA, 2012. IEEE Computer Society.

[162] Yifan Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pp. 84–93, May 2004.

About the Authors

Leandro Soares Indrusiak is a faculty member in the Department of Computer Science at the University of York. He is part of the Real-Time Systems group, with technical contributions in the areas of systems and networks with timing and energy constraints. He has more than 100 peer-reviewed publications in the top conferences and journals in real-time and embedded systems, design automation and high-performance computing. Over the past decade, he has supervised and graduated seven doctoral students, has held visiting faculty positions in five different countries, and has led projects funded by the European Union, research councils of the UK, Germany and Brazil, as well as several companies. Dr Indrusiak holds a bi-national doctoral degree jointly issued by UFRGS (Brazil) and TU Darmstadt (Germany) in 2003, is a member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC), and a senior member of the IEEE.

Piotr Dziurzanski received the M.Sc. and Ph.D. degrees in computer science from West Pomeranian University of Technology (Poland) in 2000 and 2003, respectively. From 2003 to 2012 he worked as an assistant professor at the same university. In 2013 he joined the DreamCloud project as a research associate at the University of York. He currently works as a lecturer at the Staffordshire University (UK) in the Faculty of Computing, Engineering and Sciences. His scientific interests include embedded systems, hardware-software co-design, on-chip multiprocessor systems and distributed computing.

Amit Kumar Singh received the B.Tech. degree in Electronics Engineering from Indian Institute of Technology (Indian School of Mines), Dhanbad, India, in 2006. Thereafter, he worked with HCL Technologies, India before starting his Ph.D. at School of Computer Engineering, Nanyang Technological University (NTU), Singapore. He completed his Ph.D. in 2012. He worked as a post-doctoral researcher at National University of Singapore (NUS) from 2012 to 2014 and at University of York, UK from 2014 to 2016. Currently, he is working as senior research fellow at University of Southampton, UK. His research interests include system level design-time and run-time optimizations of 2D and 3D multi-core systems with focus on performance, energy, temperature, and reliability. He has published over 45 papers in these areas in leading international journals/conferences. He was the recepient of ISORC 2016 Best Paper Award, PDP 2015 Best Paper Award, HiPEAC Paper Award, and GLSVLSI 2014 Best Paper Candidate. He has served on the TPC of IEEE/ACM conferences like ISED, MES, NoCArc and ESTIMedia.

Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing

Leandro Soares Indrusiak, Piotr Dziurzanski and
Amit Kumar Singh

The availability of many-core computing platforms enables a wide variety of technical solutions for systems across the embedded, high-performance and cloud computing domains. However, large scale manycore systems are notoriously hard to optimise. Choices regarding resource allocation alone can account for wide variability in timeliness and energy dissipation (up to several orders of magnitude). *Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing* covers dynamic resource allocation heuristics for manycore systems, aiming to provide appropriate guarantees on performance and energy efficiency. It addresses different types of systems, aiming to harmonise the approaches to dynamic allocation across the complete spectrum between systems with little flexibility and strict real-time guarantees all the way to highly dynamic systems with soft performance requirements. Technical topics presented in the book include:

- Load and Resource Models
- Admission Control
- Feedback-based Allocation and Optimisation
- Search-based Allocation Heuristics
- Distributed Allocation based on Swarm Intelligence
- Value-Based Allocation

Each of the topics is illustrated with examples based on realistic computational platforms such as Network-on-Chip manycore processors, grids and private cloud environments.

ISBN 978-87-93519-08-4

9 788793 519084

River Publishers