IMPROVING ENERGY EFFICIENCY AND QUALITY-OF-CONTROL METRICS IN RELIABLE MULTIPROCESSOR REAL-TIME SYSTEMS

by

Abhishek Roy
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

Hakan Aydin
Dr. Hakan Aydin, Dissertation Director

Dr. Songqing Chen, Committee Member

P.H.Pathak
Dr. Parth Pathak, Committee Member

Qi Wei
Dr. Qi Wei, Committee Member

Dr. David Rosenblum, Department Chair

Dr. Kenneth Ball, Dean, Volgenau School of Engineering

Date: 04-26-2021

Spring Semester 2021
George Mason University
Fairfax, VA

Improving Energy Efficiency and Quality-of-Control Metrics in Reliable Multiprocessor Real-Time Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Abhishek Roy
Master of Science
George Mason University, 2018
Bachelor of Science
Bangladesh University of Engineering and Technology, 2009

Director: Dr. Hakan Aydin, Professor
Department of Computer Science

Spring Semester 2021
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my father, Bikash Chandra Roy, and my mother, Anjana Roy, for motivating me to start this journey, and my advisor, Dr. Hakan Aydin, for leading me to complete it.

Acknowledgments

I would like to express my deepest appreciation to the following people– the completion of my dissertation would not have been possible without their continuous ministration and active support.

First, I cannot begin to express my thanks to my advisor, Prof. Hakan Aydin. He has been an excellent mentor, guide and professor to me and I am deeply indebted to him. He always took good care of me, provided me with very helpful guidance when I got stuck in my research, and inspired me to keep going during the worst of my days. I have appreciated his outstanding personality over the years and I have learned a lot of technical and non-technical things from him which I would value for the rest of my life. I was extremely lucky to have him as my supervisor.

I would also like to express my deepest gratitude to my respected committee members, Dr. Songqing Chen, Dr. Parth Pathak and Dr. Qi Wei for investing their valuable time in my dissertation thesis. I have received valuable advice and guidance from them over the years which was instrumental in shaping my research work. Their constructive criticisms enriched this dissertation to a great extent. I also want to thank my research collaborator, Prof. Dakai Zhu, whose guidance was very crucial to bring this dissertation into fruition.

I am also very much grateful to Prof. Elizabeth White for being extremely kind to me over the years. She has always been very appreciative to my research and teaching abilities which was a great inspiration for me. She was kind enough to give me GTA appointments for a series of semesters and I could not have survived without those. I would also like to thank the CS department Chair, Prof. Rosenblum for scrutinizing my thesis and providing his invaluable suggestions to improve it. My sincere thanks should also go to the wonderful office staffs in the CS department: Michèle, Ryan and Cecelia, who have always been extraordinarily helpful and provided me with genuine support countless times throughout the entire PhD journey. I am also grateful to the Provost Office and the NSF for providing financial support.

Finally, I would like to acknowledge the help and support I received from a lot of other people, and it is impossible to name them all here. I would like to thank my lab-mates and colleagues in the CS department at Mason for having interesting research discussions with me from time to time which helped me generate new ideas. I am also much grateful to my friends and family who were always there for me to pick me up when I stumbled. A very special thanks goes to my housemates who have been extremely supportive for my PhD research throughout the years.

Table of Contents

List of Tables

List of Figures

Abstract

IMPROVING ENERGY EFFICIENCY AND QUALITY-OF-CONTROL METRICS IN RELIABLE MULTIPROCESSOR REAL-TIME SYSTEMS

Abhishek Roy, PhD

George Mason University, 2021

Dissertation Director: Dr. Hakan Aydin

Energy efficiency and reliability management are two important aspects of real-time embedded systems, in addition to strict guarantees for timing constraints. The objective of this research is to optimize performance metrics such as energy consumption and quality-of-control, by means of appropriately scheduling computational tasks and applying system-level energy management techniques. Reliability is achieved via run-time fault tolerance, which must be based on additional hardware and/or software components. These additional components are often redundant for normal operation and they can cause significant increase in overall energy consumption. Therefore, simultaneously managing both energy and reliability, two opposing factors, has been an intriguing research topic for real-time embedded systems.

In this dissertation, we first address the problem of minimizing energy consumption for the emerging heterogeneous multi-core systems, which have a wide range of power-performance characteristics. Reliability is achieved via scheduling two copies (primary and backup) of each task on different processing cores. Each processing core is equipped with Dynamic Voltage and Frequency Scaling and Dynamic Power Management features in order to reduce the energy consumption.

We address the problem for independent real-time tasks on a heterogeneous dual-core processor, and we propose several algorithms which take advantage of the heterogeneity to reduce energy consumption while ensuring hard deadlines with reliability guarantees. We also address the problem for tasks with precedence constraints and develop efficient solutions. Then, we tackle the problem for the general real-time periodic task model with arbitrary periods for the same heterogeneous platform. We propose techniques for task partitioning, priority assignment, and runtime scheduling of periodic tasks to achieve reliability and energy efficiency. Evaluation of the proposed schemes was conducted by extensive simulation experiments which show their effectiveness within a wide range of system parameters.

Finally, we tackle the problem of partitioning a set of real-time tasks on a homogeneous multiprocessor control system, and at the same time, assigning periods to those tasks (within an allowable range) in order to maximize the quality-of-control for the system. We model the quality-of-control as a concave function of task invocation rates (periods). We propose a family of heuristics that are based on effectively converting the multiprocessor problem to a single but faster uniprocessor system. We conduct extensive simulation experiments that demonstrate the superior performance of our proposed algorithms.

Chapter 1: Introduction

In real-time embedded systems, guaranteeing timing constraints is of utmost importance. The real-time operation is, in general, heavily coupled with the environment, and often it is the case that, a failure in the system may have very serious, even life-threatening consequences. Some examples of such systems include industrial control systems, autonomous driving systems, flight control systems and high-confidence medical equipments. Due to their safety-critical nature, these systems are often furnished with fault-tolerant mechanisms, by provisioning redundant hardware/software components. Moreover, in order to provide timing guarantees with high confidence, these systems are often designed to incorporate the worst-case scenarios, which may result in an under-provisioning of the system and inefficient use of resources during average-case operation. Therefore, the problem of implementing real-time systems with strict timing and reliability guarantees while using system resources appropriately is an active research area.

Multiprocessor systems have become the de-facto standard for most modern computing platforms, and real-time systems have also been designed which can take advantage of multiple parallel processing cores. A recent development in multicore platforms is the *heterogeneous* multicore processors in which, multiple processing cores, with dissimilar power-performance characteristics, are combined on to a single silicon chip. These heterogeneous architectures are often seen as a key to tackle an application with varying energy or performance requirements.

Recently, energy-efficient operation has become an important aspect for many computing systems, regardless of whether it is battery-powered or not, because of the associated energy cost and the negative impact on the environment. The two most common energy saving techniques available at the system-level are *Dynamic Voltage and Frequency Scaling* (DVFS) and *Dynamic Power Management* (DPM). DVFS saves energy by reducing the processing

(clock) speed of the core, and DPM can put the core in a low-power (sleep) state. In the last decade, numerous studies explored how to minimize energy consumption using DVFS and DPM. Another important aspect of any practical real-time system is fault tolerance, which means resiliency to various runtime faults. If a fault occurs, the system should be able to detect it, mask it, recover from it and re-execute the faulty task– all within a predetermined deadline. This is generally achieved by designing the systems with redundant components as necessary, which again, results in an increase in the energy and infrastructure cost.

Since both energy and fault tolerance are particularly critical factors, and since they work in opposing directions to each other, it has been a challenging research problem to address energy and reliability of a system in tandem. Many of the recent research efforts have investigated this problem in the context of homogeneous multiprocessor systems. Some solutions were developed that exploit only software redundancy, while some others provision additional hardware components in an energy-aware manner. One promising direction is the *energy-aware standby sparing systems* in which, each processing core is equipped with a dedicated spare core. Any task executing on the primary core will have a corresponding backup task on the spare core. Several techniques have been proposed to minimize the overall energy consumption via the application of DVFS and DPM on the primary and spare cores. Another approach is the *mixed primary backup* (MPB) framework in which, processing cores are not designated as primary or spare; instead, a processing core can schedule both primary and backup copies of tasks with the only constraint that the two copies of the same task must be placed on different processing cores. Both of the approaches can be used to build energy-efficient and resilient systems which can tolerate both transient and permanent faults, which are the two most common type of faults in real-time embedded systems [1].

Many real-time tasks are periodic in nature, and they appear in two main types. When all tasks in the set has a common period, it is called a *frame-based system* [2–4]. The other real-time task model is called *general periodic tasks*, where each task can have an arbitrary period [5]. A number of real-time applications are made up of tasks with *precedence*

constraints, which means, the various tasks in the system not only has a deadline, but also there can be data dependencies between one another. Precedence constraints can capture the relationship when a task can only start *after* some other tasks have finished execution and made their results (data) available. The precedence constraints among tasks can be modeled as a *directed acyclic graph (DAG)*. It is possible to make use of the MPB approach and allocate the primary and backup copies in such a way that is consistent with the data dependency, provides reliability, and minimizes the overall energy consumption. The MPB approach is also useful for general periodic tasks, where, for each task, a job is generated once in its period, and the job must be completed within a deadline (which is usually the period boundaries.) To achieve reliability with MPB, each job is provisioned with a backup copy on the alternate core, and DVFS and DPM can be used to minimize energy consumption. To further minimize the energy overhead, it is often very useful to cancel the backup job once the primary job completes without any error. Backup jobs can be delayed considering the deadline, which can help providing more room and applying DVFS on the primary jobs. We investigated a few techniques on this general idea using simulation studies which show very reasonable savings in energy consumptions, at the same time being tolerant to both transient and permanent faults.

Another major application area of real-time systems is the industrial control systems. In general, digital control systems are seen as much more capable, flexible and efficient than their analog counterparts. It is known that the assigned period (invocation frequency) of a control task directly affects the *quality-of-control*, and therefore, the period assignment problem for control task design has been extensively studied in the literature. With the advent of high-performance homogeneous multiprocessor systems, the period assignment problem has gained a new dimension because of the multiprocessor task allocation problem. While the multiprocessor task allocation problem, in general, falls in the class of NP-complete problems, researchers have been persistently searching for efficient heuristics. The problem becomes more interesting when both task allocation and period assignment problems are addressed at the same time.

1.1 Problem Statement

In this dissertation, we investigate two problems in real-time systems scheduling. The first problem has a relatively larger scope: the *energy-efficient scheduling of real-time tasks with fault tolerance requirement*. In this setting, we tackle the three interconnected dimensions of the general problem, namely, i.) strict timing guarantees, ii.) tolerance to runtime faults, and iii.) energy efficiency. We consider a set of real-time tasks with hard deadlines that are executed on a dual-core processor with tolerance to both transient and permanent faults while minimizing the overall energy consumption. To implement fault tolerance, we create a "backup" copy for each ("primary") task, and ensure that they are always scheduled on separate processing cores. The problem is to determine the task-to-processor allocation, scheduling and runtime speed setting of the tasks satisfying all the constraints and minimizing energy consumption. As the platform model, we consider the recently emerged *heterogeneous* multicore processors which are known to be energy efficient because one can exploit their wide range of power-performance characteristics based on the current workload at hand. The problem gains a new dimension when we use heterogeneous multicores in a fault-tolerant setting to execute real-time tasks. This dissertation addresses this problem by focusing on heterogeneous dual-core processors, and considering three different real-time task models, as stated below:

- *Independent frame-based tasks:* We consider a set of real-time tasks having a common period (which is also their common deadline). All tasks become ready to run at the beginning simultaneously and they must complete before the common deadline (also called the *frame deadline*.) This "execution frame" is repeated periodically. We execute the task set on a heterogeneous dual-core system with DVFS and DPM features to manage energy consumption. To implement fault tolerance, we took two separate approaches: i.) the standby-sparing approach, which designates one core as primary and the other as backup, ii.) the mixed primary/backup approach, where each core can execute a mix of primary and backup tasks. Under such settings,

the problem is to minimize the overall energy consumption by determining: i) the allocation of tasks such that the primary and backup copies of each task are assigned to different cores, and, ii) the processing frequency (speed) assignment to individual tasks.

- *Dependent frame-based tasks:* In this direction, we consider a set of frame-based real-time tasks with a common period with additional constraints due to the data dependencies between them. These dependencies govern the order of task execution and hence called the precedence constraints. A *directed acyclic graph (DAG)* can be used to model such constraints. The task set is executed on a heterogeneous dual-core system with DVFS and DPM enabled. We consider the mixed primary/backup approach for fault tolerance. The problem is to minimize the energy consumption by determining how to schedule the primary and backup tasks in proper order (along with the execution frequencies) such that all timing and reliability guarantees are met.

- *General periodic tasks:* We consider general periodic real-time tasks in which, each task can have an arbitrary period and generates a job instance periodically, with deadlines equal to the periods. We adopt the mixed primary/backup approach and schedul a mix of primary and backup copies on each of the two processing cores. The framework is priority-driven: at any time, on a given core, the ready task instance with highest priority is executed. Priorities are assigned to tasks offline (fixed-priority system). To manage the overall energy consumption, we use two mechanisms: i.) the primary tasks are executed at low voltage/frequency levels using DVFS, and, ii.) the backup copies are delayed to the extent it is possible to enable their cancellation in case the primary completes without a fault. In this setting, the problem is to determine the task allocation, priority assignment, execution frequency assignment and backup delaying mechanism which results in the minimal energy consumption for the overall system.

For the second part of this dissertation, we consider a control-system platform and investigate the problem of maximizing the overall *quality of control*, by means of allocating tasks to processors and choosing a suitable invocation rate (feasible period) for each of the real-time control tasks. In this second problem, we specifically consider a *homogeneous* multiprocessor system. The quality-of-control of a task is defined by a concave relationship to the invocation rate (period) of each real-time control task. The objective is to maximize the overall quality-of-control of the system while guaranteeing a feasible schedule by determining: i) the task to processor allocation, and, ii) the invocation-rate (or, period) of each control task.

1.2 Contributions

In the dissertation, in the context of the first problem of *energy-aware fault-tolerant real-time scheduling of independent frame-based tasks*, we made the following contributions.

- We first considered the so-called "standby-sparing" system using a heterogeneous dual-core processor, by designating one core as the *primary* and the other core as the *backup* processor. For each real-time task allocated to the primary core, we created a backup task with the same timing constraints and assigned it to the backup core. By means of an acceptance test, we can detect if a primary copy completes without error, and we cancel the backup copy accordingly, thereby saving energy at run-time. In our investigation, we elaborate on the factors which are affected based on our choice of primary and backup core. We develop a mechanism to cancel the backup task copies as soon as possible, while keeping the reliability guarantees. The simulation results show that our algorithms can effectively save energy consumption throughout a wide range of system configurations. Using our simulation results, we also identify that, there are some cases in which allowing some backup task execution may help reducing the overall energy consumption of the system.

- We also addressed the same problem on a dual-core system, but using the "mixed

primary/backup" approach, in which, each core can execute a mix of primary and backup task copies. Compared to the standby-sparing configuration, this system has more flexibility for task allocation among the two available cores. This allows us to make scheduling decisions with more choices available to us, and we opt to exploit it to reduce the overall energy consumption of the system. We developed several techniques for task allocation and execution frequency assignment which can remove redundant execution and conserve energy. Our simulation results also confirm this and it shows that our proposed schemes perform very close to the theoretical limit.

To the best of our knowledge, this is the first study which addresses energy minimization of fault-tolerant scheduling of real-time tasks on heterogeneous dual-core systems. We conduct extensive simulation studies and our contribution is published in [6, 7].

For the problem of *energy-aware fault-tolerant real-time scheduling of dependent frame-based tasks*, we considered the same platform of a heterogeneous dual-core system and the set of real-time tasks with a common period, however, the tasks now have data-dependencies among them, which is modeled using a directed acyclic graph (DAG). We developed a "main" or default schedule assuming no runtime faults will occur, and also a "contingency" schedule which takes care of fault recovery in case a fault is detected. We used mixed primary/backup approach and developed algorithms to generate the main and contingency schedules considering the data dependencies and the common deadline. We developed techniques for task partitioning and execution speed assignment which can result in a very low energy consumption while providing timing and reliability guarantees. We also proposed a reclamation technique which can operate at runtime and reduce the energy consumption even further. Our simulation experiments demonstrate the effectiveness of the proposed techniques and the result also show that our schemes perform very close to the theoretical lower limit for energy consumption. Our findings are published in [8].

For the problem of *energy-aware fault-tolerant real-time scheduling of general periodic tasks*, we considered a platform settings similar to the previous problems, however, we considered general periodic real-time tasks in which each task may have an arbitrary period,

which is also equal to their respective deadlines. Each task generates a sequence of jobs with a periodic interval. We investigated the traditional real-time scheduling theory for periodic tasks, and adapted it to support fault tolerance with timing guarantees. We used mixed primary/backup approach and allocated a mix of primary and backup tasks on each processing core. Each task is given a fixed priority offline and all of its jobs have that same priority. To save energy consumption, we used DVFS judiciously on each core to slow down the execution of the primary jobs without missing any deadlines. Furthermore, we delayed the execution of the backup jobs as much as possible so that they can be canceled once the primary copy completes successfully in fault-free cases. We developed several algorithms for task partitioning, priority assignment and execution frequency assignment which can reduce the overall energy consumption significantly. Our framework works in both offline and online phases. In particular, we proposed a novel priority assignment scheme which, when used with a "dual queue based delaying" mechanism, can result in the cancellation of most of the backup jobs and thereby significantly reduces the overall energy consumption. We justified our schemes using extensive simulation experiments which show that our proposed schemes perform considerably better than the traditional techniques in terms of overall energy consumption. We also show that the energy consumption of the proposed technique comes very close to the theoretical limit. These results are published in [9, 10].

For the problem of *Quality of Control Management via period assignment in real-time embedded systems*, we considered a homogeneous multicore system and a set of real-time control tasks, which is a type of general periodic tasks used in digital control systems. We modeled the quality-of-control as a function of task frequencies (invocation rate) and we developed a few techniques to maximize it, subject to the available computational capacity of the multicore processor. Although the problem is fundamentally NP-hard, we propose heuristic techniques to allocate tasks to processors and assign their invocation frequency. In essence, we developed a technique which reduces the problem to a (more powerful) single processor system, and then it computes the optimal frequency assignment for each

task. Then, task partitioning is performed using another heuristic considering the optimal frequencies. In this way, we use all the tasks' periods in a "global" way to determine our scheduling decisions. We performed extensive simulation experiments which show that our proposed technique can yield significant gains in the overall quality-of-control of the system and it comes very close to the theoretical optimal limit. These results are published in [11].

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 presents the background information and literature review on the relevant topics addressed in this dissertation. Chapter 3 presents the details of our system models and assumptions. The details of energy-aware scheduling for independent frame-based tasks with fault tolerance and timing constraints are presented in Chapter 4. Chapter 5 presents the research on energy-aware fault-tolerant scheduling of real-time frame-based tasks with precedence constraints. In Chapter 6, we discuss our algorithms and results for the fault-tolerant scheduling of general periodic tasks with strict timing guarantees and energy efficiency. In Chapter 7, we present our study and results for the quality-of-control management via period assignment in real-time embedded systems used to implement digital control systems on homogeneous multiprocessor platforms. Finally, Chapter 8 concludes the dissertation and offers directions for future work.

Chapter 2: Background and Literature Review

2.1 Real-Time Embedded Systems

Real-time systems are characterized by a set of computational activities that need to be completed within their respective *deadlines*. The correctness of the system depends not only on the output of a computation, but also at what *time* the output is produced [12]. In general, these systems are heavily coupled with the environment in which it operates. As the operating environment evolves with time, the system needs to react to it appropriately in a timely manner, in order to maintain smooth and flawless operation. For example, a multimedia system playing a video file needs to process each video frame within a predefined time period, which is determined based on the real-world time scale, such that the video stream plays at the target rate. A real-time activity failing to complete within the given time-period (or, deadline) may have serious consequences on the environment. In a *soft* real-time system (such as a video player), the consequences could simply be a degraded service to the users, but in a *hard* real-time system, failing to meet a deadline may have severe ramifications for the operating environment, which can potentially put human lives at risk in cases of safety-critical application-areas (e.g., space missions, autonomous driving, nuclear power plants, railway switching systems, flight control systems.) Therefore, in many practical hard real-time systems, additional hardware components are redundantly provisioned in order to make the deadline guarantees highly reliable.

A real-time task is a computational activity which has a deadline associated with it. A task can be a single computation, in which case it is called an *aperiodic* task, or, it can be a sequence of computational jobs released with a certain periodicity, called *periodic* tasks. An instance of a periodic task is called a *job*, and a task releases such jobs with a given minimum inter-arrival time, referred to as the *period* of the task. The maximum execution

time required to complete a job is called the *worst-case execution time* (WCET) of a job. The *absolute deadline* of a job is defined as the exact time when the job must complete its execution, whereas the *relative deadline* is defined as the time difference between the release time and the absolute deadline of the job. In many cases, the relative deadline is set to be equal to the period of the task – such tasks are called *implicit-deadline* tasks. An important sub-category of the periodic task model is the frame-based model where all tasks have the same period and implicit deadline [2–4]. This often models a set of related control tasks that need to be executed according to a fixed or preferred order on a regular basis.

When a set of real-time tasks are executed on an embedded hardware platform, multiple job instances may be eligible for execution at any given moment. Therefore, a *scheduling policy* is used to determine which ready job will be executed on a given processor next, which is either driven by a predetermined execution sequence computed offline (*clock-driven*), or determined at runtime based on the priority of the jobs (*priority-driven*). For periodic tasks with priority-based scheduling, if all jobs of a given task are assigned the same priority, then the framework is called a *fixed-priority* scheduling policy. Examples of some fixed-priority schedulers include *rate-monotonic* (RM) [13,14] and *deadline-monotonic* (DM) [15] policies. On the other hand, *dynamic-priority* schedulers may assign different priorities to different instances of the same task. *Earliest-deadline-first* (EDF) and *least-slack-time-first* (LST) are two well-known examples of such scheduling policies [5].

Many real-time applications consists of not only periodic tasks with hard timing constraints, but also non real-time tasks [12]. In these cases, the system ensures all the real-time jobs' completion before their respective deadlines and it opts to minimize the average response times of the non real-time (aperiodic) jobs. This problem has been addressed in the literature both in the context of fixed priority [16–18] and dynamic priority [19, 20] schedulers. The earliest deadline late (EDL) is known to be an optimal aperiodic server for dynamic-priority systems [19,20]. For dynamic-priority systems, another popular technique which is known to provide better response times for aperiodic tasks is called the constant bandwidth server (CBS) [21, 22].

In certain applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage [12]. Many researchers have described these precedence relations by means of a directed acyclic graph (DAG) [23–26]. The DAG imposes a partial order on the jobs of the task set, which needs to be satisfied at runtime. Many such techniques have been proposed in the literature which can optimize system performance while satisfying precedence relations [27–29].

Some real-time systems have multiple tasks which may compete for a shared resource. Although synchronization tools such as semaphores [30–32] can be exploited to provide exclusive use of the resource by some tasks, this might jeopardize the deadline of some other tasks waiting on those resources. Moreover, due to the priority inversion problem [33], a high priority task may miss its deadline because of some lower priority job holding a "lock". Many researches have been conducted to address mutual exclusion with guarantees on "blocking time". Some of such techniques include the stack resource policy [34], priority inheritance protocol [33] and priority ceiling protocol [35].

In case of a multiprocessor embedded system, the scheduling policy also needs to decide on which processor should a ready job be executed. The two main approaches proposed in the literature are *partitioned scheduling* and *global scheduling* [36]. Partitioned scheduling works in two steps. First, the task set is partitioned and each processor is assigned a subset of tasks. In the second step, an uniprocessor scheduling policy is used to execute them on each of the processors. Partitioned approach has been very popular because it allows to readily deploy the results obtained from uniprocessor-research, however, partitioning a set of tasks on to a set of multiprocessors is known to be NP-complete. Global scheduling, on the other hand, does not use task partitioning, instead, it maintains a global ready queue, and the M ready jobs with the highest priorities are executed on M processors in parallel. There exists global scheduling algorithms (e.g., Pfair [37], LLREF [38]) which can execute a task set optimally[1], however, these algorithms may suffer from *migration cost*, in which, a job is preempted on one processor and resumed on another processor later, incurring extra

[1]Optimal in the sense that if there exist some algorithm which can schedule a given task set feasibly (without any deadline miss), then the referred algorithm can also schedule it feasibly.

runtime overhead and context-localization issues. It has been shown in the literature [39] that, partitioned and global scheduling policies are incomparable in the sense that there exists task sets that can be feasibly (without missing any deadlines) scheduled by one but not the other.

Several heuristics have been proposed and analyzed for partitioning real-time tasks efficiently in a near optimal way [40–42]. These heuristics order tasks according to some certain criteria (e.g., decreasing *utilization* values, which is defined as the ratio of a task's worst-case execution time to its period and it often denotes a task's CPU requirements) and typically allocate tasks by using one of the well-known rules such as First-Fit, Best-Fit or Worst-Fit [40]. Lopez et al. [40] derived utilization bounds for partitioned scheduling on homogeneous multiprocessor systems when each processor uses EDF and variants of First-Fit, Best-Fit, and Worst-Fit that order tasks according to their utilization values first (often referred to as *Reasonable Allocation Decreasing (RAD)* algorithms). Carpenter et al. [41] classified the scheduling approaches on multiprocessor systems based on the complexity of the priority assignment of the tasks, and the degree of migration allowed. More recently, Baruah [42] used the metric called *speedup factor* to compare the performances of various allocation schemes for partitioned scheduling on multiprocessors. Pointing out the ineffectiveness of the utilization bound for partitioned scheduling, Baruah suggested that the speedup factor provides a deeper insight into the behavior of partitioning techniques and derived the speedup bounds for various RAD algorithms.

2.2 Multicore Systems

Multicore processors integrate multiple processing cores on a single chip along with multiple levels of caches. A 4-core homogeneous system with a multi-level cache architecture having both private and shared cache is schematically shown in Figure 2.1. Multicore processors were first proposed to primarily address the so called "power wall" problem which indicates that, although transistor count in a given chip-area would rapidly grow following Moore's

Law [43], increasing the CPU clock frequency on a single core CPU cannot be sustained due to the exponential increase in its power consumption and consequently the increase in heat-dissipation. Multicore processors handle this problem by increasing the number of cores in a single silicon die, but operating the cores at a sustainable clock rate, such that the *Thermal Design Power (TDP)* of the entire chip is not violated. This has the benefit of keeping the power-consumption low on each core, and at the same time execute multiple concurrent execution-threads on the available CPUs. The multiple cores would exploit thread-level parallelism where application-threads are distributed across the cores and executed in parallel. Impressive performance improvements can be achieved if the application code can be parallelized appropriately [44].

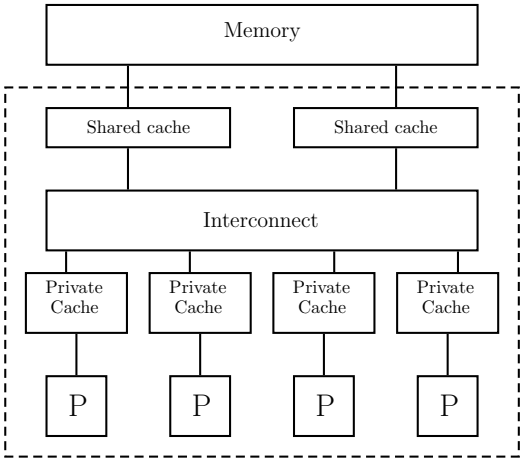


Figure 2.1: Multicore processor with shared memory

The earliest multicore processors were *homogeneous multicores*, in the sense that all the processing elements were identical in terms of micro-architectural design, functionalities, and power-performance characteristics. The historical way that the individual processors in a shared memory multiprocessor communicate has been through a common bus shared by all processors [45]. To improve memory access time, each processor has its own local cache memories, usually one or two levels. As a broadcast mechanism, the common bus

can facilitate cache coherency as well, however, common bus technologies are problematic in terms of latency and bandwidth. Some multicore processors use a cross-bar interconnect between processor modules. Other interconnection techniques include ring buses [46] and switched on-chip networks [47]. The shared memory itself can be implemented in different ways with various pros and cons, and all general purpose multicore processors today support a cache-coherent shared memory system. Various cache organization architectures have been proposed in the literature which relies on private, shared or mixed cache, and flat or hierarchical cache structures [48–51].

Multicore systems are heavily used in industrial control applications, personal computers such as laptops and desktops, mobile and embedded devices, and also in the data centers and cloud-computing. Many real-time and safety-critical applications have also been developed on homogeneous multicores [52–54] which opt to take advantage of scheduling workload via parallel threads. Although multicore processors can exploit application-code parallelism very efficiently, the increasing core-count hits the thermal power budget very quickly, and as a consequence, all the available processing cores cannot be used simultaneously– some of the cores needs to be switched off or significantly under-clocked. This gives rise to the "dark silicon" problem which is an important research area [55].

2.2.1 Heterogeneous Multicore Processors

A recent development in multicore technology is *heterogeneous*, or *asymmetric* multicores, in which, the multiple cores in a chip can be of very different nature [44, 56–58]. This difference may come from their Instruction Set Architectures (ISAs), power-performance characteristics and specializing cores for many compute-intensive but frequently occurring tasks (such as Discrete Fourier Analysis and cryptographic operations.). In a broad sense, the heterogeneous architectures can be categorized in two types: performance-asymmetric and functionality-asymmetric.

Performance asymmetry is achieved by combining processing cores which have difference in their power and performance characteristics, but all the cores share the same Instruction

Set Architecture (ISA). This is the most common type. The difference emerges from their micro-architectural features, such as in-order or out-of-order execution. The complex cores can provide very high performance, but at the cost of a high power consumption. The simpler cores are very power-efficient, but they provide only a modest performance. This is known as single-ISA heterogeneous architecture in the literature [59, 60]. The main advantage of this approach is that the same binary executable can be run on any type of cores, and the main challenge is to identify the appropriate set of cores based on the type of workload at hand [61–64]. Examples of commercial asymmetric multicores include ARM big.LITTLE [65] systems, which integrate high performance out-of-order cores with low-power in-order cores. Another example is nVidia Kal-El (Tegra 3) [66], which combines four high-performance cores with one low-power core.

The advantages of heterogeneous multicores in terms of power and energy consumption have been previously studied by Pricopy et al. [56]. It has been shown that power and performance characteristic can scale differently based on the core-type and the running application. Energy saving is obtained by switching on appropriate cores and setting a operating frequency via DVFS based on a given workload. Such task scheduling techniques have been proposed in the literature [67–71]. In [67], the authors presented a throughput-aware runtime task allocation approach called *Sparta*, in which, they predicted task behavior across heterogeneous cores and performed task-to-core mapping at runtime such that energy efficiency is maximized. In [68], the authors addressed self-adaptive multithreaded application in the context of heterogeneous multiprocessing and proposed a runtime system which can enhance "performance per watt" metric by dynamically adapting system state. [69] describes a hierarchical power management framework for asymmetric multicores by making use of control theory. Their scheme can achieve optimal power-performance efficiency while respecting the thermal design budget. The work in [70] coordinates various energy saving techniques such as dynamic voltage/frequency scaling, load balancing, and task migration based on the principles of "price theory", and develops a distributed and

scalable power management framework for heterogeneous multicores. A more detailed survey of the architectural and system-level techniques proposed for heterogeneous multicores can be found in [71].

Functionality-asymmetric processors represent a large class of heterogeneous architectures [44]. These are abundantly used in Multiprocessor System-on-Chips (MPSoC) which combine general purpose CPU cores, GPU cores, DSP blocks, and various hardware accelerators or IP blocks (e.g., video encoder/decoder, imaging, modem, WiFi, Bluetooth.)

2.3 Energy Management for Real-Time Systems

Energy management has been a very important aspect in designing embedded systems. Many embedded systems are battery powered, implying that energy-efficient operation would lead to a higher battery life. Moreover, for many ubiquitous devices (such as embedded sensors, pace-makers), it may not be practical to replace a drained battery, therefore, these systems need to be extremely low-power or be able to harvest energy from the environment. On the other hand, for high-performance embedded systems built on top of large multiprocessors, the energy and cooling cost, as well as the environment protection objectives contribute to the importance of energy management research.

The two primary energy-management techniques used in embedded systems are called *Dynamic Voltage and Frequency Scaling* (DVFS) and *Dynamic Power Management* (DPM). DVFS exploits the convex relationship between power consumption and operating frequency of a processor. By selecting an appropriate supply voltage and operating frequency, DVFS can effectively reduce the energy consumption for a given workload [72–74]. Most modern processors are equipped with DVFS for energy management, including ARM Cortex, Intel (with *Speedstep* technology) and AMD (with *PowerNow!* feature). DPM, on the other hand, aims to conserve energy by putting various system component in to a low-power state, when they are not in use. This technique has become very popular in research communities because it allows to minimize the leakage-power consumption, which is an increasing concern

in high-density silicon chips [75–78]. Most system components offer one of more low-power state(s). However, putting a device to a low-power state incurs some transition time and energy cost. As a result, the concept of *break-even time* has emerged, which denotes the minimum time-duration for which the device must remain in the low-power state in order to compensate for the cost of transition [79].

2.3.1 Energy Management for Uniprocessor Real-Time Systems

The earliest studies on DVFS for uniprocessor systems were themed to slow down the CPU as much as possible while ensuring that deadlines are met even when every task presents its worst-case workload. One of such study [72] presents an optimal offline algorithm and an online approximation algorithm. In practice, real-time jobs often complete earlier than the worst-case estimate of its execution time. When a task completes early, the system gains some additional CPU time which may be used by other tasks for further slow-down. This is known as *dynamic slack* and various techniques to reclaim it has been proposed in the literature [80, 81]. Another study [82] presents the idea of aggressive slowdown in order to take advantage of tasks' early completions in which, a task is started at a slow speed, but as it makes progress, the speed is increased. The motivation for this scheme is that the task is highly likely to complete before the system has to operate at very high frequency to finish the worst-case workload before deadline.

Although the earliest DVFS algorithms focused on minimizing the CPU energy consumption, later studies confirmed that the frequency-independent power component, which is contributed by peripheral devices such as memory, disk, I/O, have a substantial impact on the system-wide overall energy consumption [83]. This gives rise to the term *energy-efficient frequency* [83,84] which refers to the speed below which DVFS no longer provides additional energy savings. Many practical processors only provide a discrete set of speed-settings with their respective voltage levels. Such systems and the problem of adopting existing DVFS solutions to those systems have been investigated in [85].

DPM based algorithms commonly use the *task procrastination* technique, which postpones the execution of the ready jobs as much as possible by exploiting the system slack at that time, thereby compacting busy periods and yielding long idle intervals [86]. Moreover, DPM techniques are also free from some of the negative impacts of DVFS, e.g., reliability degradation [84, 87] and increased number of preemptions [88].

Several solution approaches combine DVFS and DPM techniques together [89–95]. Jejurikar et al. [96] combined the two techniques by computing a critical speed for DVFS and applying DPM whenever the CPU is idle. Later they proposed a version with dynamic slack reclamation and dynamic procrastination [97]. Quan et al. also proposed several algorithms for fixed priority tasks [98]. Devadas et al. [95] studied the effect of DVFS and DPM when multiple peripheral devices are connected to a single CPU.

2.3.2 Energy Management for Multiprocessor Real-Time Systems

Most of the existing work in multiprocessor real-time systems consider *homogeneous* multiprocessors [53, 99, 100], although in recent years some efforts have been carried out to generalize the results to heterogeneous systems with different power-performance characteristics [56, 69, 70, 101]. Most of the early papers focused exclusively on dynamic power consumption and implicitly ignored the static power while applying the DVFS technique. Gradually, the research community incorporated the static power into the power management frameworks in various ways [86]. Many studies consider a per-CPU DVFS model where a fixed speed can be set on each CPU, while others consider a per-task DVFS model where the speed of the CPU can be set at the dispatch of each task. In one of the earliest attempts, Aydin et al. [102] considered a homogeneous multiprocessor platform and partitioned the task set with the objective of eventually applying DVFS on to each CPU. They showed that a balanced partitioning is preferable for applying DVFS maximally (e.g., Worst-Fit), in the cases where each processor can be fully utilized. Chen et al. [76] proposed an approximation algorithm on similar settings, additionally considering leakage-current. [103] proposes a polynomial-time heuristic that partitions a set of periodic tasks over a multicore platform

and sets the CPU speed.

Several algorithms set the processor speed on a per-task basis. Zhu et al. [104] proposed such a family of algorithms which considers a large number of characteristics of the platforms and the typical applications running on it. They first proposed algorithms addressing only DVFS method, however, they also indicated when the system can be put to low-power state using DPM. Lu et al. [105] proposed DVFS algorithms to deal with *split tasks*, which are helpful to overcome the performance limitation of partitioned scheduling approaches in the presence of tasks with high utilization factors. Xu et al. [106] proposed an algorithm to deal with parallel tasks with frame-based execution (tasks with identical periods). They considered processors with a set of discrete frequencies with a lookup table to store the power consumption values related to each speed-setting. They formulated an Integer Linear Programming problem and presented efficient heuristics. Chen et al. [107] presented an approach based on mixed integer linear programming (MILP) which can characterize the idle intervals along with speed-settings, and thus optimizes both DVFS and DPM.

Some studies considered *voltage islands*, in which, groups of cores share the same voltage and frequency settings. Devadas et al. [52] presented an approach to reduce power consumption on a chip-multiprocessor (CMP) characterized by multiple sleep states and a single-frequency DVFS common to all cores. They identified the *global energy-efficient frequency* and suggested several algorithms based on Worst-Fit Decreasing heuristic to determine the number of active cores and the task-to-core allocation. They also suggested a technique to handle early completion of tasks. Pagani et al. [108] also considered similar settings and proposed a single-frequency-approximation algorithm. Gerards et al. [109] considered frame-based tasks with a common implicit deadline and presented an approach to determine optimal clock frequencies and the schedule to minimize energy consumption. Srinivasan et al. [110] also proposed MILP solutions using DVFS and DPM while considering discreet frequency settings and multiple sleep states with time and energy cost for state-transition.

Heterogeneous multiprocessors have been studied only recently and only a handful of

studies took advantage of power-performance asymmetry in order to minimize energy consumption [111–114]. Liu et al. [115] considered time-sensitive applications on a heterogeneous systems and proposed power-efficient algorithms. [116] also considered energy-efficient scheduling on multiprocessor and shed light on the advantages that heterogeneous architectures provide.

2.4 Reliability Management in Real-Time Systems

Computer systems are subject to different types of *faults*, which may lead to *errors*. Reliability management refers to safeguarding a computer system from such errors with some form of probabilistic guarantee. A fault is defined as a hardware or software defect that may prevent the system from operating correctly [117]. An error is a manifestation of a fault. In a broad sense, computer faults can be classified as *permanent*, *transient* and *intermittent* faults [1].

Permanent faults are typically caused by hardware failures, which can result from manufacturing defects at fabrication time, or the wear and tear due to aging. In a permanent fault scenario, a system component becomes unavailable, and never recovers. To handle such faults, additional redundant hardware components must be deployed, and a mechanism must be implemented which can detect a fault and replace the faulty component by a redundant one within some known latency [1]. The fault rate for a permanent fault can often be characterized by a bathtub curve [1]. Initially, most of the faults are caused by manufacturing defects. After that period, there is a long period of stable performance with low fault rate. Finally, when the components age, the fault rate increases due to regular wear and tear in its lifetime.

Transient faults lead to short-lived errors, which result in a malfunction of the system for a short period of time, then the system can go back to normal operation. These faults are caused by electromagnetic interferences, cosmic rays and some other environmental disturbances. Transient faults manifest in the form of *soft errors* or *single event upsets*

(SEUs) [1, 118, 119] that appear in the processor core logic or memory subsystems, leading to incorrect computations. Most of the faults experienced by a computer system are generally transient faults. To worsen the situation, it has also been suggested that the continued increase in component density due to advances in CMOS technology is increasing the vulnerability of systems to transient faults [120]. Because of the non-persistent nature of transient faults, software fault tolerance techniques can be applied to mitigate them. Transient faults which occur in "bursts" have also been addressed in the literature [121, 122]. Intermittent faults, on the other hand, appear and disappear over a long time-interval, impacting the system when they are active.

Reliability management in a real-time embedded system is implemented by fault tolerance, which refers to the technique of masking error or work around the fault encountered, e.g., by switching to a different component when a component encounters a fault. The basic idea of implementing fault tolerance is to exploit the redundancy of various resources. such as *hardware redundancy*, *software redundancy*, *time redundancy* and *information redundancy* [1]. A large body of research exists [27, 121, 123–129] that exploit *hardware* and *software* redundancy in order to provide reliability via fault tolerance.

Fault detection is an integral part of fault tolerance. Error detecting codes are often deployed for fault detection in memory and communication networks [130]. Another fault detection technique is called *acceptance tests* [1], which are software techniques that evaluate the outcome of a computation in order to accept or reject the result. It should be noted that fault detection techniques are generally not 100% accurate. For a fault detection test, a *coverage factor* is defined as the probability of detecting a fault successfully, given that such a fault has occurred [1]. Implementing fault tolerance in the presence of an imperfect fault detector has been studied in [131].

A classic hardware redundancy technique is called *M-of-N system* [1], which requires proper functioning of M components out of a total of N components in order to tolerate faults. This technique can provide tolerance to both transient and permanent faults. A classical example of the M-of-N system is a *triple-modular redundancy (TMR)* technique

which requires 2 out of 3 redundant components working correctly. A *duplex* system consist of two components that work in parallel, and their output of both components matche, then the result is considered to be correct, otherwise, the recovery procedure gets initiated.

Another widely-known hardware redundancy technique is called the *standby-sparing systems* [117]. In a standby-sparing system, each processing core (called *primary*) is associated with a dedicated backup core (called *spare*). Each task allocated to the primary core will have a corresponding backup copy allocated to the spare core, so that if a fault occurs on any of the core, the output from the other processing core may be used as correct output. The spare core can be implemented as a *hot* or as a *cold* spare. A spare is called hot, if it is executed in parallel with the primary core. A cold spare is kept idle during normal operation, and it is activated upon the detection of a fault [1]. Some other researches proposed a *primary-backup (PB)* model [132–134], which is quite similar to the standby-sparing technique in the sense that each task has one primary copy and one backup copy. However, processing cores are not designated as *primary* or *backup*, instead, any task copy can be allocated to any of the available processing cores, with the only constraint that the primary and backup copy of the same task must be placed to different processing cores. Both standby-sparing and primary-backup techniques can provide resiliency to both transient and permanent faults.

Checkpointing has been a popular technique for implementing fault tolerance in cases where restarting a job from the beginning is very expensive [135–138]. Essentially, a checkpoint is a snapshot of the entire state of the process, and it contains all information required to restart the process from that point [1]. When a fault is detected, the system is reverted to the last saved checkpoint, and the task is restarted will all the saved information. Important questions like optimality of checkpointing-interval has been studied in [139, 140]. Optimal number of checkpoints and their interval have also been addressed for low-power systems in [141].

Recovery techniques are also classified as forward and backward recovery solutions [142–145]. Backward recovery techniques usually reverts the system to a prior state, hopefully

one which precedes the fault. Forward recovery, on the other hand, attempts to recover the faulty state, e.g., by using error correcting codes. Both forward and backward recovery techniques are key to achieve different levels of fault tolerance and graceful degradation [143, 146, 147].

A software redundancy technique is called *N-version programming*, in which, N independent teams of programmers develop N versions of the same software specification, and these N copies execute in parallel and their output is voted on [148–150]. Challenges such as consistent comparison, version independence and other issues has been addressed in [151, 152]. Another technique which is very similar to N-version programming is called *recovery blocks*, in which, there are multiple version of the software, but only one version runs at any one time [153, 154]. When a fault is detected, the system executes one of the available recovery blocks. Distributed recovery blocks have also been addressed in the literature [155, 156].

2.4.1 Reliability Management on Multiprocessor Real-Time Systems

The available multiple processing cores in a shared memory chip multiprocessor system can be used not only to improve energy efficiency, but also to provide resiliency to failures. Many researchers addressed the problem of designing a fault-tolerant system built on multiprocessors [157–161]. Guo et al. proposed generalized standby sparing system in which the multiple cores are grouped into two: primary and spare, and they suggested energy-efficient scheduling of real-time applications [162]. Huang et al. suggested a binary tree analysis method to measure system-level reliability when task replicas are allocated to different processing cores [124]. Resilience and energy efficiency in high-performance computing systems built on multiprocessors has been addressed in [163]. The authors developed analytical model that capture the reliability degradation due to energy-efficient operation, and they provided guidelines on applying DVFS on such systems. Preference-oriented scheduling has been developed for fault-tolerant systems in [164] in which, primary tasks have *as soon as possible*, and backup copies have *as late as possible* execution preference. The authors argued that their scheduler can reduce the overlap between the primary and backup of the same

task, by scheduling the backups as late as possible considering their deadline. In [165], the researchers exploited the implicit relations among periods and recovery costs among tasks and develop a novel metric, called "compatibility index", to quantify how compatible a task set is when they are allocated on the same core, and thereby achieved an enhanced fault-tolerant scheduling for fixed priority systems. Task-to-core mapping was addressed in [166] for reliable systems built on multiprocessors in which, the authors developed integer linear programming (ILP) methods which generate remapping of tasks in a fault scenario, with a goal to improve communication and migration overhead. Zhao et al developed fault-tolerant scheduling for heterogeneous platforms where number of replicas are determined at runtime [167]. Overloading was also used as an efficient redundancy technique [168–170], in which overlapped backup copies can be added to the schedule as long as their primary copies have not been scheduled on the same processor. Al-Omari et al. suggested overloading primary copies with the backup copies can provide up to 25% better schedulability compared to only backup-backup overloading [169, 170].

2.5 Joint Management of Energy and Reliability

Reliability and energy management are conflicting design goals, in the sense that ensuring high reliability of a system generally comes with an additional cost in energy consumption. Reliability is implemented by means of fault tolerance, which exploits some form of redundancy in the system (hardware, software, time, and information). Deploying redundant components by just duplicating them would also double the energy consumption of the system. Therefore, researchers have been seeking to find ways to deploy redundancy while minimizing the overall energy consumption and it has been a very popular research area to tackle reliability and energy management in tandem [84, 141, 171, 172].

A time redundancy-based technique called *reliability-aware power management* (RAPM) has been proposed in the literature which can provide reliability against transient faults, and can mitigate the reliability degradation caused by DVFS [173, 174]. The idea is

to divide the system slack in to two portions, and use one of them to schedule a recovery (backup) task. The other portion of slack can be used to slow-down the processor. Zhu et al. [175] proposed an optimistic RA-PM scheme that considers the probabilistic execution time information. Zhao et al. [176] proposed an enhancement to the RA-PM scheme by considering a *shared recovery block*. The same group also proposed *reliability oriented power management* (RO-PM), in which, the system can be designed to achieve an arbitrary reliability target, while minimizing energy consumption, both for frame-based applications [177] and periodic tasks [178]. All these time redundancy techniques are effective and energy efficient, however they cannot provide tolerance to any permanent fault, as no hardware redundancy is present.

Several studies addressed the problem of tolerating both transient and permanent faults, and they use hardware redundancy techniques. Ejlali et al. proposed an *energy-efficient standby-sparing* technique [53] in which the primary copy uses DVFS and the backup copy uses DPM techniques, and backup copies are activated as late as possible considering the deadline. They attempted to minimize the *overlapped* execution of the primary and backup copies of the same task. Such overlap minimizing techniques were also proposed by Unsal et al. [179]. Later, other researchers investigated some variants of the low-energy standby-sparing systems and presented comparative results [180, 181]. Haque et al. considered the standby-sparing mechanism and proposed an energy-aware scheme for periodic real-time applications, both for dynamic priority systems [54] and for fixed-priority systems [100]. The same group also presented a characterization of the subtle interplay between the processing frequency, replication level, reliability, fault coverage, and energy consumption on DVS-enabled multicore systems [182]. Guo et al. [134] considered *primary/backup* (PB) technique and proposed energy-efficient algorithms that can tolerate a single permanent fault along with maintaining arbitrary system-level reliability with respect to transient faults. The proposed schemes demonstrated considerable energy savings, however, they are implemented on homogeneous multiprocessors.

In recent years, a handful of research has been done on hardware redundancy for heterogeneous computing platforms [183–185]. In [183], the authors developed a *fully polynomial time approximation scheme* (FPTAS) for combining partitioning and replication to achieve high reliability. [184] considered periodic real-time applications on heterogeneous systems and presented a very flexible approach for handling system schedulability and reliability at the same time. Zhu et al. [185] considered a fault-tolerant framework on heterogeneous systems which can provide a predetermined quality-of-service even in the face of a single node-failure. Xu et al. [186] considered a framework which provides reliability on heterogeneous embedded systems and also minimizes energy consumption. They used a method to transform the application's reliability goal in to each task's reliability requirement, then proposed a method to minimize their energy consumption. They considered DAG-based embedded system applications; however, they did not consider any real-time constraints.

Devaraj [187] proposed a solution to the problem of scheduling real-time tasks on heterogenous systems which guarantees timeliness and feasibility. They used a linear programming algorithm to find feasible schedules for real-time task sets considering heterogeneous execution platforms. Li et al. [188] proposed a framework to execute real-time applications on heterogeneous multicore processors, which minimizes temperature and energy consumption. They considered a graph based application model and scheduled the application tasks on to the processing cores in an energy/thermal-aware manner. Their results on real-world applications demonstrated the effectiveness of their approach, however, they did not consider task replication or fault tolerance in their work. Zhou et al. [189] considered heterogeneous platforms for applications with deadline constraints and reliability. They developed a "earliest finish-time" based algorithm for heterogeneous MPSoCs, which executes graph based real-time tasks. Their solution strives to maximize reliability to transient and permanent faults, however, they did not take energy consumption in to account. Liu et al. [190] proposed an adaptive fault-tolerant scheduling mechanism for real-time systems executing on heterogeneous multiprocessors. They used task replication and computed the number of required replicas which guarantees reliability and deadline. However, their work also did

not address the energy consumption dimension.

A number of recent research studies explored the joint management of timeliness, energy consumption, and reliability. For instance, Bansal et al. [191] proposed energy aware fixed priority scheduling for real-time tasks in which they considered execution-preferences for different tasks (primary or backup). They improved the energy consumption of the preference-oriented scheduling proposed in [192], by applying DVFS and DPM techniques. However, their work was based on a single processor system, and cannot be easily generalized to multicore platforms or tolerate permanent faults. Zhao et al. [193] proposed an energy-efficient standby-sparing technique, which executes a mix of high and low criticality tasks. They scaled the primary processor with DVFS, and also, they extended their algorithm for cluster/island systems, however, their work did not consider heterogeneous processors.

Safari et al. [194] also proposed a low-energy standby-sparing scheme for mixed criticality systems, in which, they considered graph based real-time applications, and executed it on multicore system with fault tolerance. They used convex optimization to exploit DVFS along with DPM to save energy under timeliness and reliability constraints. That paper considered homogeneous multicore systems. Kumar et al. [195] addressed heterogeneous multicore systems and proposed a framework for energy-efficient scheduling of periodic real-time tasks with fault tolerance. They modeled the problem as a constraint optimization problem and also proposed several low-overhead heuristics. However, their setting does not allow preemption of real-time tasks, which may have a strong and negative impact on the schedulability of periodic real-time task sets for which preemptive scheduling is the norm [5].

2.6 Management of Quality-of-Control

In industrial control, applications are generally implemented via a set of periodic real-time tasks. The quality-of-control metric of the application is defined as a quantitative measure expressing control performance [196, 197] . This metric is associated with each control task

and usually directly depends on the periods of the control tasks [198, 199]. In general, the shorter the period, the better the quality of control. However, due to various constraints in the system, periods of the tasks can only be varied within a given range. In a seminal paper, Seto et al. [200] presented a model where the periods of real-time control tasks could be changed up to a limit, in order to optimize some *control performance index*. They showed that in some industrial control applications, an exponential decay function can be used to model the tasks' performance indices. Seto formulated the problem as a non-linear programming problem and solved it using the *Lagrange multipliers* technique. Bini and Di Natale [201] suggested a generalized framework for maximizing various objective functions in the context of fixed-priority scheduling. They defined the set of all period assignments that can produce feasible schedules as the optimization domain and then used mathematical properties of the objective function to find the optimal assignment. They used Time-Demand Analysis technique to assess the feasibility in a single processor environment.

Buttazzo et al. [202] proposed a scheduling framework (called *elastic scheduling*) in which periodic tasks can change their rates in order to tackle the overload conditions. For each task, their model has a minimum period, a maximum period and an elasticity coefficient. When the system is overloaded, the tasks can increase their periods (compressing); and when the system is underloaded, the task periods can be reduced (decompressing). The authors developed an efficient algorithm where they modeled each task as a spring with minimum and maximum length and a rigidity coefficient. This model proved useful for supporting both multimedia and control applications in which execution rates of some computational activities have to be dynamically tuned as a function of the current system state. However, their work does not consider arbitrary control cost functions, and is developed for single-processor systems. Marinoni et al. [203] incorporated DVS technique to the elastic task model.

Chantem et al. [204] considered a setting where task periods and deadlines are chosen in order to improve the schedulability. Later, the same research group explored a setting where minimum and maximum allowable periods are associated with each task, and a

performance index is optimized [205]. Their performance index is a quadratic function of utilization of each task. They focused on uni-processor system settings. Wu et al. also used a quadratic cost function as their performance loss index which depends on the period of the tasks [206]. Fontanelli et al. [207] explored similar problem settings associated with real-time control tasks and cost functions, but their results were obtained under stochastic model. Tian et al. [208] leveraged the elasticity of tasks to improve the quality-of-control in applications using a utilization-based quadratic cost function. Kim et al. [209] proposed a task model for automotive systems called *rhythmic tasks* where period and worst-case execution time of each task can vary continuously based on some external event. They did not use any performance index, rather, their main goal was to improve schedulability. In a recent paper, Buttazzo et al. [210] extended the idea and proposed rate-adaptive tasks where period, worst-case execution time and deadline can vary according to the angular velocity of the crank-shaft. However, their main objective was to improve schedulability, not optimizing any performance index. Rajkumar et al. [211] proposed a model called Q-RAM where each real-time application are assigned resources to maximize overall system utility along multiple and discrete QoS dimensions, for single-processor systems. After resources are allocated, each application can choose its own period and execution time.

Chapter 3: System Model and Assumptions

In this chapter, we present the system model and assumptions of our research on energy efficient scheduling of real-time tasks with fault tolerance. Specifically, we present the platform, power and fault models used in this research. We present a generalized system model which assumes heterogeneous multiprocessors and covers the larger scope of this dissertation. We introduce the notations for specific system models in their relevant chapters. In Chapter 7, we tackle the different problem of maximizing Quality-of-Control on a multiprocessor system; for the sake of clarity, the definitions and assumptions pertinent to that research are presented at the beginning of that chapter.

3.1 System Model

3.1.1 Platform and Application model

We consider a shared memory dual-core system that combines two processing cores on the same silicon chip, and both cores share a common L2 cache. We consider a heterogeneous platform and one of the cores is a high-performance and power-hungry ("big") core, and the other one is a relatively slow but power-efficient ("little") core, which we refer to as HP and LP cores, respectively. The cores are assumed to have a single instruction set architecture (ISA), which has the advantage that an executable of a task can run on either of the cores. Each of the two processing cores is equipped with the *Dynamic Voltage and Frequency Scaling (DVFS)* feature that allows changing the frequency (processing speed) at run-time to manage energy consumption. Both of the cores also have the *Dynamic Power Management (DPM)* feature which allows a core to be put to a very low-power state (sleep mode.) DVFS and DPM are widely used as an energy management technique at the operating system level.

The target application consists of a set of n real-time periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, where each task τ_i is represented by an 8-tuple $(T_i, D_i, C_i^{HP}, C_i^{LP}, a_i^{HP}, a_i^{LP}, \alpha_i^{HP}, \alpha_i^{LP})$. We assume the *general periodic task* model in which the task τ_i generates a job instance periodically with the period T_i. Each instance of the periodic task τ_i must complete within the relative deadline D_i, which is equal to its period (implicit deadline). Our research tackles the joint problem of energy and reliability management on dual-core heterogeneous systems for different real-time task models on different parts of this dissertation: 1.) Frame-based independent tasks (Chapter 4), 2.) Frame-based tasks with precedence constraints (Chapter 5), 3.) General periodic tasks (Chapter 6). Additional notations and assumptions for each of these sub-models will be introduced at the beginning of the respective chapters.

A task instance of τ_i that requires C_i number of cycles on a given core may take up to $W_i(f) = C_i/f$ units of execution time on that core, if executed at the frequency level f. The worst-case number of cycles required by the instances of the task τ_i is denoted by C_i. Due to the architectural differences, a task's required number of cycles, and hence execution time, can be different on the HP and LP cores. Therefore, we use superscripts HP and LP to denote the variables on the HP or the LP core (e.g., C_i^{LP}, W_i^{LP}, C_i^{HP}, W_i^{HP}). The maximum frequency levels supported by the HP and LP cores are denoted by f_{max}^{HP} and f_{max}^{LP}, respectively. We assume $f_{max}^{HP} = 1.0$, and normalize all other frequency values with respect to that value. We define the *nominal utilization* of a task τ_i as $U_i = W_i^{HP}(f_{max}^{HP})/T_i = W_i^{HP}(1.0)/T_i = C_i^{HP}/T_i$.

3.1.2 Power Model

In our settings, each processing core is equipped with two well-known operating system-level energy management techniques, namely, *Dynamic Voltage and Frequency Scaling (DVFS)* and *Dynamic Power Management (DPM)*. DVFS can execute the processor at a reduced operating frequency and thereby achieve a reduction in its dynamic power consumption. We model the dynamic power consumption of a task τ_i as $P_i(f) = a_i\, f^3 + \alpha_i$ where a_i denotes

the switching capacitance, α_i indicates the *frequency-independent* power consumption [100, 176], and f is the processing frequency of the task. The processors support any operating frequency within the continuous range from 0.2 to their respective maximum-frequency. Due to the asymmetric nature of the dual-core system, the tasks exhibit different power consumption characteristics on different cores. Again, we use the superscripts HP and LP, to distinguish between the values of the task power consumption parameters on high-power and power-efficient cores, respectively (e.g., α_i^{LP}, P_i^{HP}).

Each core executes tasks in the *active* state, dissipating power as determined by the characteristics of the current task and processing frequency. When a core does not execute tasks, it remains in low-power *(idle)* state by virtue of the DPM technique. The low-power (idle) power consumption of the high-performance and low-power cores are denoted by P_{idle}^{HP} and P_{idle}^{LP}, respectively. We assume those figures include the *static* power consumption of the corresponding core as well. Moreover, we assume the overhead of changing frequency or power state is negligible. Finally, the energy consumption during a time interval is given by the aggregate power consumption during the same interval.

The existence of the frequency-independent power component implies the existence of an *energy-efficient frequency (f_{ee})* below which DVFS affects the overall energy consumption negatively [84]. The value of f_{ee} can be analytically derived using standard techniques [84].

3.1.3 Fault and Recovery Model

In our fault model, we make provisions to tolerate both *transient* and *permanent* faults. Transient faults are primarily caused by electromagnetic interference and cosmic rays (including alpha particles) [1]. They manifest in the form of *soft errors* (or *single event upsets (SEUs)*) that appear in the processor core logic or memory subsystems, leading to incorrect computations. Moreover, with increased technology scaling and the use of aggressive low-power design techniques such as near-threshold voltage operation, the CMOS circuits become more vulnerable to transient faults [84, 120]. By their nature, the transient faults are short-lived, and it is often possible to invoke an alternative back-up task after such a

fault is detected on a specific task.

The permanent faults, on the other hand, are often the results of manufacturing defects, aging, or adverse temperature/environmental conditions – they lead to the unavailability of the affected processing unit until it is replaced. It is often necessary to have redundant hardware, in the form of extra processing units, in order to tolerate the permanent faults [117]. Various techniques, such as lockstep configuration, redundant multithreading, built-in self test, etc., can be used to detect permanent faults [212]. When a permanent fault on any of the core is detected, all tasks are executed on the remaining operational single core until a hardware replacement is performed.

The reliability constraints we consider in our fault model states that, given a set of real-time tasks, it must be ensured that each task will meet its deadline, even when it experiences a transient fault in one of its copies (primary or backup), or a permanent fault in one of the cores. Specifically, in our framework, we tolerate the following type of faults:

- A transient fault per each (primary) task instance, and,
- A permanent fault of any of the processing cores.

Each (primary) task τ_i has an associated backup task B_i with exact same timing parameters. τ_i and B_i are allocated to different processing cores. Whenever a task instance is released, its backup copy is also released on the alternate core. In order to maintain energy efficiency, the primary copies are executed at a reduced frequency via DVFS, while the backups are executed at their maximum frequencies. The backup copy instances are delayed (using DPM) as much as possible while respecting their deadlines.

When a primary copy completes, the *acceptance* (or, *sanity*) tests [1] are performed to check the existence of errors induced by *transient* faults. If a fault is not detected, the corresponding backup copy (or, its remaining part) on the other core may be cancelled to save energy. Otherwise, the backup copy runs to completion. If a permanent fault occurs on any of the cores, the other core can still execute one copy of each task's instances. Therefore, our system can tolerate at most one transient fault per task *or* at most one

permanent fault. When a transient fault is detected and tolerated, the system can go back to normal operation and tolerate more faults, however, when a permanent fault occurs, the system can tolerate it, but after that point it is no longer able to tolerate any additional transient or permanent faults until a hardware replacement is performed.

Chapter 4: Energy-aware Fault-Tolerant Real-Time Scheduling of Independent Frame-based Tasks

In this chapter, we consider the joint management of energy consumption and reliability requirements on heterogeneous dual-core systems for *independent* frame-based real-time tasks. We assume the platform and application model presented in Chapter 3.1, however, we consider frame-based model in which all tasks have the same period, D, i.e., for each task τ_i, $T_i = D_i = D$. All tasks are released and become ready to execute at the beginning of a frame, and they are required to finish before the common period D. The same pattern of execution repeats every frame (period), hence the name *frame-based systems* [2–4]. Our heterogeneous computing platform is made up of a single-ISA dual-core system that consists of a high-performance and power-hungry ("big") core, and a relatively slow but power-efficient ("little") core, which are connected via a shared memory on the same silicon chip. To ensure reliability, each (primary) task has an associated backup task, and these two copies of the same task are always allocated to different processing cores. Each core is equipped with DVFS and DPM which can be used to implement energy-efficient operation. Primary tasks are executed at a reduced speed using DVFS while still guaranteeing the deadline, and DPM is used to put the core to sleep when it is idle. Backup copies are always executed at the maximum speed of the respective processing core, and as soon as the primary copy completes, the backup execution is cancelled (to save energy). We consider both standby sparing and mixed-primary/backup approach to solve this problem. Our results on this research are published in [6, 7].

4.1 Energy-Aware Standby-Sparing on Heterogeneous Multicore Systems

Real-time systems are usually safety-critical systems, and therefore, guaranteeing reliability requirements is of utmost importance. A well-known approach to reliability is called *Standby-Sparing* systems [53, 162, 193, 213], in which, each CPU (primarily) executing the workload has a corresponding backup CPU. All main (primary) tasks are allocated to the primary CPU, and all backup tasks are placed on the backup CPU. Any faults on the primary CPU will result in the activation of the task on the backup CPU. When we have deadline constraints, the backup CPU may need to be activated even without any fault on the primary, just to make sure the deadline is not missed. One can easily infer that the addition of a second processing core as backup significantly increases the overall energy consumption of the system.

In *energy-aware* standby-sparing configuration, the primary processor uses DVFS and DPM to execute the main tasks at reduced processing speeds and put the processor to low-power state in order to save power. The spare processor uses only DPM; it delays the back-ups as much as possible in order to cancel them dynamically to save power, if the corresponding main task(s) complete without errors. Several studies have been performed to minimize energy consumption with such systems on *homogeneous* multicores [53, 100, 162]. To the best of our knowledge, ours is the first study which addresses reliability and energy-management for real-time systems in the context of *heterogeneous* multicore processors. In this study, we re-visit the energy-aware standby-sparing problem in the context of emerging heterogeneous multi-core systems for frame-based real-time embedded applications. In addition to the problem of determining the frequency assignments on the primary, the new settings introduce a new dimension, namely, whether to use the power-efficient core as the primary or spare. We investigate both of these dimensions, propose solutions, and present a comprehensive experimental evaluation. Our results indicate that, unlike the homogeneous case, allowing some overlaps between the primary and back-up

copies may help to reduce energy, in particular when the high-performance core is used as the primary. We also show that, in general, designating the power-efficient core as the primary and high-power core as the spare is preferable, except when the system utilization is high. Our analysis is performed under the realistic assumption that the execution time and power consumption figures of different applications may scale by different ratios on the big and little cores [56].

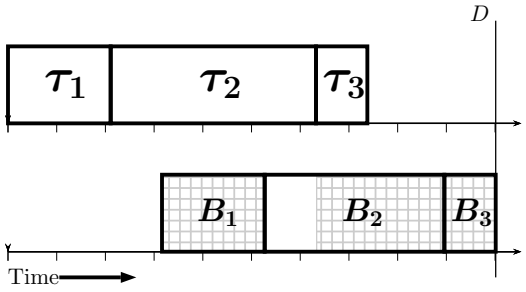


Figure 4.1: A standby-sparing system

We deploy the energy-aware standby sparing technique on the dual-processor platform as shown in Figure 4.1. Throughout this section, we show the primary processor at the top, and the spare processor at the bottom in the figures. When a task (τ_i) is allocated to the primary core, a back-up copy (B_i) is also allocated to the spare core. The main and backup tasks follow the same execution order on their respective processors. The DPM-enabled spare core remains in low-power state as much as possible, delaying the execution of the back-up tasks while still meeting the deadline (Figure 4.1). If there is no fault affecting a primary task, the (remaining part of the) corresponding back-up task on the spare processor is cancelled dynamically. For example, in Figure 4.1, B_1 and B_3 are completely cancelled, because the corresponding main tasks (τ_1 and τ_3) complete successfully before the scheduled start times of B_1 and B_3. Similarly, a part of B_2 is cancelled as soon as τ_2 completes successfully (we are using the dashed lines to indicate cancelled executions in the schedules). In case of a transient fault, the back-up task executes as specified in the spare schedule. If a permanent

fault affects the primary processor, the spare copy executes the back-up tasks after that point. The back-up tasks are executed at the maximum processing frequency of the spare core. This energy-aware standby-sparing arrangement can tolerate a single fault of either processor, while enabling the recovery of transient faults affecting any subset of the main tasks on the primary processor [53].

Problem Statement. In this section, we address the following problem: Given a set of frame-based real-time tasks and a heterogeneous dual-core system, minimize the overall energy consumption while still meeting the deadlines, by determining:

1. which core should be designated as the *primary* and *spare* in the standby-sparing system, and,

2. what processing frequency assignments should be made to tasks on the *primary* core.

We investigate these two dimensions in Sections 4.1.1.1 and 4.1.1.2, respectively.

4.1.1 Proposed Schemes

4.1.1.1 Primary/Backup Role Assignment

The first design dimension to be addressed is to whether designate as the primary processor the high-performance core (HP) or low-power (LP) core. The two options are, therefore:

- **FasterP**: Use HP as primary, LP as spare

- **SlowerP**: Use LP as primary, HP as spare

It turns out that this decision can have a significant impact on the overall energy consumption of the system, depending on the workload. For instance, if a high-performance core is used as the primary core, ideally it should be slowed down through DVFS to reduce its energy consumption. However, that would extend the completion times on the primary, and the overlapped backup executions could significantly increase the energy consumption on the slow, power-efficient, spare processor. Conversely, if the low-power core is used as

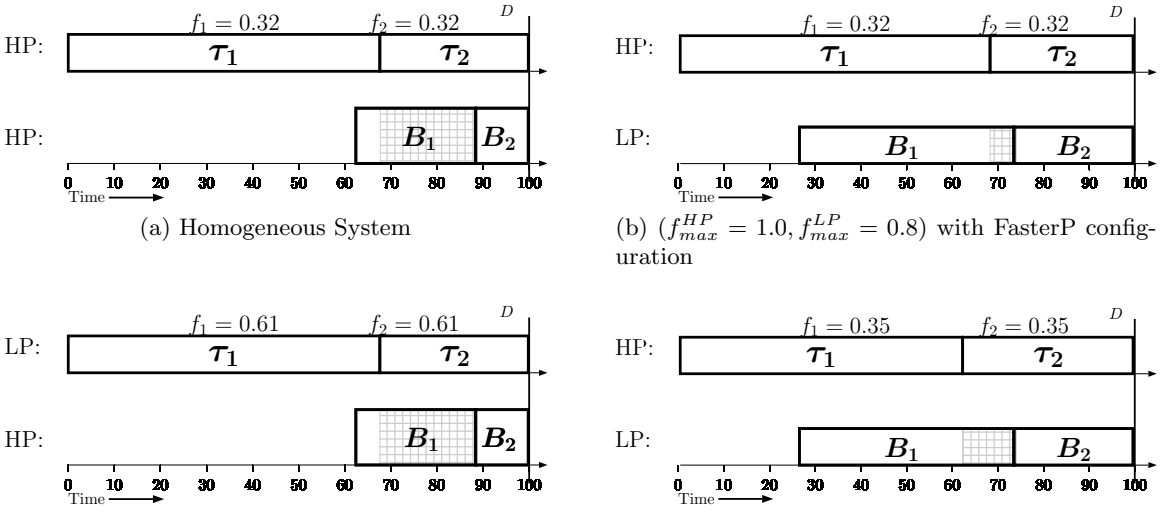
the primary, due to its modest performance, the task completion times would naturally shift even at high frequencies, increasing the overlaps with the power-hungry spare processor.

Consider two tasks with parameters given in Table 4.1, where the execution times and energy consumptions on both cores (E^{HP}, E^{LP}) under respective maximum frequencies are shown. Assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$. For both tasks, $a_i^{HP} = 1.0$, $a_i^{LP} = 0.3$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.03$.

Table 4.1: Example Task Set 1

	W_i^{HP}	W_i^{LP}	E^{HP}	E^{LP}
τ_1	22	52	24.2	9.55
τ_2	10	24	25.8	4.4

When this task set is assigned to a homogeneous system consisting of two high-power cores where primary core is slowed down as much as possible, the execution of the system is shown in Figure 4.2a. The last 22 time units of B_1's execution are dynamically cancelled when τ_1 executes successfully, but B_2 is executed fully. This yields an overall energy consumption of 0.295 Joules (in all the examples, we focus on the energy consumed in the fault-free execution sequence, because faults are rare events). But once the application is moved to a heterogeneous system where the normalized maximum frequencies are $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$, respectively, we can take advantage of the power-efficient execution on the little core, both for FasterP and SlowerP configurations. In the FasterP case (Figure 4.2b), the overlapped portion is larger than the one in Figure 4.2a, but the overall energy consumption decreases to 0.265 Joules, giving an improvement of 10.28%. This is due to the fact that the power-efficient core is used at its maximum frequency to execute the spare. When a SlowerP configuration is used (Figure 4.2c), we have the same amount of overlap as in the homogeneous case, but now the primary workload can be executed on a power-efficient core with maximum slow-down ($f_i = 0.61$). The energy consumption of the SlowerP system shows a 11.28% improvement compared to the homogeneous system.



(a) Homogeneous System

(b) ($f_{max}^{HP} = 1.0, f_{max}^{LP} = 0.8$) with FasterP configuration

(c) ($f_{max}^{HP} = 1.0, f_{max}^{LP} = 0.8$) with SlowerP configuration

(d) ($f_{max}^{HP} = 1.0, f_{max}^{LP} = 0.8$) with modified tasks (FasterP)

Figure 4.2: Executions under different configurations

Now, if we slightly change the task set parameters and increase the W_2^{HP} and W_2^{LP} values to 13 and 31, respectively, we can compute that the execution on FasterP in Figure 4.2d yields a 12.25% energy savings compared to the homogeneous system, and SlowerP yields only a 7.24% improvement (not shown). In this case, the increased workload on a SlowerP configuration causes the high-power backup to start early and execute at its maximum frequency, therefore the FasterP configuration is more advantageous to execute backups on the low-power spare during the unavoidable overlap. This example shows that *the best primary/spare configuration to minimize the overall energy is dependent on the characteristics of the workload at hand.*

4.1.1.2 Frequency Assignment on Primary

Once all the tasks are allocated to the primary processor and their respective copies to the spare processor, the next dimension is to determine the processing frequencies for the main tasks on the primary. Normally, one would want to exploit DVFS to slow down execution on the primary and save energy. But, slowing down the main copy of the task has the potential

of increasing its completion time, and thereby increasing the overlap between the main task and its backup copy, resulting in a higher overall energy consumption. Moreover, through a cascading effect, such a decision could also shift the completion times of the following main tasks and further increase the energy consumption.

We will use a heterogeneous dual-core system with $f_{max}^{HP} = 1.0$, $f_{max}^{LP} = 0.8$, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$, arranged in a FasterP configuration, to demonstrate the execution scenarios and performance trade-offs of the proposed schemes. Table 4.2 shows two tasks τ_1 and τ_2, with their worst-case execution times and energy consumptions on both low-power and high-power cores under respective maximum frequencies. For both of these tasks, $a_i^{HP} = 1.0$, $a_i^{LP} = 0.6$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.06$. In the following discussion, for brevity, all the C and f values refer to those on the primary (high-performance) core.

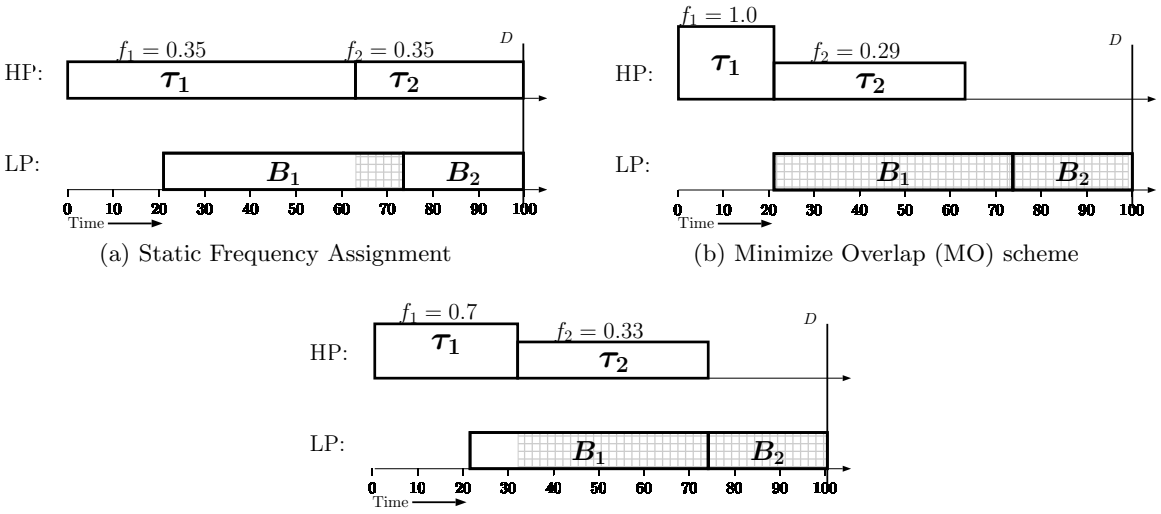
Table 4.2: Example Task Set 2

	W_i^{HP}	W_i^{LP}	E^{HP}	E^{LP}
τ_1	22	49	24.2	18.2
τ_2	13	29	14.3	10.6

We propose three schemes for frequency assignment to tasks on the primary core:

1. Static Frequency Assignment (Static)
2. Minimize Overlap for Current Task (MO)
3. Overlap-Aware Energy Minimization (OA)

Static Frequency Assignment (Static). In this scheme, the frequency assignments to the tasks are done statically, but considering the energy-efficient frequency as well as the minimum (uniform) frequency that guarantees the deadline. Specifically, task τ_i is executed at frequency $f_i = Max(f_i^{ee}, f_U)$, where, $f_U = \frac{\sum C_j}{D}$ is the minimum frequency that ensures meeting the frame deadline for all tasks.



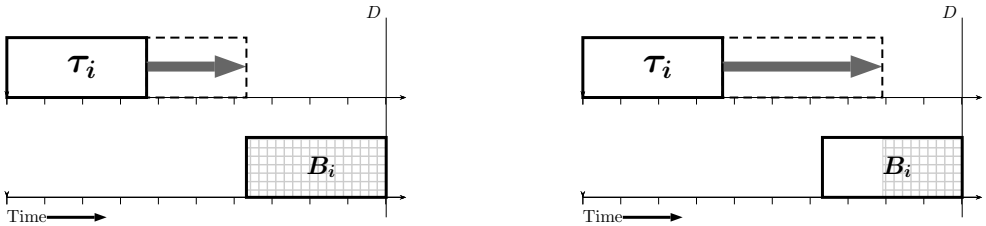
(a) Static Frequency Assignment

(b) Minimize Overlap (MO) scheme

(c) Overlap-Aware (OA) scheme

Figure 4.3: Executions under different schemes

On the spare core, the backups are executed at maximum frequency, and released as late as possible. Figure 4.3a shows the execution of the tasks in Table 4.2 under Static scheme. The tasks on the primary core are slowed down to the maximum level to save energy, which results in a large overlapped execution with the spare core, and ultimately yields an overall consumption value of 0.408 Joules.



(a) MO-scheme

(b) OA-scheme

Figure 4.4: Frequency assignment with MO and OA schemes

Minimize Overlap for Current Task (MO). In this scheme, when a main task

on the primary (τ_i) becomes ready to run, we attempt to choose a frequency that would allow the task to complete *before* its backup copy gets activated (Figure 4.4a). That is, $f_i = Min(f^{max}, f_i^*)$ where $f_i^* = \frac{C_i}{r_i - t}$, r_i is the latest time that the backup copy of τ_i can be activated without violating any deadlines, t is the current time, and f^{max} is the maximum frequency of the primary core. In case f_i is less than f_i^{ee} or f_U, we update f_i as $Max(f_i, f_i^{ee}, f_U)$.

This scheme makes sure that the activation of the current task's backup copy is avoided whenever possible. Figure 4.3b shows the execution of the tasks in Table 4.2 under this scheme. The task τ_1 runs at its maximum frequency in order to avoid any overlapped execution, which gives τ_2 enough time to avoid any overlap even when it executes at its energy efficient frequency ($f_2^{ee} = 0.29$). We have found that the overall energy consumption under this scheme is about 0.337 Joules—17.6% less than the one under Static.

Overlap-aware Energy Minimization (OA). The MO scheme opts to avoid an overlapped execution whenever it can (Figure 4.4a). However, we have found that in some cases, allowing some overlapped execution can yield better overall energy performance (Figure 4.4b). For instance, in a FasterP configuration, the energy gain due to additional slowdown on the primary *may offset* the extra energy consumption due to the overlapped execution on the power-efficient spare.

In this scheme, for a given task, first the energy consumption E_{MO} obtained with the frequency suggested by the MO scheme is computed. Then the frequency that minimizes the energy consumption of the main task and its backup copy, *under the assumption that the overlapped execution will happen,* is obtained, and the resulting energy consumption E_{OA} is evaluated. Finally the solution that yields the lower energy consumption between these two options is selected.

Specifically, the frequency f_i that minimizes the task's energy consumption *in case of overlap with the backup* is obtained by solving the following optimization problem:

$$\text{minimize} \quad E^{Pr}(f_i) + E^{Bk}(f_i)$$

$$\text{subject to:} \quad f_i^U \leq f_i \leq Min\{f_i^*, f_{max}\}$$

Above, the frequency f_i is restricted between two values, f_U which is the lowest frequency that guarantees the deadline, and f_i^* is the frequency that enables the task to complete at the start time of its backup copy (see the description of the MO scheme). $E^{Pr}(f_i) = P_i^{HP}(f_i)*(C_i^{HP}/f_i)$ is the energy consumed on the primary core, and $E^{Bk}(f_i) = P_i^{LP}(f_{max}^{LP})*(\frac{C_i^{LP}}{f_i} - (r_i - t)) + P_{idle}^{LP} * (r_i - t)$ is the energy consumed by the backup[1].

When there are multiple tasks, there is a need to assign execution windows to each of the tasks to determine the frequency suggested by the OA scheme. In fact, the slack time S (remaining time to finish all the incomplete tasks, which is $D - \sum C_i$) can be distributed among the incomplete tasks to determine individual (pseudo-) task deadlines. We found out that sharing the slack time based on the tasks' relative workload is a quite useful strategy. Therefore, each task τ_i is assigned a portion of the slack time, $S * \frac{C_i}{\sum C_j}$, and τ_i must finish execution within this time. The steps are repeated for each task iteratively.

Figure 4.3c shows the execution of the tasks in Table 4.2 under the OA scheme. τ_1 executes at low frequency, which results in some overlap with B_1. However, the increase in energy consumption due to the overlap is much less compared to the energy savings obtained by reducing frequency on the primary. This scheme yields an overall energy consumption of 0.263 Joules—35% and 22% improvement over Static and MO schemes, respectively.

4.1.2 Experimental Evaluation

We evaluated the performances of the proposed schemes by simulating the execution of large number of synthetic task sets in a discrete event simulator. The dual-core systems we simulated have a high-performance core with normalized frequency $f_{max}^{HP} = 1.0$ and a

[1]These equations assume a FasterP configuration.

low-power core with normalized frequency f_{max}^{LP} varying in the range $[0.7, 0.9]$.

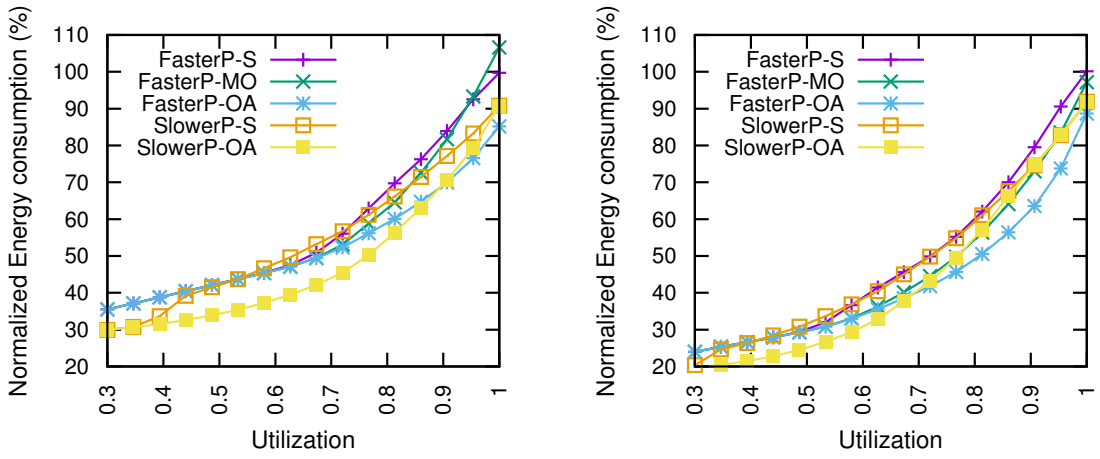
For a given target total utilization ($\sum \frac{C_i}{D}$), the number of tasks n, and a deadline D, we have used the *RandFixedSum* [214] algorithm to generate uniformly distributed individual task execution times ($\{C_i\}$ values) on the low-power core. The constants a_i^{HP} and α_i^{HP} are set to 1.0 and 0.1 respectively. Similarly, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$.

It is known that the execution time and power consumption on the high-power core for different tasks scale *by different ratios* when executed on a low-power core[56]. Therefore, for each task τ_i, we define a time-scaling factor $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, and a power-scaling factor $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$. The measurements reported in [56] suggest that $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(pscale_i * tscale_i) \leq 2.1$. We generated $tscale_i$ and $pscale_i$ values randomly in these ranges, thereby obtaining execution times and power characteristics of tasks on both types of cores.

The two core role assignment options (FasterP vs SlowerP), and three frequency assignment schemes (Static, Minimize Overlap (MO), and Overlap-Aware (OA)) give six different combinations shortened as **FasterP-S, SlowerP-S, FasterP-MO, SlowerP-MO, FasterP-OA** and **SlowerP-OA**. Each of the generated task sets is executed by all these six schemes and the results are reported (we compare the energy consumption of fault-free executions, as faults are very rare occurrences). The common frame period/deadline D is 100ms. For each data point shown in the plots, we computed the average of 3,000 task sets, each containing $n = 10$ tasks.

The trends indicate that for SlowerP configuration, practically in almost all cases, the Overlap-Aware scheme has chosen to minimize the overlap, as suggested by the Minimize Overlap (MO) scheme. This is to be expected, because creating an overlap with the back-up running on the high-performance spare core almost invariably hurts the energy savings in the SlowerP configuration. Therefore, the results we report will not show SlowerP-MO (whose performance is identical to that of SlowerP-OA). However, as we will see, FasterP-MO and FasterP-OA may perform quite differently.

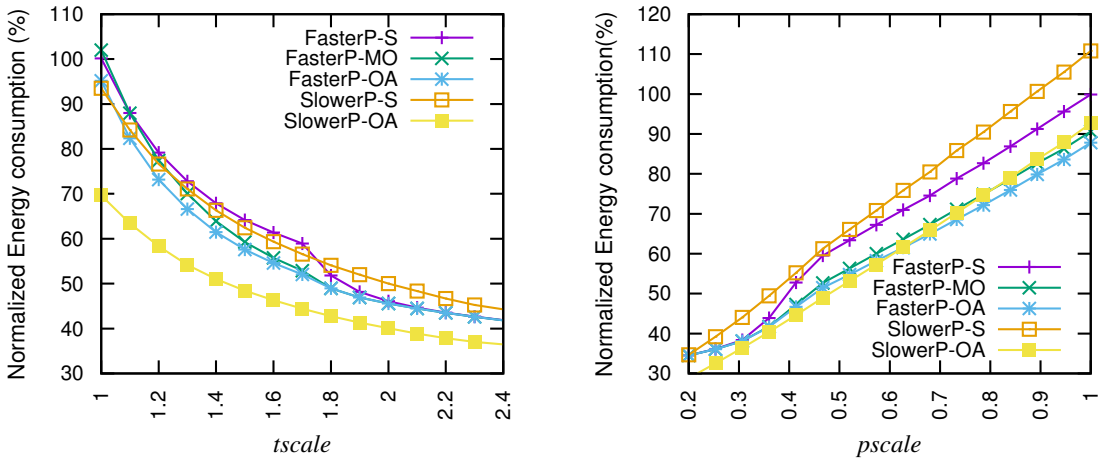
4.1.2.1 Impact of utilization



(a) $f_{max}^{LP} = 0.7$ (b) $f_{max}^{LP} = 0.9$

Figure 4.5: Impact of utilization

Figure 4.5a and 4.5b show the impact of system load on two different platforms with f_{max}^{LP} set to 0.7 and 0.9, respectively; $f_{max}^{HP} = 1.0$ in both cases. The X-axis shows the utilization with respect to the low-power core. The results are normalized with respect to the energy consumption of FasterP-S at $U = 1.0$. As expected, the energy consumption of all the schemes increases with increasing system load. In these plots, the OA (and MO) schemes in the SlowerP group perform best for low load settings, but FasterP-OA starts to outperform when the system load is heavy. For low-load, SlowerP is better because it can use the energy-efficient low-power core to execute all the primary workload, and the overlaps with the back-ups on the high-power spare can be mostly avoided. For heavy load, the situation changes – the spare core generally turns on earlier to meet the deadline requirements of backups. Since the spare always runs at maximum speed, having a slower, power-efficient core helps to reduce energy, and FasterP configurations prove more advantageous.



(a) Impact of *tscale*

(b) Impact of *pscale*

Figure 4.6: Impact of *tscale* and *pscale*

4.1.2.2 Impact of *tscale*.

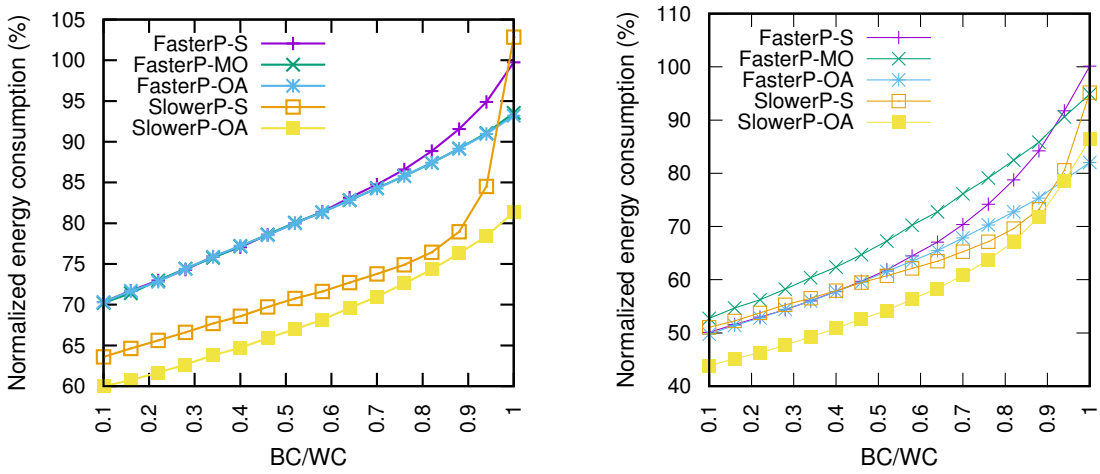
Figure 4.6a shows the impact of *tscale* for a moderately-loaded (62.5% on low-power core) system with $f_{max}^{LP} = 0.8$. It can be seen that for the entire region, SlowerP-OA performs distinctly better than others. This is because for this moderate load, the high-power backup core can be kept idle most of the time, and the low-power core will execute the workload avoiding backup activation. We can see that FasterP-S scheme performs worse for low *tscale* values, but improves as *tscale* increases. This is because higher *tscale* values imply a low-power core with large task execution times, and other techniques cannot do much to avoid significant overlapped executions.

4.1.2.3 Impact of *pscale*.

In general, low *pscale* values imply increased energy-efficiency of the low-power core. For these cases, SlowerP-OA perform better for moderate load (62.5% on the low-power core with $f_{max}^{LP} = 0.8$), as can be seen in Figure 4.6b. But as *pscale* grows, executing the main tasks on the faster core becomes less problematic from the energy standpoint, and at some

point FasterP-MO and FasterP-OA start to outperform the SlowerP group.

4.1.2.4 Impact of Workload Variability.



(a) System load = 62.5%

(b) System load = 87.5%

Figure 4.7: Impact of workload variability

The results reported so far correspond to the scenarios where every task takes its worst-case execution time. However, in practice, many tasks complete earlier, without presenting its worst-case workload. To investigate the impact of the workload variability, we define the ratio BC/WC as the ratio of *best-case execution time* to *worst-case execution time*. During the experiments, the actual execution time of every task is randomly generated between its worst-case and $BC/WC\times$ worst-case execution time, using a uniform distribution.

Figure 4.7a shows the impact of BC/WC ratio on a moderately loaded system (62.5% on the low-power core). With BC/WC, the energy consumption increases, because the system generates less and less dynamic slack that can be used to save energy. SlowerP-OA outperforms other schemes throughout the BC/WC spectrum. FasterP schemes stay worse than the SlowerP group except when BC/WC is very close to 1.0, at which point FasterP-MO and OA outperform SlowerP-S. In fact, FasterP schemes do outperform even

SlowerP-OA at high BC/WC values for highly loaded systems, as shown in figure 4.7b.

4.2 Energy-Aware Mixed Primary-Backup Scheduling on Heterogeneous Multicore Systems

In this section, we consider implementing a fault-tolerant framework using *mixed-primary/backup (MPB) scheduling* on heterogeneous dual-core systems while keeping the energy consumption at a minimum level. In contrast to Section 4.1, we do not designate the processing cores as *primary* or *backup*, instead, we employ "mixed-scheduling" in the sense that primary and backup copies of different tasks can now be allocated to the same processing core, with the requirement that the primary copy and the backup copy of a given task are always placed to different cores. Backup tasks are executed at the maximum frequency of the corresponding core, and they are cancelled as soon as the primary copy completes without error. This allows the system to tolerate the permanent fault of any single core, since each processor has exactly one copy of each task (primary or backup) [1]. Moreover, the transient faults detected in any primary task can be recovered from by the execution of the respective backup task. Since primary tasks can be placed on any available processing cores, this MPB approach enjoys much more flexibility in task-partitioning decisions which can be exploited towards energy-efficient operation. This research differs from the existing studies [53, 54], in that: i.) we allow scheduling a mix of primary and back-up tasks on each processor, and, ii.) we consider heterogeneous multicore systems.

To keep the energy consumption under control, the backup tasks are delayed as much as possible on their corresponding processors, because a backup can be canceled as soon as the corresponding primary completes successfully (i.e., without a fault). This also gives a chance to apply DVFS with maximum efficiency during the execution of the primary tasks on each core. We develop and propose schemes, i.) to partition all primary and backup tasks, and, ii.) assign frequency (speed) to all the primary tasks to minimize the energy consumption, while meeting timing and fault tolerance constraints.

Our experimental results suggest that the list-scheduling based partitioning techniques, coupled with a speed assignment approach that dynamically avoids the overlaps with the backups, exhibit superior performance which is close to the theoretical lower bound in terms of energy consumption. Our framework directly incorporates a salient feature of heterogeneous cores, namely the fact that the energy consumption and execution time figures of different tasks scale by different ratios when executed on different cores [56].

Problem Statement. Given a set of real-time tasks and a heterogeneous dual-core system, minimize the energy consumption by determining

1. The allocation of tasks such that the primary and backup copy of each task are assigned to different cores, and,

2. The processing frequency (speed) assignment to individual tasks.

In the rest of this chapter, we investigate these two interconnected dimensions and propose several efficient schemes.

4.2.1 Proposed Schemes

Before describing the specific algorithms that we propose, we present a number of general principles that guide our solution framework. To start with, in general, the concurrent execution of a primary task and its backup, though possible, is not desirable because it incurs the full energy cost of the back-up execution (Figure 4.8a). However, in case when the backup's execution can be delayed, by the time the primary completes successfully, its remaining part can be cancelled (Figure 4.8b)[2].

This further suggests that on a given core, all the primary tasks must execute *before* the backup tasks allocated to that core. Moreover, provisions are made to execute all *backup tasks* at the maximum frequency on their respective cores, should there be a need – obviously this choice minimizes their overlap with their respective primary tasks on the *other* core,

[2]Throughout the chapter, we show the cancelled part of the backup tasks by dashed patterns in all the figures.

(a) Execution with full primary-backup overlap

(b) Execution with partial overlap

Figure 4.8: Concurrent Execution of Primary and Backup Tasks

and in addition, since faults are rare events, the full speed execution of the backups has only a minimal impact on the average-case energy consumption. Clearly, this choice also leaves maximum slow-down opportunities for the primary tasks scheduled on that core through DVFS.

Thus, we define the *canonical execution order*, in which on a given core all primary tasks are started as soon as possible, whereas backup tasks are delayed as much as possible subject to the deadline constraints, and executed at the maximum frequency if needed. Figure 4.9 shows a *canonical execution* on a single processing core to which three primary tasks (τ_1, τ_2, τ_3), and two backup tasks (B_4 and B_5) are assigned. In the rest of the section, we commit to this canonical execution order to execute primary and backup copies of tasks on all cores, once the partitioning is done. The schemes we propose consist of *task partitioning*

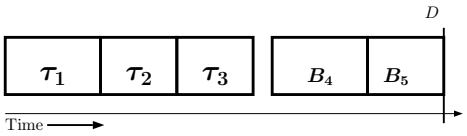


Figure 4.9: Canonical Execution Order

and *speed (frequency) assignment* phases which are described next.

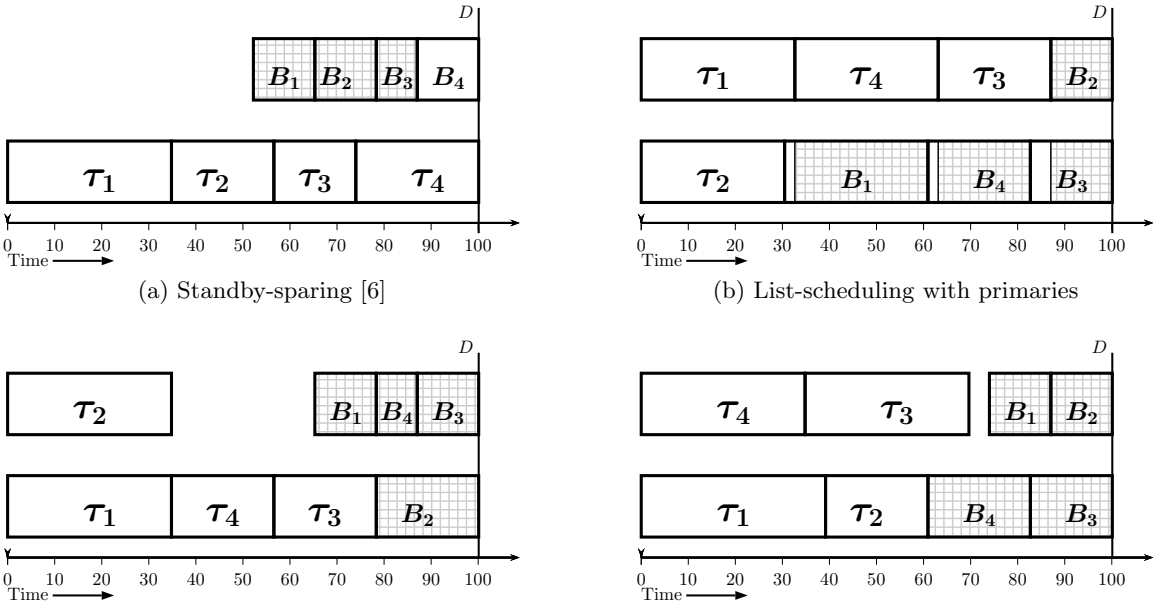


Figure 4.10: Task partitioning algorithms

4.2.1.1 Task partitioning

Task partitioning, in general, is an intractable problem; however, a well-known approach is based on the *list-scheduling* technique. We first describe two variants based on list scheduling for our task partitioning phase.

List-scheduling with Primaries (LSP). In this algorithm, we consider the primary copies of the tasks and employ list-scheduling algorithm to allocate them. First, the tasks are ordered according to their decreasing nominal utilizations. Then, each primary task is placed on a processing core that has the maximum *free capacity* after the placement. *Free capacity* on a core is defined by $(f_{max} - \sum_{\tau_i \epsilon \Gamma_p} \frac{C_i}{D})$, where Γ_p is the set of all primary tasks assigned to that core, augmented by the task under consideration. f_{max} and $\{C_i\}$ values are defined in the context of the core under consideration. Observe that the first few primary tasks will always go to the HP core, until its free capacity matches that of the LP core. Once the distribution of the primary tasks is complete, a backup copy for each primary task

is allocated to the alternate core. Also, at each stage of the primary task allocation, the feasibility of both cores, in terms of time constraints, are checked.

Table 4.3: Example Task Set 1

	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}
τ_1	13.2	30.4	14.63	5.58
τ_2	10.7	19.4	11.77	3.56
τ_3	10.6	18.8	11.66	3.45
τ_4	10.2	18.9	11.22	3.47

We illustrate the behavior of the algorithm on an example task set given in Table 4.3. The table gives task execution times (in ms), and energy consumption (E_i^{HP}, E_i^{LP}) on both cores (in mJ), under respective maximum frequencies. The 4-task set is scheduled on a dual core system with $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$, and for all tasks, $a_i^{HP} = 1.0$, $a_i^{LP} = 0.3$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.03$. For demonstration, we use a simple runtime policy (called *static* policy) in which, each primary task is slowed down as much as possible without violating the frame deadline. The *canonical execution order* is adopted on each core.

Figure 4.10b shows the task allocation under this scheme for our example task set in Table 4.3. The first task, τ_1 is allocated to the HP core, because it has the most free capacity among the two cores. τ_2 is allocated to the LP core whose free capacity is higher at that time. Similarly, tasks τ_3 and τ_4 are allocated to the HP core. It should be noted that, in contrast to the standby-sparing configuration shown in Figure 4.10a (which uses the partitioning method *SlowerP*, one of the best-performing scheme in [6]), the extent of primary-backup overlapped executions is much less in the LSP solution.

List-scheduling with Backups (LSB). This algorithm works in the same way as LSP, but this time, the backup copies of the tasks are considered while partitioning. Once the backup copies are distributed, their corresponding primary copies are allocated to the

respective alternate processing cores. By its very nature, this algorithm tends to allocate a few initial primary tasks to the LP core, before their backups are allocated to the HP core thanks to the LSB rule.

Figure 4.10c shows the task allocation under this scheme for our example task set in Table 4.3. This partitioning is a mirror image of the LSP partitioning. It can be noted that, all primary-backup overlapped executions are avoided.

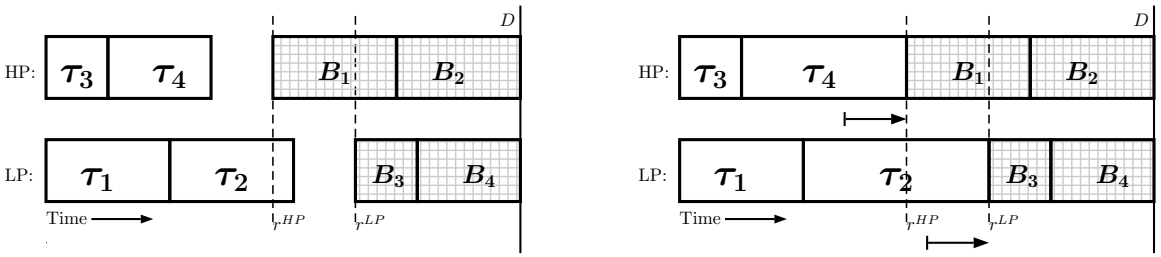
Fixed-Threshold Algorithm (FTH). In this algorithm, the primary tasks are at first ordered according to their decreasing nominal utilizations and processed one by one. Tasks are assigned to the LP core, as long as its load does not exceed a pre-defined *threshold* value. Otherwise the primary task is assigned to the HP core. After each primary task assignment, its backup copy is allocated to the counterpart core. The *threshold* value can assume any value between 0.0 and 1.0.

For our example Task Set 1, this heuristic produces the task-allocation shown in Figure 4.10d when the threshold value is 0.6. Tasks τ_1 and τ_2 are allocated to the LP core. When task τ_3 is processed, the total used capacity on the LP core exceeds 60% if it is assigned to the LP core. Therefore, it is assigned to the HP core. Similarly, τ_4 is allocated to the HP core.

4.2.1.2 Speed assignment

Once the task partitioning phase is complete, the next step is to determine the speed (frequency) of the primary tasks on each core, while committing to the canonical execution order. Speed assignment to the primary tasks is critical not only because it determines directly the primary's energy consumption, but also indirectly, that of the corresponding backup whose overlap extent may change as a result of that assignment. Below we propose three speed assignment policies.

Static Speed Assignment (SSA). Figure 4.11 illustrates the basic principles of the SSA policy. The scheme reserves capacity for each allocated backup task (which runs at the maximum frequency of the core), and assigns a latest-start-time to each of them such



(a) Initial partitioning

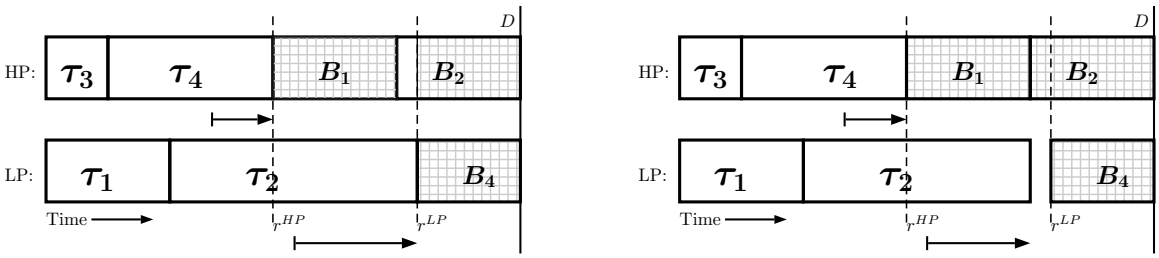
(b) After speed assignment

Figure 4.11: Static Speed Assignment

that no deadlines are missed. In Figure 4.11a, r^{HP} and r^{LP} denote the latest start time for the first backup task on the HP and LP cores, respectively. Primary tasks are slowed down as much as possible, subject to the energy-efficient frequency bound (f_{ee}). Letting r denote the start time of the first backup task on a specific core, and Γ_P denote the set of all primary tasks on that core, then, the common frequency that finishes all these primary tasks before time r is given by $f_U = (\sum_{\tau_i \epsilon \Gamma_P} C_i)/r$. Then, each primary task τ_i is assigned the frequency $f_i = Max(f_i^{ee}, f_U)$. Figure 4.11b shows the extended execution times for primary tasks, derived through this principle.

Dynamic Backup Cancellation (DBC). In this scheme, as in SSA, the processing capacity is reserved for backup tasks and primary copies are slowed down as much as possible, subject to the energy-efficient frequency. However, the speed assignment routine is re-invoked at runtime: each time a primary task completes without fault, the reserved capacity for its backup copy is deallocated and used to further slow down the next primary tasks on that core. For example when τ_3 finishes without error, the reserved capacity for B_3 on the LP core is reclaimed to further slow-down τ_2 (Figure 4.12a). Note that, this introduces some overlapped execution for B_2. In general, when task τ_i is about to run at time t, its speed is chosen as $f_U = (\sum_{\tau_i \epsilon \Gamma_x} C_i)/(r - t)$, where Γ_x is the set of unfinished primary tasks on the same core, and r represents the earliest start time among the unfinished backup tasks, again on the same core. When a primary task completes without error, the

earliest backup activation time on the alternate processing core is updated at runtime. The chosen speed value is subject to the energy-efficient frequency, therefore, for each task τ_i, the speed is set to $f_i = Max(f_i^{ee}, f_U)$.



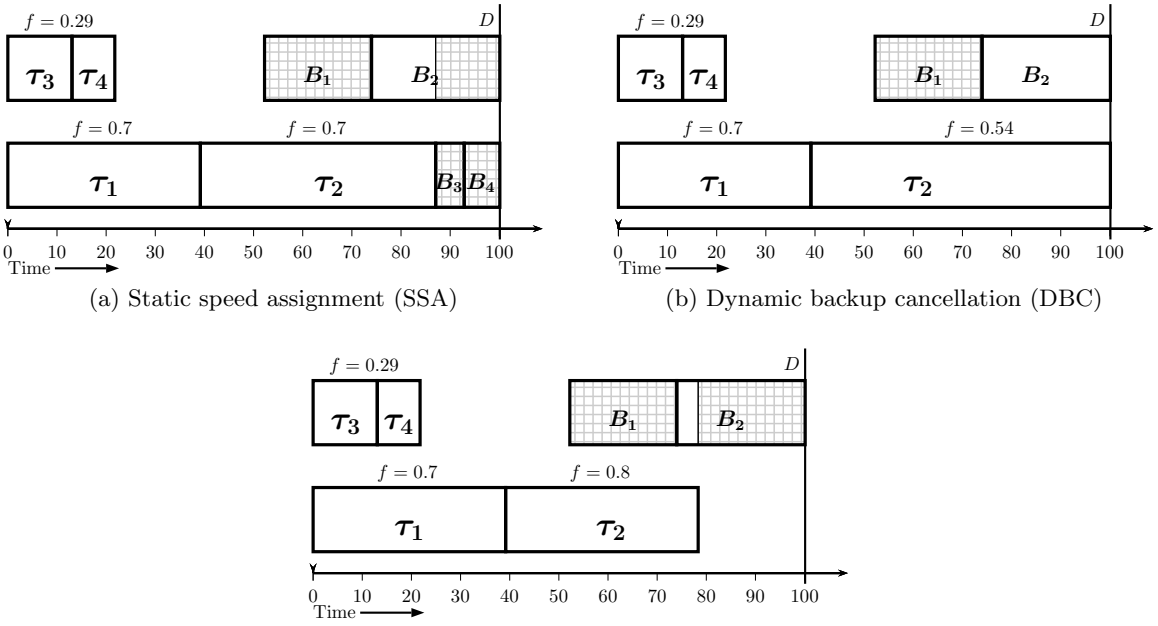
(a) Dynamic Backup Cancellation

(b) Dynamic Backup Cancellation with Minimum Overlap

Figure 4.12: Dynamic Policies

Dynamic Backup Cancellation with Minimum Overlap (DMO). This scheme works as the DBC scheme; but when setting the speed of the primary tasks at run-time, it attempts to minimize the overlapped-execution with back-ups. As shown in Figure 4.12b, when DVFS is applied to τ_2 at the beginning of its execution, it is not maximally slowed down; instead, the overlapped execution with B_2 is avoided by running somewhat faster than the DBC policy. Under this policy, the speed of τ_i is chosen to be $f_i = Min(f^{max}, f_i^*)$ where $f_i^* = \frac{C_i}{r_i - t}$, where r_i is the latest time the backup copy of τ_i can be activated (on the alternative core) without violating any deadlines, and t is the current time. This speed is subject to the deadline constraint and the energy-efficient speed, therefore, f_i is updated as $f_i = Max(f_i, f_i^{ee}, f_U)$. In this scheme, f_U is re-computed with a dynamically updated r value as in the DBC scheme.

To contrast the impact of these schemes, we use the 4-task set in Table 4.4 with $a_i^{HP} = 1.0$, $a_i^{LP} = 0.3$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.03$ for each task. The task set is executed on a dual core system with $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$. Figure 4.13a shows the execution of the task set under LSB partitioning and static speed



(a) Static speed assignment (SSA)

(b) Dynamic backup cancellation (DBC)

(c) Dynamic cancellation with minimum overlap (DMO)

Figure 4.13: Execution under different schemes

assignment. The HP core (at the top) uses the energy-efficient frequency for tasks τ_3 and τ_4, and the LP core (at the bottom) is slowed down maximally ($f = 0.7$) so that all backup copies (B_3 and B_4) can make their deadline. The overall energy consumption is 24.7 mJ.

Figure 4.13b shows the execution of the same task set under LSB partitioning and DBC policy. The scheme reclaims the reserved capacity for the backup copies B_3 and B_4 whose primaries complete without fault, and uses this capacity to further slow down the primary task τ_2 to speed $f = 0.54$. However, this introduces overlapped execution for B_2, and in this case, hurts the energy savings. The overall energy consumption of this system is 36.7 mJ. Finally, Figure 4.13c shows the execution under LSB partitioning and DMO runtime policy. Although this scheme could use all the reclaimed capacity from B_3 and B_4, it runs τ_2 at the maximum speed of the LP core ($f = 0.8$) to minimize the overlap with B_2. This execution yields an overall energy consumption of 20.2 mJ, which is 18% lower than that of the static policy.

Table 4.4: Example Task Set 2

	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}
τ_1	20.3	36.8	22.33	6.76
τ_2	19.1	39.4	21.01	7.23
τ_3	4.3	10	4.73	1.84
τ_4	1.5	3.02	1.65	0.55

4.2.2 Experimental Evaluation

We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. We simulated dual core systems with $f_{max}^{HP} = 1.0$ and f_{max}^{LP} varied from 0.6 to 1.0. For conciseness, we will show the results for $f_{max}^{LP} = 0.8$, and analyze the impact of varying f_{max}^{LP} separately in Section 4.2.2.3.

It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [56]. Therefore, as in section 4.1, we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which models how execution time changes on the LP core for a given task, τ_i. Moreover, following section 4.1, we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as $\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$.

For each experiment, the simulator generates a task set containing n tasks, and a given total utilization, U. The utilization value is calculated with respect to the LP core (which is more constrained in terms of performance) and normalized considering its maximum speed. Hence, $U = (\sum \frac{C_i^{LP}}{D})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [214] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U. We set the frame deadline $D = 100ms$. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within ranges suggested in [56]. Specifically, $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ hold. We assume for

all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. In addition, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$ for all experiments.

Each generated task set is partitioned upon the HP and LP cores according to one of the proposed partitioning algorithm. For every partition obtained in this way, we simulate the execution according to the speed assignment policies that we suggested, and record the energy consumption. Every combination of a partitioning scheme and a speed assignment algorithm gives us a valid overall algorithm, whose name is indicated by the concatenation of the member schemes (e.g., LSP-SSA, FTH-DMO). We use task sets with $n = 10$ in all the results shown, but we discuss the impact of varying the number of tasks in Section 4.2.2.3. Every reported data point is the average of 3000 runs.

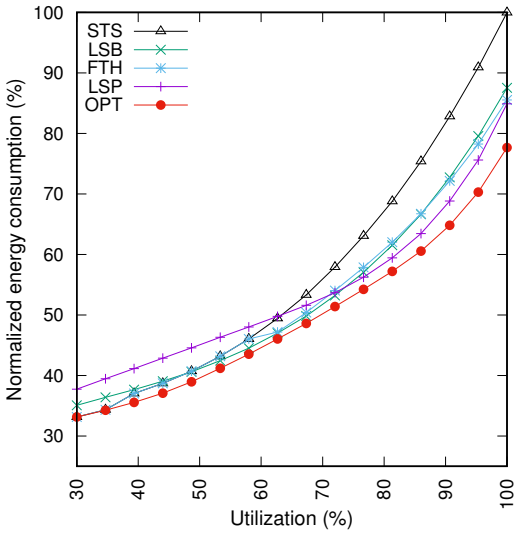
We report the average energy consumption in fault-free executions, since faults are very rare events. The obtained energy consumption numbers are normalized with respect to the maximum energy consumption (observed in the considered parameter spectrum) of a standby-sparing system with static speed assignment and in which all the primary copies are allocated to the LP core [6].

Due to the multiple dimensions of the problem and large number of scheme combinations, in our evaluation, we will adopt a hierarchical approach. We will first discuss the performance of the partitioning algorithms by fixing the speed assignment policy. Next, we will compare the performance of the proposed speed assignment policies, and also investigate the impact of the chosen threshold value on the FTH algorithm. Finally, we show the effect of the maximum speed of the LP core and the effect of the number of tasks.

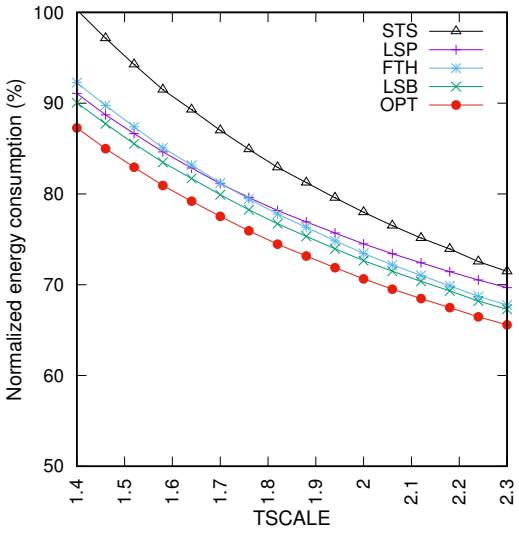
4.2.2.1 Evaluation of Partitioning Algorithms

We implemented the following partitioning schemes in our simulator:

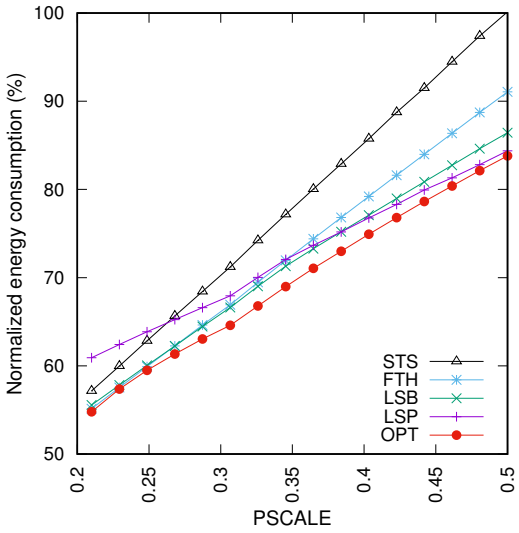
- List-scheduling with Primaries (LSP)
- List-scheduling with Backups (LSB)
- Fixed-threshold Algorithm (FTH)



(a) Impact of utilization



(b) Impact of tscale



(c) Impact of pscale

Figure 4.14: Performance of partitioning algorithms

- Standby-sparing (STS)
- Optimal Partitioning (OPT)

The optimal partitioning we show in the plots is obtained by exhaustively enumerating all possible task allocations, and measuring their runtime energy consumption, then choosing

the best. This is implemented by the exhaustive search which becomes impractical when the number of tasks grows beyond 15. The STS algorithm is adopted from the *SlowerP* scheme in [6], because it is shown to be the best-performing one in its respective context. The threshold value for the FTH algorithm is fixed as 0.6. The energy consumption of the partitioning algorithms is shown using the static speed assignment algorithm (SSA); we obtained similar trends with the other (DBC and DMO) algorithms.

Impact of Utilization. In Figure 4.14a, we show the impact of utilization on normalized energy consumption. When the utilization is low, the FTH algorithm's performance approaches the optimal one, suggesting that allocating all primary tasks to the LP core, and all the (delayed) backups to the HP core is the best strategy. This is because under low load, LP can finish the primary workload quickly and in a power-efficient way, allowing the backup tasks to get cancelled on the HP core early. This is evident for the STS scheme too, because it allocates all the primary workload to one core as well. As the load increases, FTH drifts from the optimal scheme and LSB becomes a comparable scheme. This is due to the fact that, as the load grows, a more balanced partitioning is preferable which can allow a suitable distribution of the reserved space for backup copies such that their activation is seldom needed. Both FTH and LSB give relatively balanced partitionings, but LSB generally allocates more primary copies to the LP core, with an energy advantage. LSP scheme, performing very poorly on the low-load case, starts to outperform both LSB and FTH when the utilization exceeds 80%, and comes within 5% of the optimal scheme. For heavy load, executing primary copies on the HP core is preferable because in this case, the backup copies cannot, in general, get cancelled and executing them at the maximum speed of the LP core is preferable to executing them at the maximum speed of the HP core. For the same reasons, STS performs the worst for heavy load cases.

Impact of tscale. Figure 4.14b shows the impact of *tscale* on the performance of the partitioning algorithms. *tscale* is varied within the range of 1.4 to 2.3, which is obtained from [56]. In general, larger *tscale* values indicate that tasks take much longer to complete on the LP core, despite its power-efficiency. In these experiments the utilization is fixed

at 70%, and therefore, increasing *tscale* implies additional unused capacity on the HP core. We see that LSB performs consistently within 3% of the optimal scheme throughout the entire range of *tscale*. This is because executing the primary copies of the workload on the power-efficient core results in less energy consumption, and LSB tends to allocate primary workload to the LP core. LSP, on the other hand, has a tendency to assign primary workloads to the HP core, and in general, it lags behind LSB. FTH comes very close to the performance of LSB as *tscale* increases.

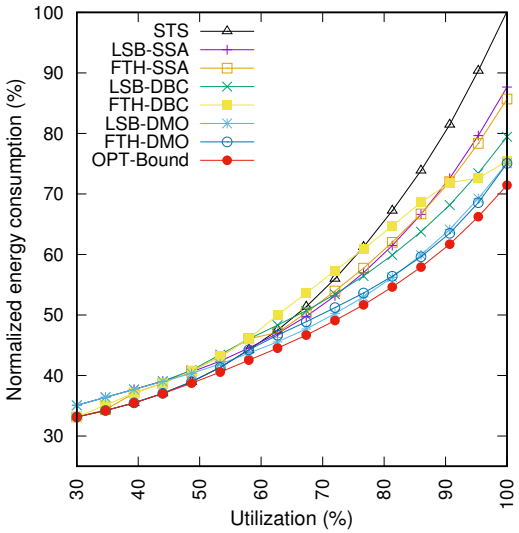
Impact of pscale. Figure 4.14c shows the impact of *pscale* on the performance of the partitioning algorithms. When the LP core is very power-efficient, i.e., *pscale* is low, FTH and LSB come very close to optimal scheme. This is because at the fixed 70% system load, FTH assigns most of the primary workload on the LP core, and that helps saving energy. As *pscale* grows, FTH drifts away from the optimal scheme the most, because it is no longer efficient to use the LP core for most of the primary workload. However, LSB can still perform within 5% of the optimal scheme, because it produces a more balanced partitioning with a bias to allocate the primary tasks to the LP core. LSP, which produces a balanced partitioning with a bias to assign the primary tasks to the HP core, performs poorly for low *pscale*, but starts to outperform LSB for *pscale* greater than 0.4 and comes 2% of the optimal scheme.

4.2.2.2 Evaluation of Speed Assignment Algorithms

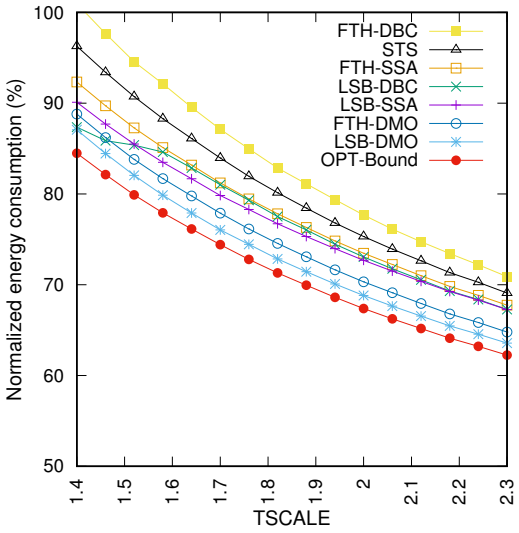
We implemented the following speed assignment policies.

- Static Speed Assignment (SSA)
- Dynamic Backup Cancellation (DBC)
- Dynamic Backup Cancellation with Minimum Overlap (DMO)
- Bound

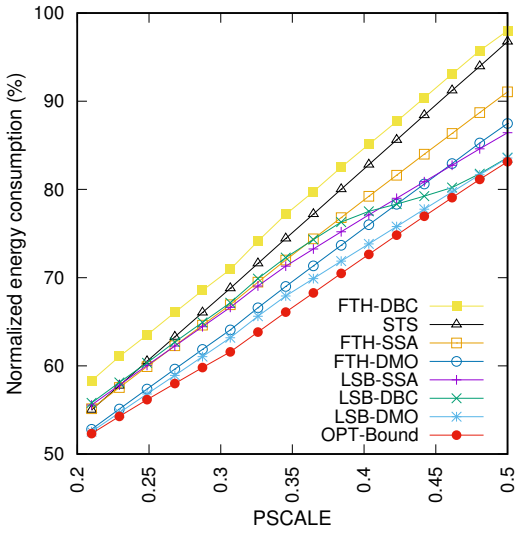
The *Bound* algorithm is implemented as a yardstick speed assignment algorithm. After partitioning the tasks the executions slots are still reserved for backup tasks – those slots



(a) Impact of utilization



(b) Impact of tscale



(c) Impact of pscale

Figure 4.15: Performance of the speed assignment algorithms

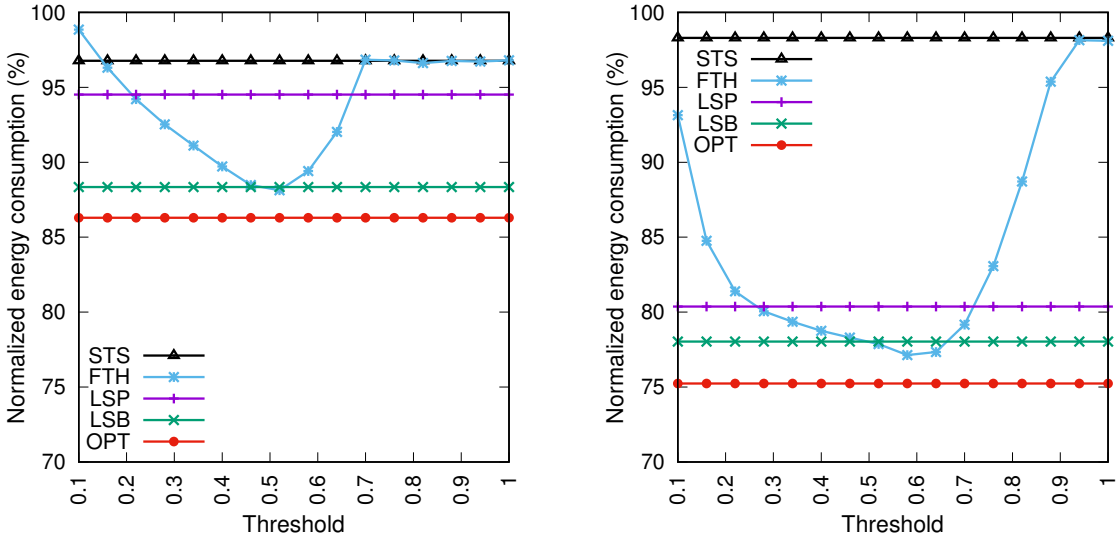
are dynamically released (as in DBC), but no extra energy consumption is recorded for the overlapped execution of the backup tasks at run-time. Since the backup executions essentially incur zero energy cost, no speed assignment algorithm can outperform *Bound*.

We matched *Bound* with the exhaustive search based *Optimal* partitioning algorithm, obtaining a combined scheme denoted by *OPT-Bound*, which gives the lower bound on the performance of any realistic MPB algorithm. Given the large number of partitioning/speed assignment scheme combinations, for other schemes, we are showing only the results we obtained with the best performing partitioning algorithms, namely LSB and FTH. We are using the *Overlap-Aware* speed assignment scheme for STS, as it is shown to be the best performing scheme for standby-sparing in [6].

Impact of Utilization. In Figure 4.15a, we see that both FTH-DMO and LSB-DMO perform within 2% of Opt-Bound. This is because dynamically reclaiming the capacity for backup tasks and minimizing overlap while applying DVFS is a very effective strategy, as done within DMO. This is also true for STS at low-load, because it allows some carefully calculated overlapped execution. As the load increases, STS drifts away from Opt-Bound the most, because it has the restriction that it cannot allocate primary and backup copies on the same processor. FTH-DBC and LSB-DBC perform poorly for moderately loaded systems due to the large overlapped executions that it creates. However, for heavy load, backup copies need to run until deadline anyway, therefore the performance of the DBC scheme improves. Both FTH-SSA and LSB-SSA offer decent performance levels unless the load is very high.

Impact of tscale. As we change *tscale* value (when the load is fixed at 70%), LSB-DMO performs the best and stays within 3% of Opt-Bound (Figure 4.15b). The next best performing scheme is FTH-DMO. This again suggests the superiority of DMO thanks to its dynamic but moderately aggressive approach in applying DVFS while avoiding overlaps. The plot also shows that FTH-DBC performs the worst, and LSB-DBC performs the worst among all the LSB algorithms. This is because DBC aggressively slows down a task without regard to the overlapped execution.

Impact of pscale. Varying *pscale* yields similar trends (Figure 4.15c). LSB-DMO and FTH-DMO perform the best, within 3% of *Opt-Bound*, by exploiting the overlap avoidance strategy of DMO. LSB-DMO's performance, however, decreases as the LP core becomes less



(a) Utilization = 70%

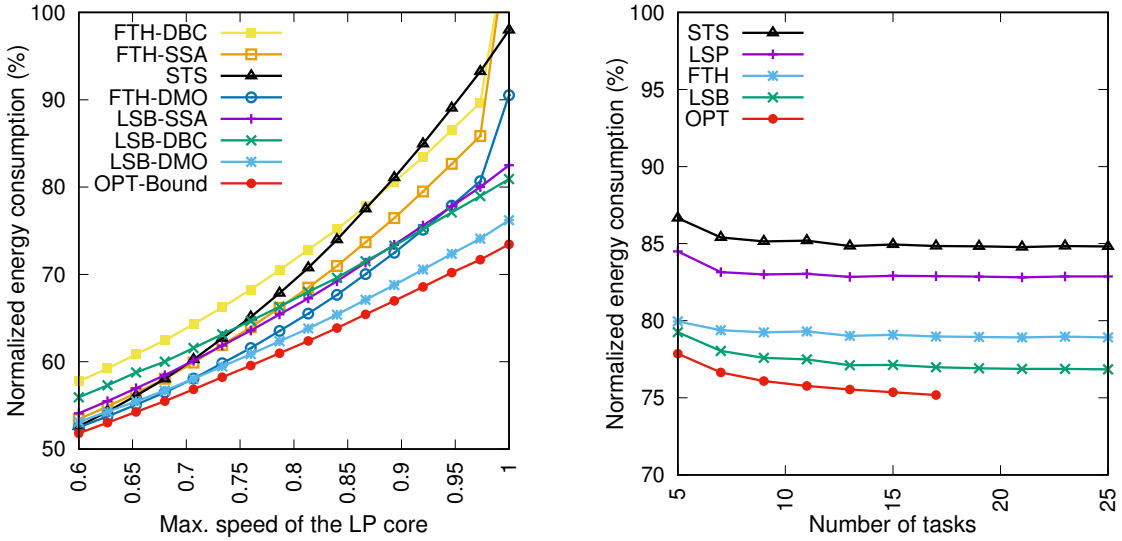
(b) Utilization = 90%

Figure 4.16: Impact of threshold value in *FTH* algorithm

power-efficient (*pscale* increases). This is because, with less power-efficient LP core, it is no longer favorable to assign primary workload to the LP core up to a threshold. Due to the aggressive frequency scaling of the DBC scheme, FTH-DBC performs the worst throughout the entire spectrum. LSB-DBC, performing poorly for low *pscale*, starts to improve when *pscale* is greater than 0.4, and comes within 1% of Bound. This is because when the LP core is less power-efficient, slowing it down as much as possible proves helpful from the energy consumption perspective.

4.2.2.3 Additional Results

Impact of the threshold value in the FTH algorithm. The Fixed-Threshold (FTH) algorithm works by allocating all primary tasks to the LP core until a *threshold* utilization is reached. Figure 4.16a shows the impact of the threshold value on a system that is 70% loaded and with DMO policy. The results indicate that the energy consumption of *FTH* decreases as we increase the threshold value, and at about 0.45, it outperforms the otherwise best performing algorithm, LSB. Its energy consumption is minimized at some threshold value



(a) Impact of the maximum speed of the LP core

(b) Impact of the number of tasks

Figure 4.17: Additional Evaluations

around 0.50. The energy consumption goes up as we increase the threshold and becomes constant at some point – because when the *threshold* value exceeds the utilization, all of the workload is assigned to the LP core. The threshold-independent algorithms, naturally yield a constant energy consumption. Figure 4.16b shows a similar pattern for a system with 90% load. The results suggest that choosing a threshold value in the range [0.5, 0.6] is generally a very good choice when using the FTH algorithm.

Impact of the maximum speed of the LP core. In this set of experiments, we varied the maximum speed of the LP core while fixing the load at 70% for each configuration (Figure 4.17a). The performance of LSB-DMO remains within 5% of Opt-Bound for the entire region, suggesting that it is applicable in a wide range of heterogeneous systems. FTH algorithms, on the other hand, tend to drift away from Opt-Bound as the maximum speed of the two processing cores become close to each other. We also see that the energy consumption of all schemes increases with increasing f_{max}^{LP}. This is because, when the utilization is kept fixed at 70%, when we increase f_{max}^{LP}, the effective amount of workload

on the system is increased, which is reflected in the results.

Impact of the number of tasks. Figure 4.17b shows the impact of number of tasks for a system with utilization 70%. We see that for small number of tasks, the performance of all the schemes is affected. As the number of tasks grows, the average task size decreases and the performances of various schemes stabilize. We can see that LSB performs within 3% of the optimal scheme, and FTH is about 3% worse than LSB, for the entire region. LSP performs worse than the other two, but it also shows stable performance when the number of tasks grows. For the optimal scheme, we could only calculate energy consumptions for up to 17 tasks due to its prohibitive computational complexity.

4.3 Concluding Remarks

In this chapter, we presented our studies on the fault-tolerant scheduling of independent frame-based real-time tasks implemented upon heterogeneous multicore processors with a goal of minimizing the overall energy consumption of the system. To ensure reliability, we associated each task with a backup copy, and we always placed the primary and backup copies of the same task to different processing cores. For energy management, we employed DVFS and DPM to slow down execution and idling the processing cores respectively, while providing strict guarantees on the deadline constraints. We implemented standby sparing technique on heterogeneous multicores and proposed several algorithms to minimize the overall energy consumption. Furthermore, we employed mixed-primary/backup approach which allows additional flexibility in allocating tasks to processors, and we proposed a few algorithms that can exploit this feature to yield energy-minimal operation. We conducted extensive simulation experiments which show the superior performance of our proposed schemes.

Chapter 5: Energy-aware Fault-Tolerant Real-time Scheduling of Frame-based Tasks with Precedence Constraints

In this chapter, we propose a fault-tolerant and energy-efficient framework for frame-based real-time tasks with precedence constraints executing on heterogeneous dual-core systems. In many practical real-time applications, there are often dependency relationships among real-time tasks [29, 215, 216], which generally stems from the input/output data dependencies between them. As an example, consider three tasks τ_x, τ_y and τ_z. If τ_z takes the output of τ_x and τ_y as input, then τ_z can only start to execute after both τ_x and τ_y have finished execution and made their output data available. We model the dependencies using precedence constraints as directed acyclic graphs (DAGs) [217, 218]. An example DAG is shown

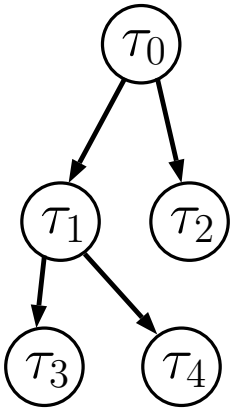


Figure 5.1: An Example Task Graph (DAG)

in Figure 5.1. An edge $\tau_j \rightarrow \tau_k$ in the DAG indicates that τ_k can only start to execute when τ_j completes successfully. A common approach to scheduling tasks with precedence

constraints is to process a *topological ordering* of the tasks. A topological ordering is an order of the tasks which ensures, if $\tau_i \rightarrow \tau_j$ is in the task graph, then τ_i will precede τ_j in the given order. For the example in Figure 5.1, one possible topological order is τ_0, τ_1, τ_2, τ_3 and τ_4. As a naive method, we can schedule the tasks in this order using standby sparing or mixed-primary/backup technique presented in Chapter 4, however, this arrangement fails to provide proper fault tolerance when multiple tasks are affected by transient faults during a given frame. Previously, the primary and backup copies could be scheduled independently in time to each other, but in this case, if a task is affected by a transient fault, then none of its subsequent tasks can start until its backup has completed successfully. Moreover, after the first recovery, each of the subsequent tasks is susceptible to subsequent transient faults, and therefore a backup for each subsequent task must be scheduled as well. Another approach to scheduling dependent tasks is to allocate each task's primary and backup copy in pairs to the available two processing cores, following the topological order. However, this approach is deficient for two reasons: i.) this limits any parallel execution of two primary tasks which do not have any precedence relationship, and ii.) this limits the opportunity for energy saving because the backup tasks cannot be maximally delayed. We designed algorithms and techniques by exploring how to exploit the parallelism in the heterogeneous cores as well as its power-performance characteristics using DVFS and DPM such that the overall energy consumption is minimized.

Our proposed solution has two components: 1.) A main schedule where tasks are executed at low processing frequency (speed) levels using DVFS to save energy as long as faults are not encountered, and, 2.) The *contingency schedule* according to which recovery tasks are executed upon the detection of transient and/or permanent faults. Task partitioning and speed assignment algorithms are designed by respecting the precedence constraints and allowing the necessary recovery times in the contingency schedule, while minimizing energy consumption in most common (fault-free) execution scenarios. The fault model we assume can tolerate a separate transient fault for each real-time task, as well as the permanent fault of any single core – in fact, the system can recover from the permanent fault of a processing

core, even after multiple tasks have incurred transient faults and have been re-executed thanks to the hardware and time redundancy offered by the contingency schedule. All the components of the proposed framework guarantee the precedence and timing constraints. The experimental evaluation suggests that our proposed schemes can offer non-trivial energy gains over a broad parameter spectrum. To the best of our knowledge, this is the first study on the energy-aware fault-tolerant operation of real-time tasks with precedence constraints, executing on heterogeneous multicore systems.

Our proposed framework, as discussed in Section 5.1.1, includes copies of each task τ_i to be executed in a potential recovery mode, upon the detection of run-time faults. If a transient or permanent fault is detected, the system switches to the *recovery mode* and executes all the incomplete tasks at the maximum speed according to a *contingency schedule*, whose details are provided in Section 5.1.1. In essence, the contingency schedule allows re-executing faulty tasks and also tolerating additional transient faults that may affect other tasks. Moreover, it offers the capability to recover from a permanent processing core fault that may occur after any number of transient faults. Note that, should a permanent fault occur, the system loses the capability of tolerating any more (transient or permanent) faults until the faulty core is repaired or replaced. In the rest of this chapter, we present the details of our proposed framework and experimental results.

5.1 Proposed Framework

Our proposed framework has two complementary phases for task mapping and scheduling. First, since the faults are rare events, there is a need to determine the default schedule according to which tasks are executed in each frame as long as faults are not encountered. We call this default schedule the *main schedule*. An important objective for the main schedule is to minimize energy consumption in the most common (i.e., fault-free) frame execution scenarios. Consequently, computing the main schedule involves:

- Allocating the primary tasks on the HP and LP cores and determining their execution

order (*task partitioning and ordering*), and,

- Determining the voltage/frequency levels for individual primary tasks (*speed assignment*),

to minimize energy consumption while meeting the precedence and timing constraints. However, while generating the main schedule, it is necessary to take provisions to tolerate transient and permanent faults in a timely manner. In what follows, we first discuss the *recovery mode* of execution and the derivation of the *contingency schedule* according to which the tasks are (re-)executed in a frame upon the detection of a fault. Then we elaborate on the two components of the main schedule computation (task partitioning and speed assignment).

5.1.1 Recovery Mode and Contingency Schedule

When a transient fault is detected at the end of task τ_i, the task must be re-executed, in addition to other tasks that are yet to be executed before the end of the frame (deadline), while satisfying the precedence constraints. In our framework, upon the detection of a fault, the system switches to the *recovery mode* and executes the (incomplete) tasks according to a pre-computed *contingency schedule*.

It should be noted that, we could not simply schedule a backup copy on the alternative processing core (as our approach in Chapter 4), because due to the precedence constraints, this approach fails to provide fault tolerance when multiple transient faults affect multiple tasks during the same frame. In order to tolerate multiple transient faults, it is imperative to execute the backup copy of the faulty task along with both copies of all subsequent tasks.

As long as both cores are functional, the system must preserve its capability to recover from the permanent fault of any of the cores. Since this invariant must hold even during the recovery mode which may have been triggered by a transient fault, in the contingency schedule, it is necessary to schedule two distinct copies of tasks allocated on two different cores; with only one copy of a task τ_j allocated on a specific core, it would not be possible

to re-execute τ_j, should that core experience a permanent fault.

Consequently, in the contingency schedule, associated with each task τ_i there are two contingency tasks[1] ρ_i and ρ'_i with the exact same timing parameters as those of τ_i. We make sure that ρ_i and ρ'_i are allocated to different cores, as a precaution against a permanent fault. When a primary task τ_i starts to execute, the contingency task ρ_i is cancelled, and if the primary task τ_i completes successfully, the contingency task ρ'_i is cancelled; hence they do not incur any time or energy overhead in most common fault-free execution scenarios. Thus, the contingency schedule consists of a sequence of paired contingency tasks executing at the maximum speed of their respective cores, if the system enters the recovery mode for that frame. Moreover, their executions are delayed as much as possible to minimize overlaps with the tasks in the main schedule (see Fig. 5.2c as an example).

Specifically, once the primary tasks are mapped to the HP and LP cores, the contingency schedule is determined according to the following rules:

1. A topological order of tasks satisfying the precedence constraints implied by DAG is obtained. In addition, this task sequence complies with the execution order of tasks observed on each core in the main schedule.

2. Two contingency copies of each task (ρ_i and ρ'_i) are placed in parallel on the HP and LP cores, according to the order derived in Step 1. ρ_i is placed on the same core as its primary copy in the main schedule, whereas ρ'_i is placed on the alternative core.

The contingency tasks are shifted towards the deadline as much as possible such that each primary copy can get a larger execution-window and run at a slower speed by applying DVFS in order to save energy (Fig. 5.2c). The activation times of the contingency tasks are computed such that both copies complete at the same time with their respective worst-case execution time on the HP and LP cores. These activation times represent the latest start time of a contingency copy such that any faulty task and all of its subsequent tasks

[1]Note that, as opposed to our general system model which has a single backup task B_i, in this chapter we are using two tasks ρ_i and ρ'_i instead as contingency copies.

can be executed before the frame deadline if needed. The pseudocode for constructing the contingency schedule is given in Algorithm 1.

Algorithm 1 Schedule-Contingency

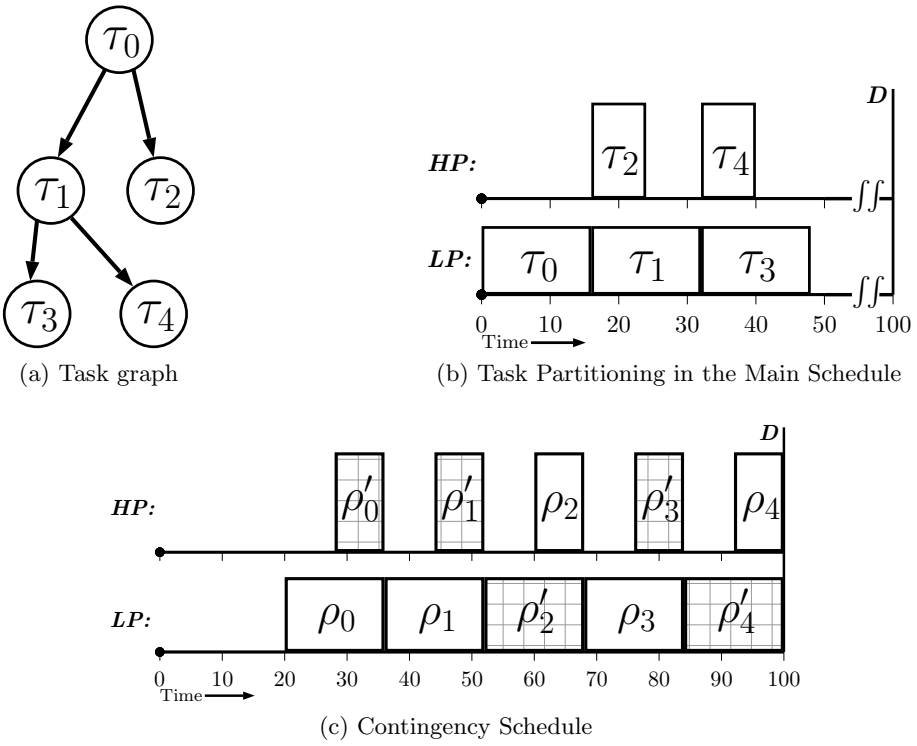
Input: Task set (Γ), Allocation Task Graph (G), Deadline (D)
Output: Latest activation times $\{start_{\rho_i}\}$ and $\{start_{\rho'_i}\}$
Create a list λ by obtaining a topological order from G, breaking ties by choosing largest (C_i^{HP}) task first
$\lambda \leftarrow Reverse(\lambda)$ /* Reverse the list λ */
$time \leftarrow D$
for each task τ_i in λ **do**
  if τ_i is allocated to HP **then**
    $Z \leftarrow HP$, $Z' \leftarrow LP$
  else
    $Z \leftarrow LP$, $Z' \leftarrow HP$
  end if
  $start_{\rho_i} \leftarrow time - C_i^Z$
  $start_{\rho'_i} \leftarrow time - C_i^{Z'}$
  $time \leftarrow time - Max\{C_i^{HP}, C_i^{LP}\}$
end for
return $\{start_{\rho_i}\}$ and $\{start_{\rho'_i}\}$

As an example, consider a set of tasks given by the DAG shown in Fig. 5.2a and deadline $D = 100ms$. For each task τ_i, we have $C_i^{HP} = 8$ and $C_i^{LP} = 12$, expressed in millions of cycles. Assuming $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.75$ GHz, therefore, each task takes 16 ms and 8 ms when executed on the LP and HP cores at maximum speed, respectively. Considering the allocation of the primary tasks shown in Fig. 5.2b, we show an example contingency schedule in Fig. 5.2c. First the topological order τ_0, τ_1, τ_2, τ_3, τ_4 is obtained and the corresponding contingency tasks are maximally pushed towards the deadline.

In summary, the system operates according to the following rules at run-time:

R1. As long as there are no faults, the system continues with the main schedule. When a primary task τ_i completes successfully, the contingency schedule is updated to reflect the cancellation of the contingency tasks ρ_i and ρ'_i.

R2. If a fault is detected at the end of τ_i, the system immediately transitions to the recovery mode, and it starts executing the contingency copies of all incomplete tasks at the maximum speed of both cores according to the contingency schedule, including



(a) Task graph

(b) Task Partitioning in the Main Schedule

(c) Contingency Schedule

Figure 5.2: An Example Contingency Schedule

ρ_i and ρ'_i. Once the end of the frame is reached, the system resumes the execution according to the main schedule in the new frame, at low processing speeds to save energy.

R3. If a permanent fault is detected on HP or LP, the system cancels the execution of main schedule on the remaining operational core, and immediately starts executing the incomplete tasks in the contingency schedule at the maximum speed. The system operates on a single core executing the contingency schedule until the faulty core is replaced or repaired.

The pseudocode in Algorithm 2 describes the operation of the system at runtime in normal mode, whereas Algorithm 3 describes the operations in contingency mode. It should be noted that even though the task start times are computed according to the *as late as*

Algorithm 2 Runtime-Events (Normal mode)

Event: New frame (period) starts
  All tasks are released and marked as *incomplete*
  // Let τ_H and τ_L be the first *incomplete* tasks in HP and LP core, respectively
  if τ_H is *ready* **then**
    Cancel contingency copy ρ_H
    Dispatch τ_H
  end if
  if τ_L is *ready* **then**
    Cancel contingency copy ρ_L
    Dispatch τ_L
  end if

Event: A task (τ_i) completes
  Run *acceptance test* for detection of transient fault in τ_i
  if no error is detected **then**
    Cancel contingency copy ρ'_i and mark τ_i as *complete*
    // Let τ_H and τ_L be the next incomplete tasks in HP and LP core, respectively
    if τ_H is *ready* **then**
      Cancel contingency copy ρ_H
      Dispatch τ_H
    end if
    if τ_L is *ready* **then**
      Cancel contingency copy ρ_L
      Dispatch τ_L
    end if
  else
    Switch to contingency mode
  end if

Event: A permanent fault is detected on a core
  Switch to contingency mode immediately

possible principle in the contingency schedule, when the system transitions to the recovery mode upon the detection of a fault, the required contingency tasks are dispatched *immediately* in the specified order, without waiting until the latest possible start times indicated in the contingency schedule. We conclude this section by the following remarks that justify fault tolerance capability of the proposed framework:

FT1. As long as both cores are functional, the system is able to recover from transient faults affecting any number of tasks, even when some of these faults may occur in the recovery mode, affecting a single copy of each pair of the contingency tasks ρ_i and ρ'_i.

Algorithm 3 Runtime-Events (Contingency mode)

Event: New frame (period) starts
  if Both HP and LP core are functional **then**
    Switch to Normal mode with Event "frame begins"
    return
  else
    All tasks are released and marked as *incomplete*
    Go to event: "Start Contingency mode"
  end if

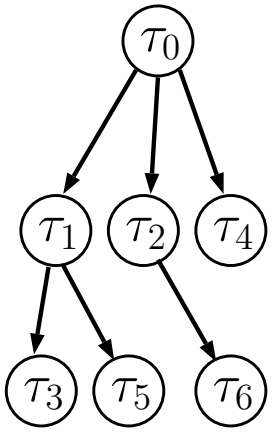
Event: Start Contingency mode
  // Let τ_X be the first *incomplete* tasks in the contingency schedule
  if ρ_X is not cancelled **then**
    Dispatch ρ_X on its respective core
  end if
  if ρ'_X is not cancelled **then**
    Dispatch ρ'_X on its respective core
  end if

Event: A contingency task (ρ_i) completes
  Run *acceptance test* for detection of transient fault in ρ_i
  if no error is detected **then**
    Cancel the other contingency copy (ρ'_i) if it is executing, and mark τ_i as *complete*
    // Let τ_X be the next incomplete task in the contingency schedule
    if Both HP and LP cores are functional **then**
      Dispatch ρ_X and ρ'_X on their respective cores
    else
      Dispatch ρ_X on the remaining operational core
    end if
  end if

FT2. The system can tolerate one permanent fault of any of the cores, even when the fault may occur during the execution of the contingency schedule thanks to the paired arrangement of the contingency tasks.

5.1.2 Task partitioning and ordering

Now we turn our attention to the problem of allocating the primary tasks on two cores and ordering them to satisfy the precedence constraints. In general, partitioning a set of real-time tasks on a multiprocessor system is a well-known NP-Hard problem. For this reason, our framework generates the task partitions decisions offline, based on the *list scheduling*

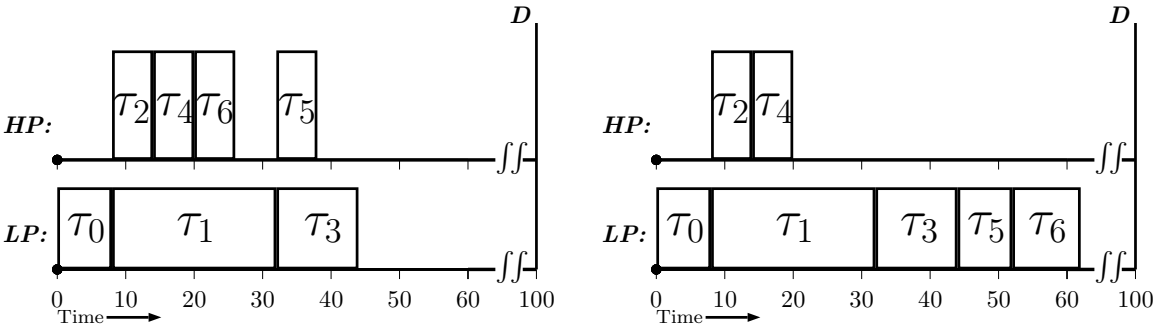


(a) Task graph

	W_i^{HP}	W_i^{LP}	a_i^{LP}	α_i^{LP}
τ_0	3.0	7.5	0.28	0.028
τ_1	10.0	23.1	0.45	0.045
τ_2	6.0	13.5	0.30	0.03
τ_3	6.5	14.6	0.40	0.04
τ_4	5.0	9.4	0.25	0.025
τ_5	4.5	7.9	0.20	0.02
τ_6	4.0	7.0	0.20	0.02

(b) Task parameters

Figure 5.3: Task Set for the Running Example

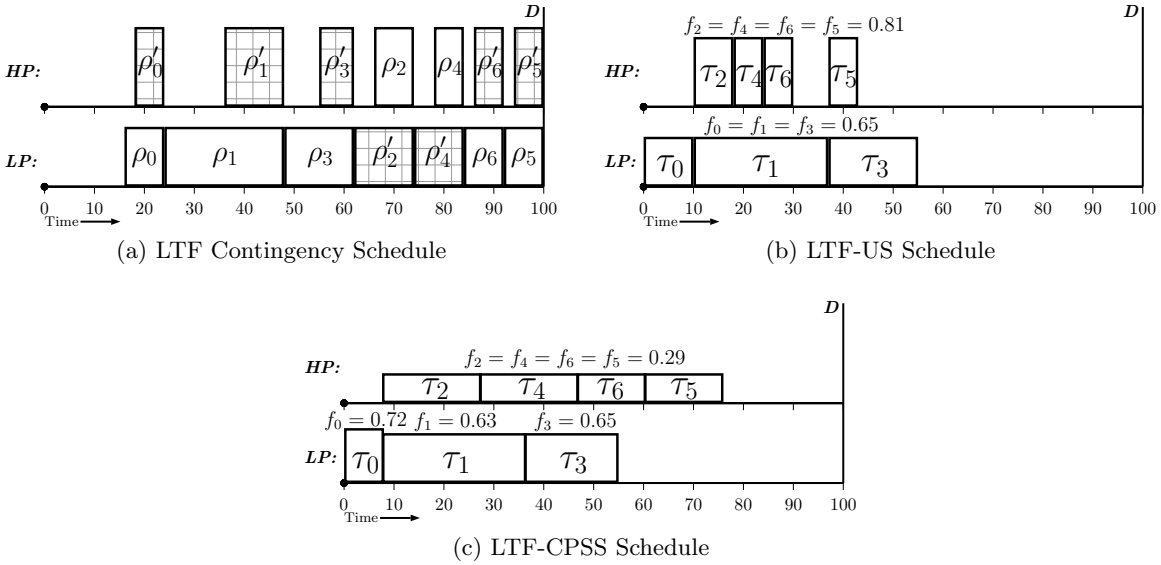


(a) LTF Partition

(b) TBLS Partition

Figure 5.4: Task partitioning algorithms

approach which is widely used to schedule tasks with precedence constraints [218,219]. In list scheduling, tasks are allocated to the available cores one at a time, starting with tasks with no predecessors (root tasks). A task whose all predecessors have been already allocated becomes also eligible for allocation. The algorithm keeps track of the tasks allocated to individual cores, as well as the current length of the schedule (*makespan*) on every core. If multiple tasks are eligible for allocation, ties may be broken using various parameters, such as execution time or power consumption of the tasks. Below we describe two heuristics



(a) LTF Contingency Schedule

(b) LTF-US Schedule

(c) LTF-CPSS Schedule

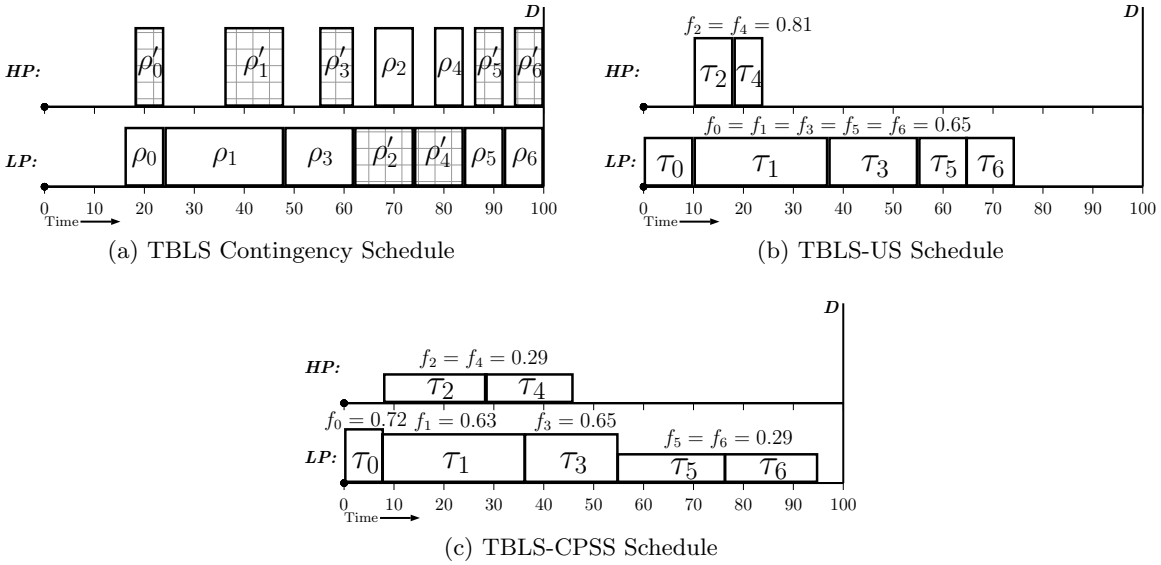
Figure 5.5: Contingency Schedules and Speed Assignments under LTF scheme

based on this list scheduling technique. It should be noted that once the task partitioning is determined through our heuristics, it is not changed at run-time; i.e., migration of the tasks is not allowed.

We use a running example to demonstrate the operation of our heuristics. As shown in Fig. 5.3, we have a task set with 7 tasks along with their dependencies indicated by the task graph. For each task, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. The other parameters are shown in Table 5.2b, where W_i values are computed assuming maximum speed on the respective core. For the HP and LP cores, we assume $f_{max}^{HP} = 1.0$, $f_{max}^{LP} = 0.8$, $P_{idle}^{HP} = 0.05$, and, $P_{idle}^{LP} = 0.02$.

Largest Task First (LTF). In this variant of list scheduling, tasks again are allocated one by one, and in each iteration, the next highest-priority eligible task is allocated to the earliest available processor. The priority of a task is determined by its size (the C_i^{HP} value). In case both of LP and HP cores are available at a given iteration, then we choose the LP core (assuming the deadline is still met), to save energy.

This method produces a good mix of tasks on the HP and LP core and generally produces



(a) TBLS Contingency Schedule

(b) TBLS-US Schedule

(c) TBLS-CPSS Schedule

Figure 5.6: Contingency Schedules and Speed Assignments under TBLS scheme

a schedule with a short makespan. The overall complexity of the algorithm is $O(n^2 + nE)$, where n is the number of tasks and E is the number of edges (or, dependencies) in the task graph. The operation of LTF for our running example task is shown in Fig. 5.4a. All tasks can finish execution by $t = 45$ (when executed at the maximum speed of their respective cores.) The corresponding contingency schedule for this task-allocation is shown in Fig. 5.5a.

Threshold-based List Scheduling (TBLS). This method attempts to exploit the low-power feature of the LP core. Specifically, we define a *threshold* value for the utilization on the LP core. If the total task utilization (on the LP core) is less than this threshold value, then all tasks are placed on LP. Otherwise, the algorithm allocates some tasks on the HP core in order to keep the LP core's total utilization under the predefined *threshold* value.

In this method, we use the list scheduling technique to allocate eligible tasks (by considering the precedence constraints) to the LP core as long as its utilization does not exceed *threshold*. If the total utilization of the task set exceeds *threshold*, then we choose some

tasks to be allocated to the HP core, such that LP core's utilization is kept under *threshold*. When there are multiple eligible tasks, the task with the highest C_i^{HP} value is chosen. This method prefers LP core to allocate most of the tasks, but can take advantage of the HP core when the task set's utilization is high. The runtime complexity of the algorithms is the same as LTF: $O(n^2 + nE)$. The operation of this algorithm on our example task set with a *threshold* value of 65% is shown in Fig. 5.4b. The corresponding contingency schedule is shown in Fig. 5.6a.

5.1.3 Speed assignment

After an allocation of tasks and their execution order is obtained on each core, we need to determine the execution speed (frequency) of the tasks, such that the frame deadline can be met by also considering the slots reserved for the contingency schedule. Assuming maximum frequency on each core, we first compute the activation and completion time of each task with worst case number of cycles. We call this the *canonical schedule*. When assigning speeds to tasks, we make sure that each primary task can complete its execution before the activation time of its contingency copy on the alternate core. By doing so, in the fault-free case, contingency copies are never activated, since we cancel the reservation in the contingency schedule as soon as the primary copy completes, thereby saving energy. For each primary task, its contingency copy activation time can be seen as its *pseudo-deadline* which is earlier than the frame deadline D. Using this method, we developed the following two techniques.

Uniform Scaling (US). In this method, we start with the canonical schedule and slow down all the tasks by a uniform scaling factor $S \leq 1.0$, ensuring that all tasks complete by their contingency activation times. The scaling factor, multiplied by the maximum speed of the respective core, gives the execution frequency (speed) of that core. This method determines the task with the *tightest* timing constraint (the task with the minimum [*contingency activation time - primary completion time*] difference), and scales the frequencies accordingly. The algorithm (with $O(n)$ complexity) ensures that no tasks would execute

beyond its contingency activation time. Therefore, scaling all tasks by the same (minimum) amount cannot possibly result in a deadline miss.

The schedules for our running example under LTF and TBLS and Uniform Scaling, named as LTF-US and TBLS-US, are shown in Fig. 5.5b and 5.6b, respectively. The uniform scaling is determined by the pseudo-deadline of τ_1 (which is the tightest) and gives a speed of 0.81 on HP and 0.65 on LP, for both LTF and TBLS schemes. The energy consumption of LTF-US and TBLS-US schemes are 28.49 mJ and 23.34 mJ, respectively. We can see that by limiting the use of HP core, TBLS consumes 18% less energy than LTF under US.

Critical Path based Static Speed (CPSS). This method is developed as an extension to the critical-path based DVFS algorithm originally proposed in [220]. Similar to Uniform Scaling, this method also assigns a pseudo-deadline for each task, which becomes the deadline for all paths ending at that task node. As opposed to imposing the same scaling factor to all the tasks, this method computes different scaling factors on the basis of each execution path, starting from the most critical one in terms of timing constraints. However, an assumption of the technique is that all processors are homogeneous and hence have the same power dissipation characteristics.

The algorithm in [220] starts by augmenting the original task graph based on the allocation and order of the tasks obtained from the task-partitioning phase, and then it identifies the source and sink nodes. In our context, each node is a sink node because it has a pseudo-deadline equal to its contingency copy activation time. Next, for each pair of source and sink nodes, the original method computes all the possible paths based on the augmented task graph. Each node can lie on more than one paths, and each path is associated with a (pseudo-)deadline. The authors identified the most critical path for each given node and each reachable sink, and claimed that this set of paths capture all the dependency and deadline constraints in the system. Once a path is scaled, its tasks are marked *assigned*, and the algorithm determines the next tightest path. It then scales only the tasks which are still *unassigned* in this path. It proceeds in this way until all tasks have been assigned

an execution speed.

In our adaptation of the algorithm from [220], we differ in the following way: i) we consider all possible paths from any source to any sink to be in our set of critical paths, and ii) when scaling a path, we use the system-level DVFS algorithm ENERGY-LU[83] to exploit the heterogeneity of the cores to minimize energy consumption, as opposed to using a common scaling factor for all tasks on a given path as done in [220]. In order to determine which path should be processed with ENERGY-LU first, we define *criticality* of a path by the ratio of 'time taken by all tasks with their assigned speeds' to the 'total time available until deadline'. A criticality value of 1.0 with a set of assigned speeds implies that if all tasks are executed at their respective speeds, then the total execution time will be equal to the deadline. In order for each path to meet the deadline constraints, its criticality value with assigned speeds should never exceed 1.0. Furthermore, the criticality value also indicates which path is the *tightest* among all paths in the set. In our method, we identify the *most critical path* in each iteration, and use the ENERGY-LU technique to determine a final frequency for each task on the path. Our algorithm terminates when all the tasks have an assigned frequency.

To identify the *most critical path*, first, we invoke ENERGY-LU on all the paths in order to get a tentative speed assignment for each task. For each task, its tentative speed is the maximum of all the speeds suggested for it by ENERGY-LU on all the paths it lies on. With tentative speed assignment for all tasks, we determine the path with the highest criticality value. Then we invoke ENERGY-LU again on that path one last time for a final frequency value for each of the tasks on the path and mark them as *assigned*. Paths in which all tasks have an assigned frequency are removed from the set of all paths. We proceed by processing the next *most critical path* and we consider only the unassigned tasks in that path. The time available for the unassigned tasks is computed by subtracting the total time taken by the assigned tasks with their respective assigned speeds, from the time available until deadline. We iterate through this loop until all tasks have an assigned frequency.

The schedules produced by CPSS are shown in Fig. 5.5c and 5.6c, for LTF and TBLS

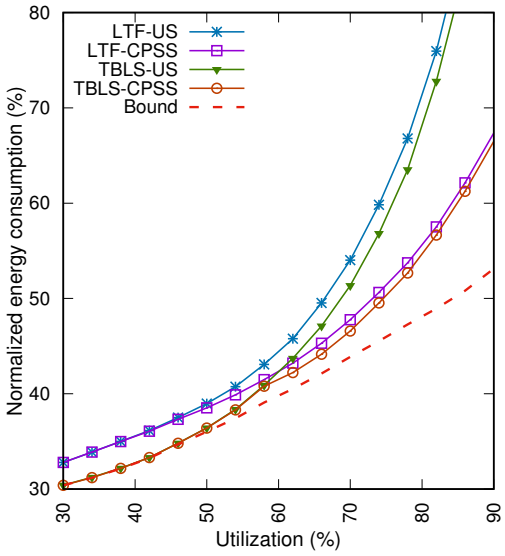
partitioning and our running example, respectively. They show that CPSS assigns relatively low execution-speed for tasks on the HP core, and also for τ_5 and τ_6 on the LP core. The energy consumption for LTF-CPSS is 19.30 mJ, which is 32% less than the uniform scaling in LTF-US scheme. By limiting the use of HP core, TBLS-CPSS consumes even less energy, 17.33 mJ, which is 25% and 10% better than TBLS-US and LTF-CPSS, respectively.

The time complexity of this algorithm depends on the number of all distinct paths from all source to all sink nodes– we denote it by k. The system-level DVFS technique takes $O(n^2 \log n)$ time for a path with n tasks. In each iteration of our algorithm, finding the *most critical path* would take $O(kn^2 \log n)$ time, and then applying ENERGY-LU for a final set of frequencies would take another $O(n^2 \log n)$. The iterations would run for at most n times, therefore, the algorithm's overall complexity is $O(kn^3 \log n)$.

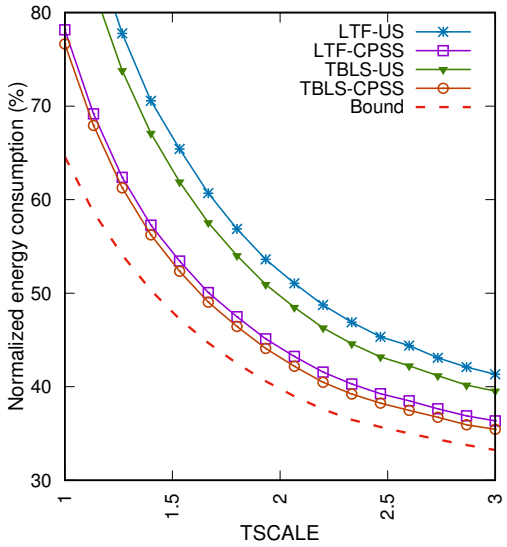
5.1.4 Dynamic Reclamation

Real-time systems must be designed to deal with the worst-case workload scenarios. However, real-time tasks often finish earlier than their worst-case estimates. Thus, to exploit the early completions and save more energy at run-time by dynamic slow-down, we developed a dynamic slack reclamation algorithm. In this algorithm first we compute offline speeds for each task based on any of our speed assignment algorithms. Then, using these speeds and the worst case number of cycles for each task, we compute a *reference activation time*, which corresponds to the time point when a task should start its execution when all tasks run at their assigned speeds and present their worst-case workload.

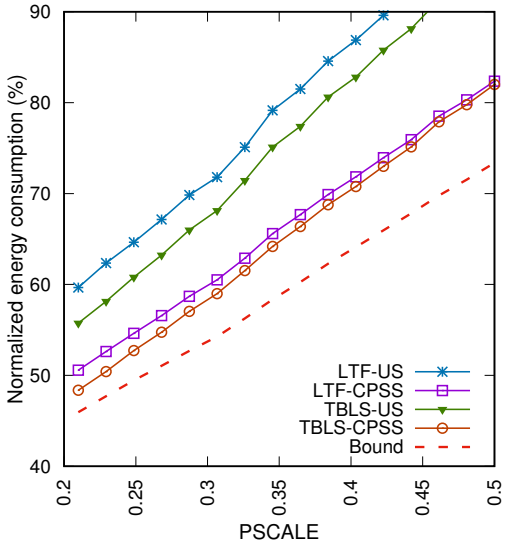
At runtime, when a task is about to be dispatched, we check the difference between its reference activation time and the current time. This difference is denoted as its *slack*. We recompute the assigned speed of the ready task by giving all the slack time to it, i.e., slow it down further such that it completes at its original (offline) reference completion time. Let f_i be the offline assigned speed for τ_i with C_i worst-case number of cycles, and s_i be the dynamically generated slack available at its dispatch time. Then, its dynamically adjusted speed, f_i^* is computed as $f_i^* = \frac{C_i f_i}{C_i + s_i f_i}$. The algorithm is invoked at dispatch time for every



(a) Impact of Utilization



(b) Impact of tscale at load = 75%



(c) Impact of pscale at load = 75%

Figure 5.7: Impacts of Utilization, tscale, and pscale.

task and it has constant time complexity ($O(1)$).

5.2 Experimental Evaluation

We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. In our simulator, we implemented the task partitioning schemes LTF and TBLS, as well as the speed assignment schemes US and CPSS, giving four combinations named as LTF-US, LTF-CPSS, TBLS-US, and TBLS-CPSS.

We also implemented a scheme named *Bound*: This scheme is based on brute-force search for all possible task partitioning and choosing the one with lowest energy consumption. *Bound* does not implement any fault tolerance, and removes all contingency tasks, setting the deadline of all tasks to D. The tasks are allocated on two cores using the topological order and the CPSS technique is used for speed assignment. We use this scheme as a lower bound for energy consumption and compare our proposed schemes that offer fault tolerance. The obtained energy consumption numbers are normalized with respect to the maximum energy consumption (observed in the considered parameter spectrum) of LTF-US scheme.

For each experiment, the simulator generates a task set containing n tasks, and a given total utilization, U. The utilization is calculated with respect to the LP core (which is more constrained in terms of performance) and normalized considering its maximum speed. Hence, $U = (\sum \frac{C_i^{LP}}{D})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [214] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U. We set the frame deadline $D = 100ms$. In order to experiment with arbitrary task-graphs, we use TGFF tool [221] to randomly generate a DAG with n nodes.

It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [56]. Therefore, as in [6], we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which models how execution time changes on the LP core for a given task, τ_i. Moreover, following [6], we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as

$\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within ranges suggested in [56]. Specifically, $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ hold. We assume for all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. In addition, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$ for all experiments.

We use task sets with $n = 10$ tasks, $f_{max}^{LP} = 0.8$ and $f_{max}^{HP} = 1.0$, unless otherwise stated. The value of *threshold* is set to 0.6 for TBLS. Every reported data point is the average of 1000 runs. We report the average energy consumption in fault-free executions, since faults are very rare events.

Impact of Utilization. Figure 5.7a shows the impact of utilization on normalized energy consumption. When the utilization is low, the energy consumption is largely dependent on the partitioning method, and not very much on the speed assignment schemes. This is because at low load, most tasks would typically be able to run at their energy-efficient frequency f_{ee} in both speed assignment techniques. TBLS puts all the tasks in to the LP core resulting in a better performance, whereas LTF utilizes the HP core for some tasks and spends somewhat more energy. As the load increases, all the schemes show an increase in the energy consumption, however, the CPSS schemes can keep the energy consumption low compared to the US schemes. This is because CPSS can set a suitable speed to each of the task, whereas US has to commit to a common speed for all tasks. With increase in the load, the advantage of CPSS becomes very significant (up to 35%). For a given speed assignment technique, TBLS partitioning performs slightly better than the LTF technique. Among all schemes, TBLS-CPSS performs the best and stays within 10% of Bound for up to 75% of system load.

Impact of tscale. Figure 5.7b shows the impact of varying *tscale* value for all tasks when the system load is fixed at 75%. As *tscale* increases, the normalized energy consumption of the system decreases. This is because a low value for *tscale* indicates a very efficient LP core. The plot shows that for the entire range of *tscale* values, TBLS-CPSS is performing the best, very closely followed by the LTF-CPSS scheme. Compared to Bound, both of

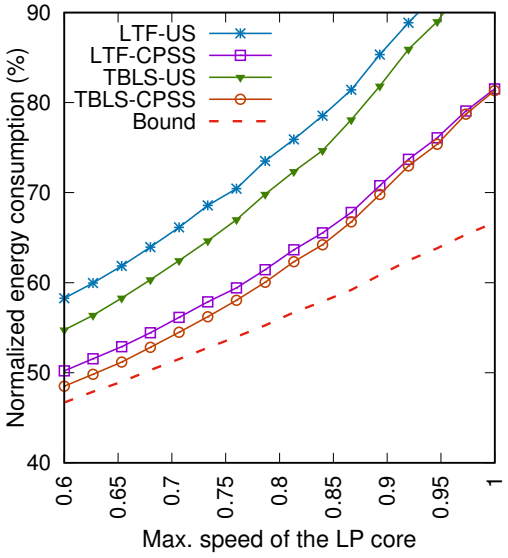
the CPSS schemes perform very close (around 6-12%) for the entire spectrum.

Impact of pscale. Varying *pscale* also has similar effect as shown in Fig. 5.7c. As *pscale* increases, the overall energy consumption of the system also increases. Increasing the value of *pscale* implies making the LP core more power-hungry, resulting in higher overall energy consumption. We can see that TBLS-CPSS performs best throughout the entire *pscale* spectrum, closely followed by LTF-CPSS.

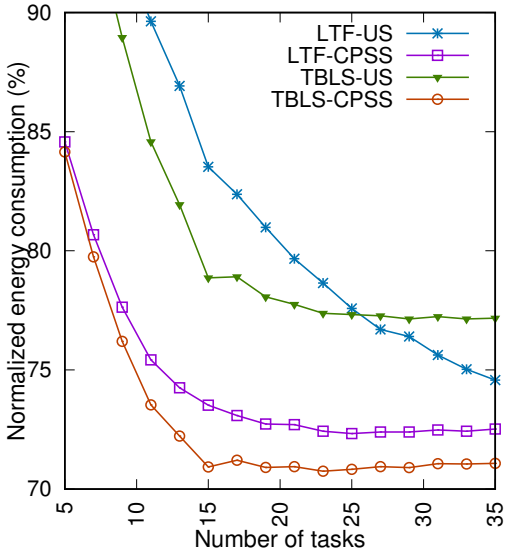
Impact of maximum speed of LP core. In this set of experiments, we varied the maximum speed of the LP core while fixing the load at 75%, as shown in Fig. 5.8a. We see that the energy consumption of all schemes increase with increasing f_{LP}^{max}. This is because, when the utilization is kept fixed at 75% (which is computed relative to f_{LP}^{max}) the effective amount of workload on the system increases with increasing LP core speed, which is reflected in the results. We observe again that TBLS-CPSS performs the best, closely followed by LTF-CPSS. In fact, Bound scheme performs only 5% better than TBLS-CPSS when LP core's maximum speed is low. However, their difference increases with the LP core's maximum speed.

Impact of number of tasks. Fig 5.8b shows the impact of number of tasks for a system with 75% load. We see that for small number of tasks, the performance of all the schemes is affected. As the number of tasks grows, the average task size decreases and the performances of various schemes stabilize. For TBLS partitioning, the advantage of using CPSS scheme over US can be up to 18% when number of tasks is low, but it stabilizes at 10% with the increase in number of tasks. The plot also shows that LTF-US starts to outperform TBLS-US as soon as number of tasks exceeds 25. This is because when the average task size decreases, this favors the LTF scheme greatly due to the reduced cost of using the HP core with US. *Bound* is not shown in these experiments due to its prohibitive running time with increased number of tasks.

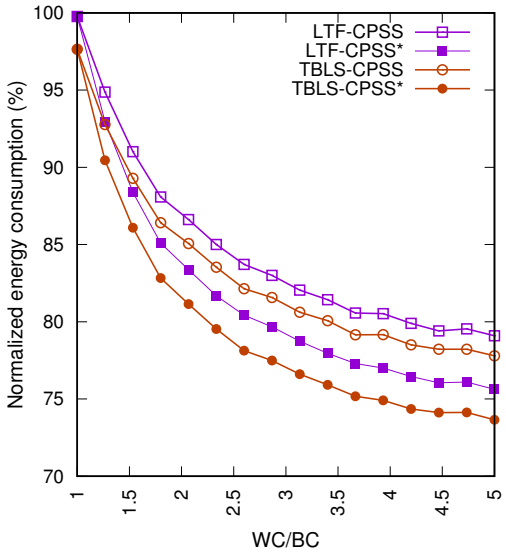
Impact of Workload Variability. To evaluate the gains due to our dynamic reclamation scheme in the presence of the workload variability, we first define the ratio WC/BC as the ratio of the worst-case execution time to the best-case execution time. During the



(a) Impact of maximum speed of LP core at load = 75%



(b) Impact of number of tasks at load = 75%



(c) Impact of WC/BC at load = 75%

Figure 5.8: Impact of the LP core's maximum speed, number of tasks, and workload variability

experiments, the actual execution time of every task is randomly generated between its worst-case and best-case execution time, using a uniform probability distribution. A higher

value of WC/BC indicates larger amount of runtime slack being generated, providing opportunities for further energy savings. In these experiments, we evaluated 1000 execution frames for each task set. Figure 5.8c shows that our techniques with dynamic reclamation enabled (indicated by '*') are able to save additional energy at runtime. The dynamic schemes are about 6-8% more efficient than their static counterparts.

5.3 Concluding Remarks

In this chapter, we proposed a fault-tolerant framework for dependent real time tasks executing on a heterogeneous dual core system. We presented task partitioning heuristics for allocation and ordering of tasks to the available processing cores, and developed speed assignment techniques which can ensure low-energy consumption while providing reliability against transient and permanent faults. Simulation results demonstrate that our proposed techniques are capable of energy efficient operation and perform close to a theoretical lower bound.

Chapter 6: Energy-aware Fault-Tolerant Real-Time Scheduling of General Periodic Tasks

In this chapter, we investigate energy-efficient and fault-tolerant implementation of periodic real-time systems upon heterogeneous dual core systems. In previous chapters, we focused on frame-based systems where all tasks share a common period. However, the actual implementation of a number of real-time systems are based on general periodic tasks which are invoked at different rates [5]; so we believe this study fills an important gap.

In our framework each periodic real-time task (called the *primary*) is assigned to one of the cores: all the instances of that periodic task ("jobs") are released and executed at periodic intervals on that core. In addition, for every such primary task, a *backup* periodic task is assigned to the alternate core. Since every real-time job has a backup allocated to the other core, transient faults in all primary jobs can be tolerated. Similarly since there are two copies of each periodic task assigned to alternate cores, the system can tolerate one permanent fault of any core by switching to the functional core if necessary.

To manage the energy consumption, we use two mechanisms: i.) the primary tasks are executed at low voltage/frequency levels using Dynamic Voltage and Frequency Scaling, and, ii.) the backup copies are *delayed* to the extent it is possible to enable their cancellation in case the primary completes without a fault. In addition, we use the Mixed Primary/Backup Scheduling framework in which a given core may be assigned both primary and backup copies of (distinct) tasks [7]. This is in contrast to the so-called Standby-Sparing systems in which one core is exclusively dedicated to the primaries and the other one (spare) executes only the backups [53, 54, 99, 100, 181]. While the mixed primary-backup scheduling framework significantly improves the schedulability and energy saving potential, it also presents challenges in terms of task allocation and scheduling.

A significant challenge in these settings is to find an efficient way to delay the backup copies: because of periodic and preemptive execution settings, a backup copy can be preempted multiple times by other jobs; but it should still meet its deadline when needed. To tackle this, we leverage two mechanisms: one is *assigning proper priority levels* to periodic primary and backup tasks, and the other one is the actual delaying of the backups using the *dual-queue* mechanism [100, 222]. Specifically, the backup tasks are promoted and become eligible for execution after a pre-determined time interval after their release. These promotion times are shown to be safe, in that they are derived using the well-known critical instant analysis technique for fixed priority periodic real-time tasks [5]. In particular, we show that when combined with the dual queue based delaying mechanism, assigning tentatively high-priority levels to the backup tasks gives the maximum delaying and energy saving opportunities. To the best of our knowledge, this research effort is the first study that explores fault-tolerant and energy-aware mixed primary backup scheduling of periodic real-time tasks on heterogeneous dual core systems.

Similar to the previous chapters, we consider a heterogenous dual-core system with a high-performance (big) core and a low-power (little) core, denoted as HP and LP respectively. The details of the system model is given in Chapter 3.1.

Each (primary) task τ_i has an associated backup task B_i with exact same timing parameters. τ_i and B_i are allocated to different processing cores. Whenever a task instance is released, its backup copy is also released and is allocated to the alternate core. In order to maintain energy efficiency, the backup copy instances are delayed as much as possible while respecting their deadlines. When a primary copy (τ_i) completes, the *acceptance* (or, *sanity*) tests [1] are performed to check the existence of errors induced by *transient* faults. If a fault is not detected, then B_i (or, its remaining part) on the other core may be cancelled to save energy, as depicted in Figure 6.1, which shows a single periodic task with period 20 ms and execution time 7.5 ms. If a fault is detected, the backup copy runs to completion. If a permanent fault occurs on any of the cores, the other core can still execute one copy of each task's instances.

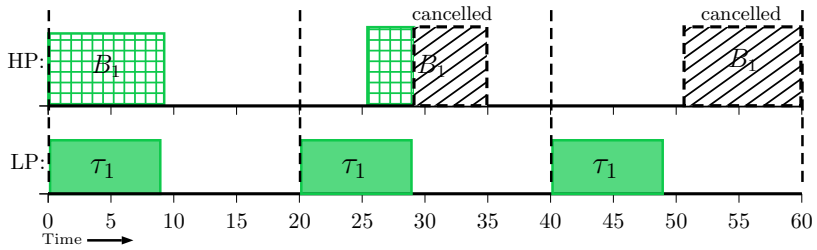


Figure 6.1: Primary/Backup Overlap

It should be noted that when a backup copy executes in the fault-free case, it is essentially a redundant execution which increases the energy consumption of the system significantly. Ideally, we would like to minimize redundant execution in order to conserve energy. As shown in Figure 6.1, the first instance of τ_1 and its backup B_1 execute in parallel, wasting a lot of energy. In the second instance of τ_1, we delayed B_1 to some extent and were able to cancel some parts of it. In the third instance, B_1 was delayed enough so that its execution could be entirely omitted.

Problem Statement. Given a set of real-time independent periodic tasks and a heterogeneous dual-core system, minimize the energy consumption by determining

1. The allocation of tasks to cores such that the primary and backup copy of each task are assigned to different cores, and,

2. The priority assignment, scheduling, and processing frequency assignment decisions for individual periodic task instances.

In the following sections, we first present preliminary (background) material and then we develop multiple components of our proposed framework.

6.1 Preliminaries

6.1.1 Work-Conserving Fixed-Priority Periodic Scheduling

Most of the traditional hard real-time scheduling theory is based on the **work-conserving** approach, in which the processor never idles as long as there are ready jobs to execute [5]. A well-known framework is **fixed-priority scheduling (FPS)** in which all the jobs generated by a given periodic task are assigned the same priority level during execution.

The real-time *feasibility analysis* is concerned with assessing if all the real-time jobs will meet their deadlines given the characteristics of the workload at hand [5]. In FPS, it is known that the worst-case response time of a periodic task occurs when it is released at the same time as all high-priority tasks. Specifically, the worst-case response time S_i of a task τ_i can be computed using the following iterative formula [223]:

$$S_i^{(k+1)} \;=\; c_i \;+\; \Sigma_{\tau_j \in\, hp(\tau_i)} \lceil (S_i^{(k)}/T_j) \rceil \times c_j \tag{6.1}$$

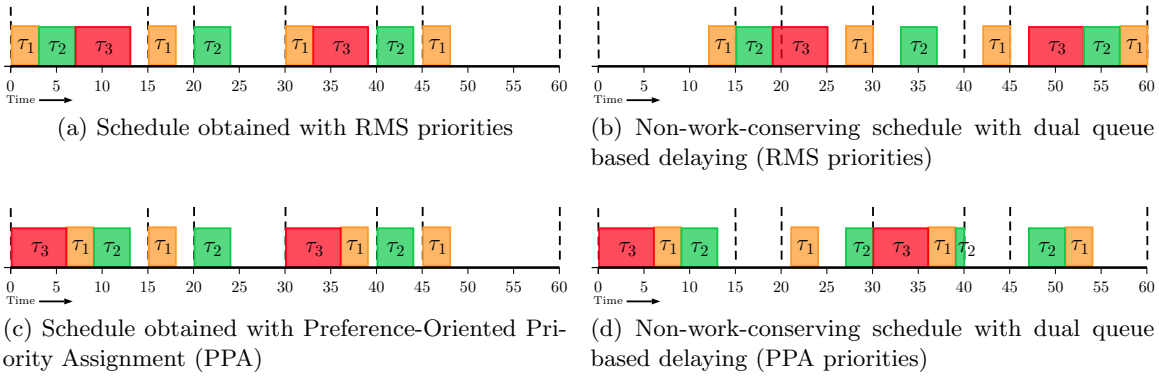
Above c_i is the worst-case execution time of τ_i and $hp(\tau_i)$ denotes the set of tasks which are assigned a priority level higher than that of τ_i. In this iterative approach, initially $S_i^0 \;=\; c_i$ and the iterations continue until $S_i^{(k+1)} \;=\; S_i^{(k)}$. If at any point $S_i^{(k)}$ exceeds the period (relative deadline) T_i, then the task will not meet its deadline. Otherwise, the task will meet its deadline and the worst-case response time S_i is found as the last value obtained for $S_i^{(k)}$.

An important FPS policy is *Rate Monotonic Scheduling (RMS)* in which the priorities are inversely proportional to the periods. RMS is known to be optimal among all periodic fixed-priority assignments, in the sense that all task sets that can meet their deadline with any fixed-priority assignment can also do so using RMS [224]. This optimality makes RMS the most widely known and adopted fixed priority assignment policy for periodic real-time tasks [224].

As an example consider the task set given in Table 6.1 with three periodic tasks, τ_1, τ_2

Table 6.1: Example Task Set 1

	Period	Execution Time
τ_1	15 ms	3 ms
τ_2	20 ms	4 ms
τ_3	30 ms	6 ms



(a) Schedule obtained with RMS priorities

(b) Non-work-conserving schedule with dual queue based delaying (RMS priorities)

(c) Schedule obtained with Preference-Oriented Priority Assignment (PPA)

(d) Non-work-conserving schedule with dual queue based delaying (PPA priorities)

Figure 6.2: Work-conserving and non-work-conserving fixed-priority schedules

and τ_3. For illustration purposes, we assume all tasks execute at maximum frequency. The corresponding schedule obtained using RMS is shown in Figure 6.2a. The period boundaries are denoted by vertical dashed lines in the figure. As it can be observed, all periodic task instances meet their deadlines.

6.1.2 Non-Work-Conserving Fixed-Priority Periodic Scheduling

There are a number of scenarios where it is desirable to *delay* periodic tasks as long as they can still complete before their respective deadlines. For instance, when the workload includes non-real-time aperiodic jobs that arrive at unpredictable times, a common objective is to execute them as soon as possible to minimize their response time. In this case, periodic hard real-time tasks may be delayed maximally to enable early execution of the aperiodic jobs [5].

In these cases, work-conserving policies such as conventional RMS are no longer appropriate; instead non-work-conserving approaches are considered. For example the *dual-queue* based approach [100, 222] works as follows: The system is equipped with two (dual) queues, named *upper queue* and *lower queue*, respectively. Upon arrival, each periodic real-time job is first put to the lower queue, and remains there until a certain *promotion time* at which it is moved to the *upper queue*. Only jobs in the upper queue are eligible for execution; and they are dispatched according to the underlying fixed (e.g., RMS) priorities.

The crux of the scheme is to choose the promotion time safely and maximally, in order to delay the execution of the periodic jobs as much as possible. Specifically, the promotion time of a job of τ_i after its release time is computed as:

$$Y_i = T_i - S_i$$

where T_i is its period and relative deadline, and S_i is its worst-case response time computed through the iterative formula (6.1) based on the critical instant analysis. It is based on the observation that the job would still meet its deadline after being moved to the upper queue Y_i time units after its release time, even if it is subject to the maximum possible interference by higher-priority jobs [222]. The task promotion times (Y_i values) may be computed offline (before execution) for all periodic tasks.

Returning to our example task set, we can compute the task promotion times as: $Y_i = T_i - S_i$. The S_i values are obtained by applying the formula (6.1) iteratively, and the promotion times are found as $Y_1 = 12, Y_2 = 13$ and $Y_3 = 17$, for τ_1, τ_2, and τ_3. The resulting non-work-conserving fixed-priority schedule (where tasks are first delayed in the lower queue and then eventually dispatched from the upper queue with RMS priorities) is shown in Figure 6.2b. It should be noted that many real-time jobs are significantly delayed, but they still meet their deadlines. This dual queue based approach will be instrumental in our mixed primary/backup energy-aware scheduling approach, as we elaborate in Section 6.2.

6.1.3 Preference-Oriented Priority Assignment (PPA)

A more recent study considers the *execution preferences* of real-time tasks explicitly in the scheduling phase [192]. Specifically, periodic real-time tasks are classified as ASAP or ALAP, depending on whether there is a preference to execute them *as soon as possible* or *as late as possible*, respectively, but still before all the hard deadlines.

In [192], the problem of finding a fixed priority assignment to satisfy the periodic real-time tasks' execution preferences while meeting the deadlines is considered. The solution is obtained through the Audsley's Optimal Priority Assignment Algorithm (AOPA) [225], which runs in time $O(n^2)$ for n periodic tasks. AOPA, which was originally proposed for tasks with potentially different release times [226], proceeds by first assigning a task to the lowest priority level, by making sure that that task would meet its deadline even in the worst-case activation pattern. Then it proceeds in iterative manner for priority levels $n-1$, ..., 1. The preference-oriented priority assignment (PPA) scheme, proposed in [192], proceeds in the same way, but it assigns low priority levels to the ALAP tasks and high priority levels to the ASAP tasks as much as possible, while still preserving the timing constraints.

For our example task set, now assume that τ_1 and τ_2 are ALAP tasks, while τ_3 is an ASAP task. PPA assigns the lowest priority to τ_2, medium priority to τ_1, and highest priority to τ_3. The resulting fixed-priority schedule where all the deadlines are met is presented in Figure 6.2c. It should be noted PPA is, just like RMS, an optimal fixed-priority assignment; but it incorporates the task execution preferences whenever possible in the priority assignment phase.

While PPA takes into account task's execution preferences, it is still by default a work-conserving approach. It is possible to combine PPA with the dual queue mechanism to further delay the ALAP tasks (thereby creating a non-work-conserving schedule). When applied to the schedule in Figure 6.2c, we obtain the solution in Figure 6.2d, where the ALAP tasks are delayed until their promotion times. This time promotion times for ALAP tasks τ_1 and τ_2 are computed as $Y_1 = 6$ and $Y_2 = 7$. We observe that using the dual queue

mechanism helps to increase the delay in the execution of the ALAP tasks, and all the deadlines are still met.

6.2 Mixed Primary/Backup Scheduling of Periodic Tasks

Our dual objective in fault tolerance (Section 3.1.3), in terms of tolerating transient faults can be achieved by scheduling a separate backup copy of each periodic task instance. Moreover we require that the primary and backup copies of a given task are scheduled on different cores to provision for the permanent fault of any single core. Note that by scheduling a separate backup copy which is, if needed, executed at the maximum core speed, we also guarantee to fully mitigate the task-level reliability loss (with respect to the transient faults) induced by the application of DVFS [174].

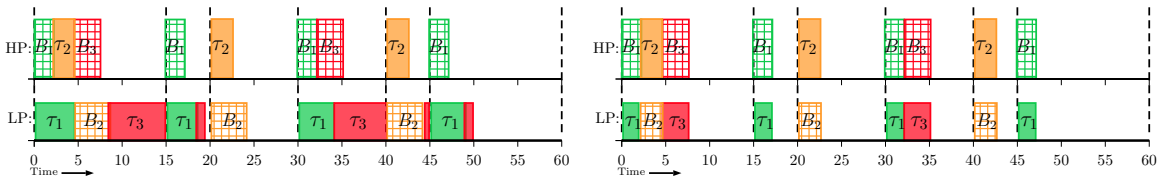
Unlike the standby-sparing systems [6, 53, 100, 181] where one core is allocated only the primary copies of tasks and another one solely to the backups, in our work we adopt the *mixed primary/backup scheduling* approach: a given core can execute primary and backup copies of various tasks for maximum flexibility with respect to energy awareness and schedulability.

Table 6.2: Example Task Set 2

	T_i	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}	a_i^{LP}	α_i^{LP}
τ_1	15	1.8	3.8	1.98	0.84	0.36	0.036
τ_2	20	2.0	4.0	2.20	0.64	0.26	0.026
τ_3	30	3.5	7.9	3.85	1.84	0.38	0.038

For illustration purposes, we consider a running example throughout this section. Consider the task set with parameters given in Table 6.2. It has 3 tasks τ_1, τ_2 and τ_3, with respective backup copies B_1, B_2 and B_3 for fault tolerance, which are to be executed on the HP and LP cores. In the table, the period and execution times are expressed in milliseconds, while the energy values are expressed in millijoules. We chose $f_{max}^{HP} = 1.0$ and

$f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$, and for each task, $\alpha_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. We assume that the primary copy of τ_2 and the backup copies B_1 and B_3 are allocated to the HP core, while the primary copies τ_1 and τ_3, along with the backup copy B_2 are allocated to the LP core. This makes sure that the two copies of the same task are always on different processing cores.



(a) Without backup Cancellation (b) With backup Cancellation

Figure 6.3: Mixed Primary/Backup Scheduling with RMS

In Figure 6.3a we present the mixed primary/backup schedules that we obtain if we use RMS as the scheduling policy on each core where all tasks are executed at the maximum frequency of their respective cores. Observe that all primary and backup copies meet their deadlines, and the fault tolerance objectives are achieved.

The energy consumption on each core can be computed as the sum of core energy when executing tasks (which can be computed by the sum of E_i values of all task instances allocated to that core) and idle core energy (which can be computed by the product of idle processor time and idle processor power). The total energy consumption of both cores obtained in this way is 33.49 mJ. We observe that a significant portion of the total energy is due to the duplicate execution of the backup tasks.

As mentioned in Section 3.1.3, in case a fault is not detected at the end of execution of the primary, the remaining part of the corresponding backup copy can be cancelled. This gives a powerful mechanism to reduce the energy consumption. If we use this dynamic backup cancellation mechanism, we obtain the schedule in Figure 6.3b. It can be observed that some portions of the tasks (primary and backup) on the LP core are cancelled due to

the completion of the counterpart task, and we obtain a reduced energy consumption of 29.17 mJ.

While it is important to guarantee the timely completion of backup tasks in case faults are detected, since faults are rare events, the average energy consumption in all execution scenarios will be dominated by fault-free execution scenarios. Consequently, in all subsequent examples we show only fault-free executions and assume that all backups (primaries) are cancelled when the corresponding primary (backup) completes successfully on the other core. For clarity, we do not show the cancelled parts of the tasks in the schedules.

While incorporating this fundamental dynamic backup cancellation mechanism, our framework consists of multiple solution layers aimed at reducing the energy consumption dynamically. In particular, we use DVFS to lower the execution speed of the primaries, and use appropriate mechanisms to delay the execution of the backups to maximize the opportunities for their cancellation.

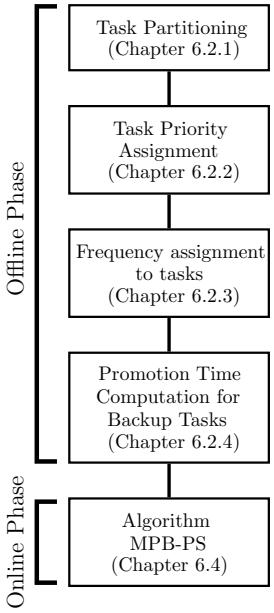


Figure 6.4: Mixed Primary/Backup Scheduling Components for Fixed-Priority Periodic Tasks

Specifically, our solution consists of *offline* and *online* phases, as shown in Figure 6.4. In the offline phase, we use *task partitioning*, *priority assignment*, *frequency assignment* and *backup promotion time computation* mechanisms. In the online phase, on each core the tasks (and backups, if needed) are executed at the pre-determined priority and frequency levels, and the backups are delayed until their pre-computed promotion times in order to enable their cancellations dynamically.

6.2.1 Task Partitioning

Our framework generates the task partitioning (allocation) decisions offline, based on the well-known *list scheduling* approach. Specifically we propose the following *List-Scheduling with Primary/Backup (LSPB)* variant. In this algorithm, we consider the primary copies of the tasks and employ list-scheduling algorithm to allocate them. First, the tasks are ordered according to their decreasing nominal utilizations. Then, each primary task is placed on a processing core on which it is *feasible* and which has the maximum *free capacity* after the placement. *Feasibility* is checked by using Time Demand Analysis and RMS priority assignment, which is known to be an optimal fixed priority assignment. *Free capacity* on the HP core, Ω^{HP}, is defined by $(1.0 - \sum_{\tau_i \epsilon \Gamma_p} \frac{C_i^{HP}}{T_i} / f_{max}^{HP})$, where Γ_p is the set of all primary tasks assigned to the HP core, augmented by the task under consideration. Similar equation is used for computing Ω^{LP}. After all the primary tasks are feasibly placed on the processing cores, their backup copies are allocated on the respective alternative core. Finally, feasibility is checked again taking the backup copies into account. The partitioning shown in Figures 6.3a and 6.3b were in fact obtained using the LSPB technique. It can be observed that it generates a relatively balanced workload distribution which opens up opportunities to reduce energy consumption.

6.2.2 Priority Assignment

After determining the task partitioning and obtaining a task-set for each core, we turn our attention to the priority assignment to tasks. In fixed-priority scheduling, the execution order of task instances depends directly on their priorities (Section 6.1.1). The RMS priority scheme, in which tasks with smaller periods receive higher priorities, is a natural option on each core. However, in addition to RMS, there have been other fixed priority assignment algorithms proposed in the literature including the *Preference-Oriented Priority Assignment (PPA)* policy (Section 6.1.3), which considers execution preferences of different tasks [192]. In our setting, we can invoke the PPA scheme to assign priorities after designating the primary tasks and backup tasks as "as soon as possible (ASAP)" tasks and "as late as possible (ALAP)"' tasks, respectively. We show the execution schedule for PPA in Figure 6.5, using our example task set. Although we assign low priorities to the backup tasks, the schedule shows that it is not very effective in cancelling the backup copies– it incurs high energy consumption figure of 29.3 mJ. We can attribute this to the fact that the presented solution still generates work-conserving schedules for backup tasks.

In Section 6.3, we will introduce another priority assignment policy, which, when coupled with other components, gives much more improved energy saving opportunities.

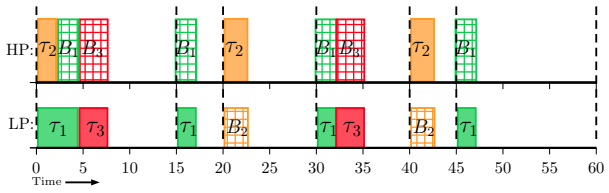


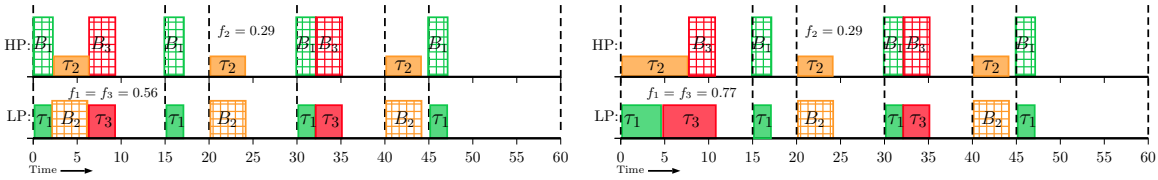
Figure 6.5: Schedule for PPA

6.2.3 Frequency Assignment

After an allocation of tasks and their priority assignment is obtained on each core, we can use DVFS to slow down the primary copies and reduce energy consumption. We apply

DVFS to primary copies only, and the backup copies are executed at the maximum speed. Not scaling the backups allows to delay their execution further, and also, it mitigates the task-level reliability-loss incurred due to DVFS [174]. For the primary tasks, we need to determine the speed (frequency) of the task-execution, such that the deadlines of all task instances (primary or backup) can be met.

We use a modified version of the well-known *Sys-Clock* algorithm proposed in [85], which assigns a common (minimum) execution-speed to all the tasks on a given core without violating any deadlines. *Sys-Clock* has very low computational overhead. In our modified version of *Sys-Clock*, only the primary tasks are scaled while the backup tasks are assigned the maximum speed of their respective cores. We modified the original algorithm by considering that backup tasks are always executed at maximum frequency levels, and only the primary tasks are scaled. For our example task set, Figures 6.6a and 6.6b show the execution schedules for RMS and PPA priorities, respectively when DVFS is applied. It shows that with DVFS, RMS consumes 22.86 mJ and PPA consumes 23.1 mJ, which is about 20% improvement in both cases compared to the cases without DVFS.



(a) Schedule for RMS with DVFS

(b) Schedule for PPA with DVFS

Figure 6.6: Schedules with DVFS

6.2.4 Promotion Time Computation for Backup Tasks

Once the task allocations, priorities, and frequencies are determined, we aim to delay the execution of the backup instances as much as possible, given that no instance should miss

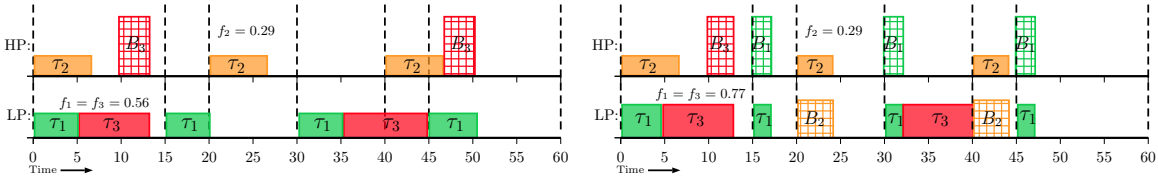
its deadline. A delayed backup copy has a greater chance of getting cancelled by its primary copy's completion, which helps to reduce their energy consumption in fault-free cases. In order to delay the backup copies, we used the dual queue based non-work-conserving scheduling algorithm discussed in Section 6.1.2, by adapting to heterogeneous processors and mixed primary/backup execution with DVFS.

In this delaying technique, when a backup task instance is released, it is first placed on a lower queue, and it can only be promoted to an upper queue at a precomputed promotion time. Promotion times are computed as the same way discussed in Section 6.1.2, by first computing the *worst-case response time*, using the iterative formula (6.1). Specifically, for a backup task B_i, its worst-case response time S_i can be computed as:

$$S_i^{(k+1)} \;=\; W_i \;+\; \Sigma_{\tau_j \in\; hp(B_i)} \lceil (S_i^{(k)}/T_j) \rceil \times (W_j(f_j)) \tag{6.2}$$

In this formula, W_i is the worst-case execution time of the backup task B_i on its assigned processing core under maximum speed. $hp(B_i)$ is the set of all higher priority (primary or backup) tasks on that specific core. For such a high priority primary task τ_j, we use its execution time, W_j, after scaling it with the *Sys-Clock* speed, f_j. If τ_j is a backup task, then we consider its execution time at the maximum speed of its processing core. Iterative computation continues until $S_i^{(k+1)} \;=\; S_i^{(k)}$, which gives the worst-case response time S_i. After that, the scheme computes the promotion time Y_i for each backup task by subtracting its worst case response time (S_i) from its relative deadline (period) T_i.

A backup task is dispatched on the processor only if the promotion time has elapsed and it is in the upper queue. Primary task instances, on the other hand, are always placed in the upper queue directly, and they are dispatched according to their assigned fixed priorities. Figure 6.7a and Figure 6.7b show the execution schedules with backup-delaying for RMS and PPA priorities respectively, when applied along with DVFS. It is evident that many of the backup task executions are cancelled in these schemes, which provide energy efficient performance (16.36 mJ for RMS, 19.33 mJ for PPA.) RMS with backup-delaying is almost 28% better than RMS without delaying (Figure 6.5).



(a) Schedule for RMS with DVFS and Dual-Queue Delaying)

(b) Schedule for PPA with DVFS and Dual-Queue Delaying)

Figure 6.7: Schedules with Backup Delaying

6.3 Reverse Preference-Oriented Priority Assignment (RPPA)

We observed that while PPA made an attempt to delay the backup copies by assigning lower priorities, in the motivational example, it did not translate directly to energy savings, even with dual queue based delaying, compared to RMS. In fact, our detailed experimental evaluation (Section 6.5) will confirm the generality of that observation.

We note that this is primarily due to the fact that during the computation of promotion times (delays) for low-priority backup tasks using Equation (6.2) the execution times of high-priority tasks act as a negative factors: the lower the scheduling priority of a backup, the higher will be the interference that need to be taken into account when computing the promotion times, and the smaller will be the promotion time (delay) we can afford for the backup.

This suggests an alternative but seemingly counter-intuitive solution: **assign high priorities to backup tasks before applying the dual queue based delaying technique**, relying on the fact that this will help to increase their promotion times, and the total amount we can delay in them in the lower queue. Even though they will indeed execute at high priority eventually, this will only happen when they are maximally delayed in the lower queue through the extended promotion times.

The proposed scheme, called the *Reverse Preference-Oriented Priority Assignment (RPPA)*, is very similar to PPA, but it assigns higher preference to backup tasks, and lower preference to the primary tasks to the extent it is possible. Like before, the backup tasks are delayed

using the dual queue based delaying technique.

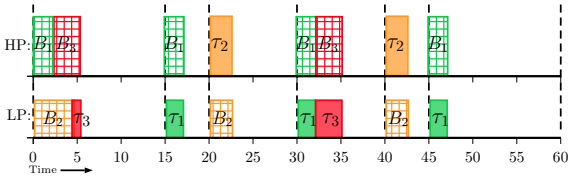
RPPA priority assignment is also optimal as PPA and RMS – it never results in a loss in schedulability as long as there exists a feasible solution. In the extreme case where a given task set cannot be scheduled by assigning low priority to (most of) backup tasks, it will generate another priority assignment which may resemble PPA or RMS, even though we observed that in practice it is able to generate a feasible priority assignment by assigning backup tasks high priorities in many cases.

For our example task set, the schedules with PPA and RPPA are shown in Figure 6.7b and 6.8c, respectively (with DVFS and backup-delaying enabled). It shows that for PPA, promotion times for B_1, B_2 and B_3 are 0.3, 0 and 10, respectively. This means B_2 gets promoted as soon as they arrive, and it could not be delayed at all. B_1 and B_3 are only marginally delayed. In contrast, for RPPA, promotion times for B_1, B_2 and B_3 are found as 13.2, 16 and 24.7, respectively. This big improvement in backup-delaying translates to more backup cancellation which reduces the overall energy consumption. With DVFS and backup-delaying enabled, RPPA consumes only 9.42 mJ of energy, which represents about 42% and 51% improvement compared to RMS and PPA, respectively.

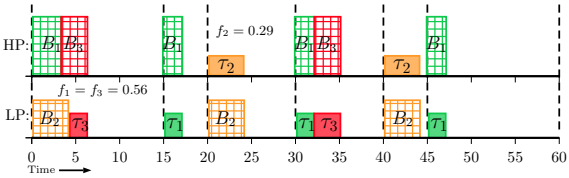
We also observe that reversing the priorities in RPPA by itself (without dual-queue based delaying) does not help in consuming less energy, as it can be seen in Figure 6.8a and 6.8b, which consume 26.52 mJ and 22.23 mJ energy, respectively. However, when RPPA is combined with the non work-conserving dual priority algorithm, then its big potential for energy consumption becomes clear (in this case, giving more than 55% improvement.)

6.4 Algorithm MPB-PS

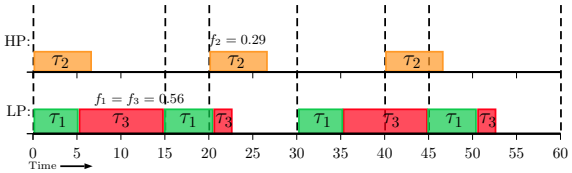
This section describes the algorithm executed in the *online* phase of our framework, called the *Mixed Primary/Backup Periodic Scheduling (MPB-PS)* algorithm. In this algorithm, the runtime events are processed on the HP and LP core separately. In the offline phase, the tasks are allocated to the HP and LP cores, priority assignment is made to tasks on each core, and the execution frequency and promotion times are computed.



(a) Schedule for RPPA (Without DVFS or Dual Queue)



(b) Schedule for RPPA (with DVFS but without Dual-Queue)



(c) Schedule for RPPA (with DVFS and Dual Queue)

Figure 6.8: Schedules for RPPA

At runtime, when a primary task is released, its backup copy is also released on the alternate core (with the same deadline). There are four important events that our runtime algorithm needs to consider: task release, completion, promotion and cancellation. The details of this algorithm are given in Algorithm 4.

On each processing core, we have two queues: the upper and the lower queue. Tasks are eligible for execution only if they are in the upper queue. Backup copies are initially put to the lower queue, and they get promoted to the upper queue at the precomputed promotion times. As shown in Algorithm 4, when a task is released, it is checked whether it is a primary or backup copy, and then it is added to the appropriate queue.

A timer for the "promotion event" is set in case of a backup task. After all events, the highest priority task in the upper queue (which may be primary or backup) on each core is dispatched, possibly preempting any running low-priority task. The algorithm sets the task's frequency to the precomputed frequency value if it is a primary task, otherwise it is executed at the maximum speed on the corresponding core.

When a task completes, the corresponding actions are shown in Algorithm 4. An *acceptance test* is run to check whether there is an error in the task's output. If no error is detected, then its alternate copy is cancelled (on the alternate core.) If an error is detected,

Algorithm 4 MPB-PS

Event: A task τ_i (B_i) is released at time t

if released task is *primary* **then**

Add τ_i to the upper queue at the proper priority level

else

Add B_i to the lower queue

Let Y_i be the promotion time computed at offline phase

Set a timer for promotion event at $t + Y_i$

end if

Dispatch highest priority tasks in the upper queues of both cores at pre-computed frequencies

Event: A task τ_i (B_i) completes

Run *acceptance test* for detection of transient fault in task τ_i (B_i)

if no error is detected **then**

Generate a "task cancelled" event for the alternate copy $B_i(\tau_i)$ on the alternate core

end if

Dispatch highest priority tasks in the upper queues of both cores at pre-computed frequencies

Event: Timer signals the promotion time of the backup task B_i

Move B_i from the lower to the upper queue on its core at the proper priority level

Dispatch B_i on its core if it has the highest priority in the upper queue at the maximum frequency

Event: A task τ_i (B_i) is cancelled

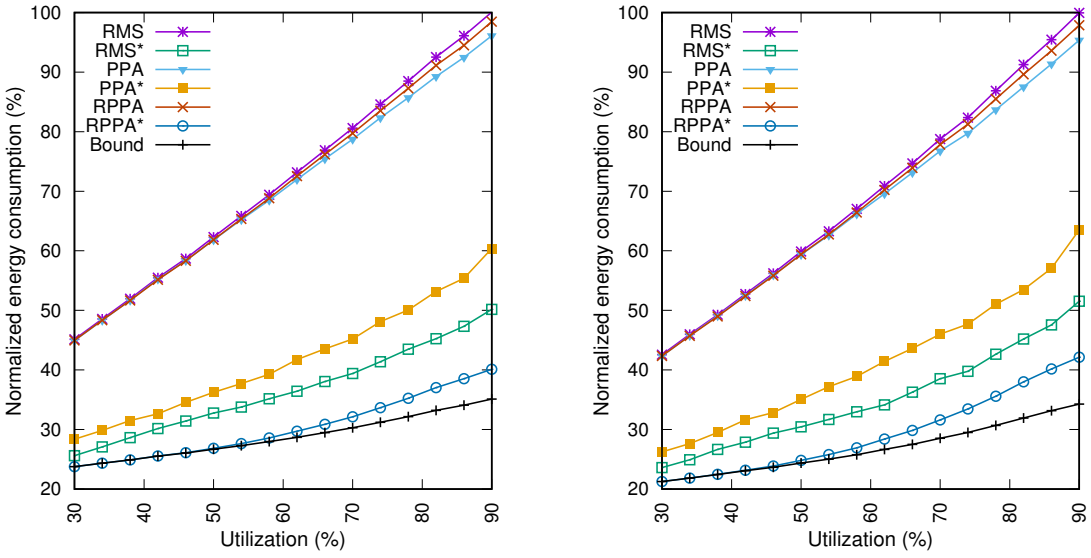
if task τ_i (B_i) is currently executing **then**

Dispatch the highest priority task in the upper queue of the core where τ_i (B_i) is cancelled

end if

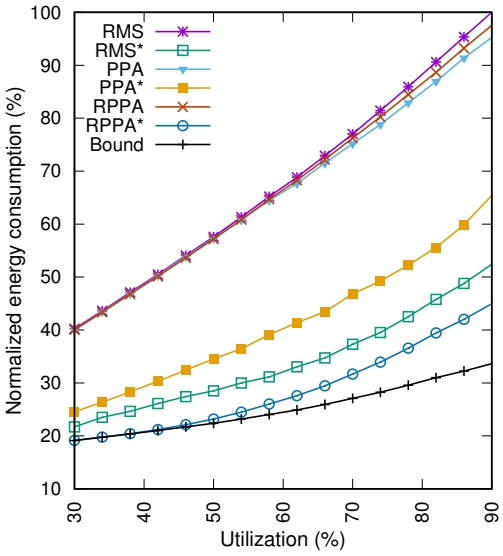
the algorithm does not take additional steps: it is expected that the alternate copy on the other core should produce correct results before the deadline. On the event when a backup task is promoted, it is moved to the upper queue.

This runtime algorithm ensures that a primary copy of a task is executed at the scaled speed, while backup tasks are executed at maximum speed of its core. Whenever one of them completes, the other one gets cancelled to conserve energy. In case of a permanent fault, the remaining core can execute one copy of each task without missing any deadline. At each invocation, this algorithm runs in $O(n)$ time, where n is the number of tasks on each core.

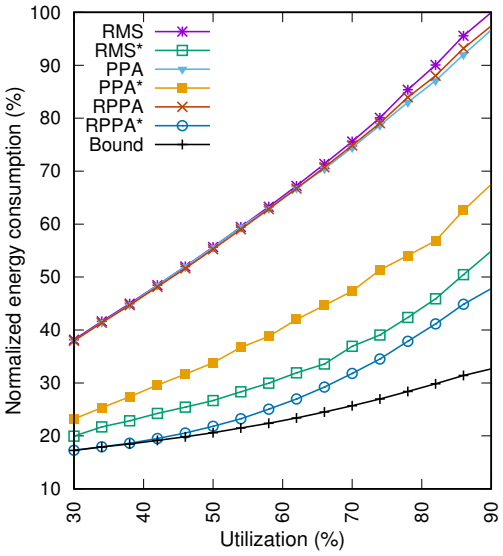


(a) Impact of Utilization ($f_{max}^{LP} = 0.6$)

(b) Impact of Utilization ($f_{max}^{LP} = 0.7$)



(c) Impact of Utilization ($f_{max}^{LP} = 0.8$)



(d) Impact of Utilization ($f_{max}^{LP} = 0.9$)

Figure 6.9: Impact of Utilization

6.5 Experimental Evaluation

We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. The tasks are partitioned using the LSPB scheme (Section 6.2.1). The

priority assignment schemes RMS, PPA and RPPA are evaluated, along with their variants which enable the dual queue based backup delaying technique (denoted as RMS*, PPA* and RPPA*, respectively). For all cases, we used DVFS to scale the frequency of the primary tasks and the *Sys-Clock* [85] algorithm was used for both cores.

We also implemented a scheme named *Bound*, where we remove all the backup tasks and allow the primary copies to scale their speed. This scheme does not offer any fault tolerance; but it is used as a theoretical lower bound on the energy consumption of all six schemes with fault tolerance features and backup task overheads.

We simulated dual core systems with $f_{max}^{HP} = 1.0$ and f_{max}^{LP} varied from 0.6 to 1.0. Due to space limitations, we will show the results for $f_{max}^{LP} = 0.8$, and analyze the impact of varying f_{max}^{LP} in a separate plot.

For each experiment, the simulator generates a task set containing n tasks, and a given total utilization, U. The utilization is calculated with respect to the LP core (which is more constrained in terms of performance) and normalized considering its maximum speed. Task periods are randomly chosen from a log uniform distribution ranging from 10 to 100. Hence, $U = (\sum \frac{C_i^{LP}}{T_i})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [214] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U.

It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [56]. Therefore, as in [6, 56], we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which models how execution time changes on the LP core for a given task, τ_i. Typical values for $tscale_i$ are reported to be in the range [1.4, 2.3] [56]. Moreover, following [6], we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as $\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$. From experimental measurements, it has been found that $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ [56]. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within the ranges

suggested in [56]. Specifically, $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ hold. We assume for all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. In addition, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$ for all experiments. We use task sets with $n = 10$ tasks, $f_{max}^{LP} = 0.8$ and $f_{max}^{HP} = 1.0$, unless otherwise stated. Every reported data point is the average of 1000 runs. We report the average energy consumption in fault-free executions, since faults are very rare events. The results in each plot are normalized with respect to the highest energy consumption of any scheme in that plot.

Impact of Utilization. Figures 6.9a, b, c, and d show the impact of utilization on normalized energy consumption for increasingly faster LP core (for f_{max}^{LP} set to 0.6, 0.7, 0.8 and 0.9, respectively). As expected, the normalized energy consumption of all schemes increase with the load. However, some schemes can save more energy than others. It also shows that, in all the cases the work-conserving RMS, PPA and RPPA schemes (without dual-queue based back-up delaying) perform worst and they perform very close to each other. This is because, in these schemes, the backup-delaying mechanism is not used, and therefore, backup copy executions overlapping with primaries are very frequent.

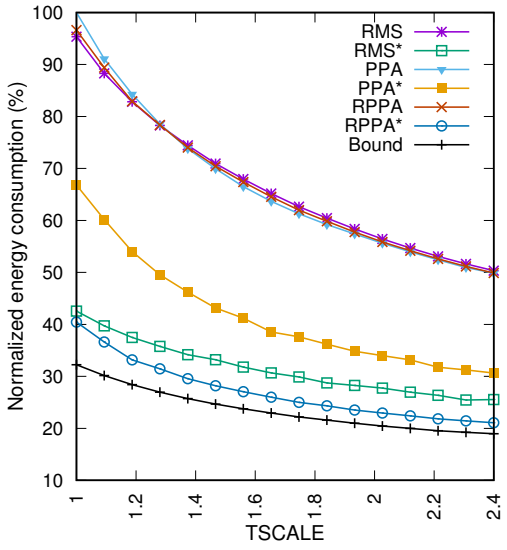
This problem is addressed by enabling the dual-queue based backup-delaying and non-work-conserving schedules in RMS*, PPA* and RPPA* schemes. As shown in Figure 6.9, PPA* can save more than 35% energy at average compared to the no-backup-delay schemes, throughout the entire range of system utilization values. RMS* performs even better than PPA* and it saves about 13% more energy on low-load and up to 20% for high-load task sets. Our proposed scheme, RPPA* performs the best and it saves 32% more energy than PPA*, and about 18% more energy than the RMS* scheme.

The performance level of PPA* warrants some elaboration. If backup copies are assigned lower priorities than primary copies, then due to the fact that primary copies are being slowed down through DVFS, we have only little room to delay the low-priority backups at run-time. This causes the backups to get activated early in the schedules, which results in increased energy consumption. On the other hand, the RPPA* scheme assigns higher priorities to the backup copies and the worst-case response time of backup tasks does not

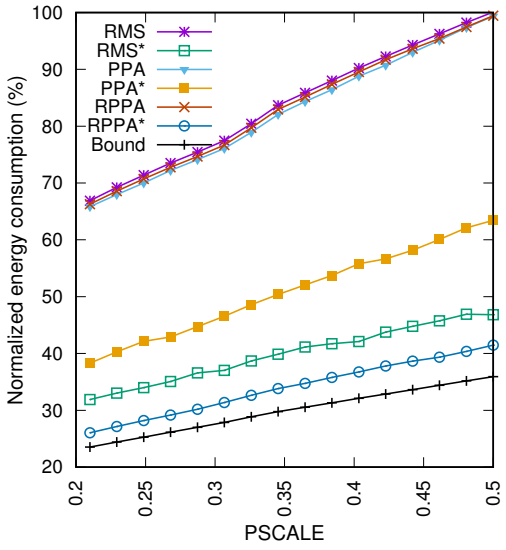
include the DVFS-enabled primary tasks, which allows the backups to remain in the lower queue for much longer time. This allows us to significantly delay, and in many cases, eventually cancel them. If a backup needs to be activated, it gets a higher priority on the processor, enabling it to finish still before deadline. This effect is reflected in the results and the RPPA* scheme performs much better throughout the entire spectrum. It performs very close to *Bound* for low utilization and it drifts away only moderately as the load increases. RMS* yields somewhat better results than PPA* (by virtue of the fact that some back-up tasks accidentally receive high priority based on their small periods), however, it is consistently worse than RPPA* in the entire spectrum. These observations hold true for all the utilization and the maximum speed configurations for the LP core.

Impact of *tscale*. Figure 6.10a shows the impact of varying *tscale* for the tasks, while keeping the system utilization at 65%. A lower *tscale* means the increase in task cycle requirements is modest on the LP core, indicating higher energy efficiency. The figure shows that the overall energy consumption of all schemes decrease as *tscale* increases. This is because since the utilization is fixed, a higher *tscale* value represents a lower number of cycles for the HP core, and the overall energy consumption decreases. The results show that the no-backup-delaying schemes. RMS, PPA and RPPA are performing the worst throughout the entire region. PPA* scheme improves energy consumption by about 35% compared to PPA. RMS* outperforms PPA* by a moderate amount of 20%. The best performing scheme is RPPA*, which is about 15% better than RMS* and it performs very close to *Bound* in the entire *tscale* region. This is because the coupling of RPPA priority assignment and dual-queue based backup-delaying techniques was able to cancel many of the backup executions and reduce the overall energy consumption close to *Bound* (which does not consider the back-ups).

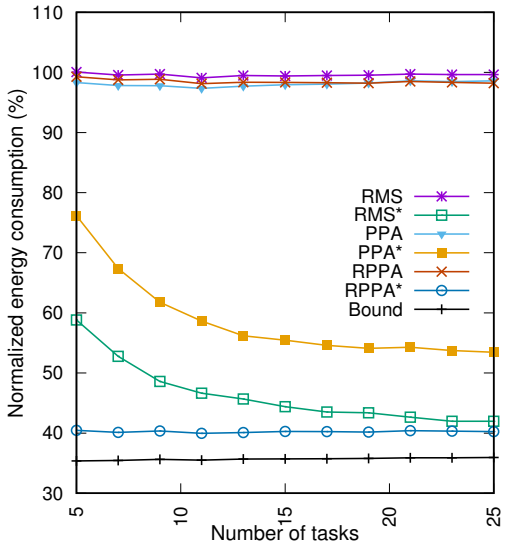
Impact of *pscale*. The impact of *pscale* on energy consumption is demonstrated in Figure 6.10b. As *pscale* grows, the LP core becomes less power efficient, and the effect is visible in the results. All the schemes show increased energy consumption with growing *pscale*, however, the RPPA* schemes can keep it very low. Throughout the entire spectrum



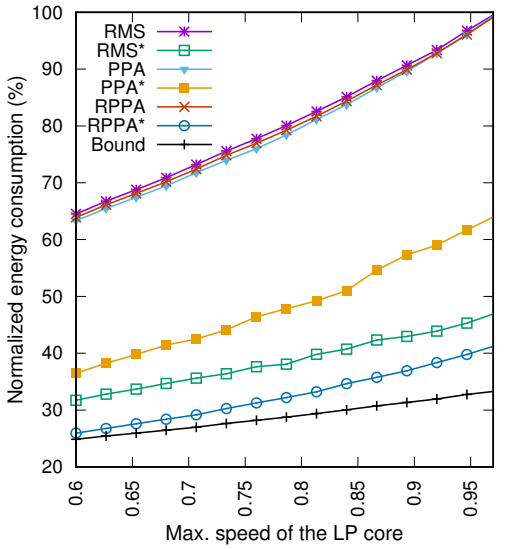
(a) Impact of tscale ($f_{max}^{LP} = 0.8$, Load = 65%)



(b) Impact of pscale ($f_{max}^{LP} = 0.8$, Load = 65%)



(c) Impact of Number of Tasks ($f_{max}^{LP} = 0.8$, Load = 65%)



(d) Impact of Max. speed of LP core (Load = 65%)

Figure 6.10: Impact of various system parameters

of *pscale*, RPPA* schemes perform about 35% better than PPA* schemes, and about 20% better than RMS* schemes. Due to the effective cancellation of back-ups, RPPA* was able to conserve a lot of energy and it performs very close to *Bound* (within 10%.)

Impact of Number of Tasks. Figure 6.10c shows the impact of number of tasks. It shows that RPPA* scheme performs the best throughout the entire spectrum and stays very close to *Bound*. The schemes with no backup-delaying shows very high energy consumption regardless of the number of tasks. The PPA* and RMS* schemes show higher energy consumption for small number of tasks, but as the number of tasks grow their energy consumption decreases. This is because, increasing the number of tasks while keeping the utilization same increases the granularity of the task set (giving smaller back-up tasks on the average), and the proposed algorithms are able to cancel back-ups more effectively. The observation that RPPA* outperforms PPA* and RMS* is still prevalent in these results.

Impact of the maximum speed of the LP core. Figure 6.10d shows the impact when we change the maximum speed of the LP core relative to the HP core, keeping the system utilization at 65%. As the LP core's maximum speed grows, its capacity and the system's actual workload also grows. That is reflected in the results by showing increased energy consumption for higher maximum speed of the LP core. The plot also shows that throughout our entire range of experiments, RPPA* schemes perform about 15% better than RMS* schemes, and about 35% better than PPA* schemes. *Bound* performs very close (within 2%) to RPPA* for lower values of f_{max}^{LP}, and it drifts away slowly as both cores' speeds become more similar.

6.6 Concluding Remarks

In this chapter, we investigated energy-aware scheduling of preemptive fixed-priority tasks upon heterogeneous cores. The fault tolerance requirements dictate scheduling a separate backup copy of each primary real-time job on a separate core. In addition to scaling the primary jobs through DVFS to save energy, we developed a comprehensive framework to maximize the cancellation opportunities for back-ups through the use of priority assignment and dual queue based task delaying. We evaluated the performance of the proposed schemes under different workload conditions. Our proposed Reverse Preference-Oriented

Priority Assignment (RPPA) scheme is shown to yield very high energy savings, by virtue of exploiting the dual-queue based delaying mechanism maximally.

Chapter 7: Quality-of-Control Management via Period Assignment in Real-Time Embedded Systems

In this chapter, we investigate a real-time control application implemented on a homogeneous multiprocessor and show how to choose periods for the control tasks in order to maximize the *quality-of-control* metric. This research effort is distinct from the previous chapters in the sense that we consider a *homogeneous* multiprocessor system and we optimize the *quality-of-control* metric (rather than energy consumption.) We published our results on this research in [11]. The rest of the chapter presents the details of our proposed heuristics and experimental evaluations.

7.1 Task Period Assignment on Multiprocessor Real-Time Control Systems

We study a real-time control-application implemented on a set of homogeneous multiprocessors with the objective of maximizing the quality-of-control by means of intelligent task-allocation and period assignment. A typical real-time control task is activated periodically, and it has to perform the sense-compute-actuate cycle with timing guarantees. However, at design time, there is often some flexibility for assigning the activation period (invocation rate) of a control task. Based on the dynamics of the state variable that the control task is associated with, the designer can derive bounds that must be satisfied by the sampling period. This flexibility can be exploited towards a better utilization of the system resources [200]. The controller's performance (sometimes referred to as *performance index*) is maximum when a sampling period is set to an ideal value (i.e., the performance of the digital controller approaches that of an ideal analog controller) [5]. Decreasing the period

below a limit does not further improve the performance; however the performance degrades as the period value is increased, due to the corresponding decrease in the frequencies of sampling and control signal. In fact, increasing the period further may cause stability problems. Hence, from the digital control point of view, there are constraints on the minimum and maximum periods that should be considered at design time.

In general, the computational capacity of the target system may not be sufficient to schedule all the tasks at their preferred (minimum) activation periods, or in other words, at their maximum invocation rates. It may often be necessary to increase the sampling periods of a number of tasks, thereby decreasing their utilizations. In these systems, the control cost increases sharply as the sampling period moves further from the lower bound. Most of the existing research on optimal period assignments in real-time control systems consider single-processor systems. However, modern control systems are increasingly implemented on multiprocessor platforms due to the increased performance requirements. Hence, the main objective of this research is to investigate the period assignment problem for homogeneous multiprocessor systems. In this work, we focus on partitioned scheduling, and we consider the problem of assigning periods to individual tasks and allocating them to M processors in order to minimize the overall control cost.

A solution approach for this problem is to first partition the tasks (using one of the heuristics based on initially assigning each task its maximum allowable period), and then on each processor, to make the period assignment to minimize the cost locally. We call this class of algorithms *local period assignment algorithms*. A limitation of this approach is that the algorithm does not have any global view of the task set and it cannot use that information towards minimizing control cost. In this study, we identify another approach which allows assigning the periods *before* the partitioning step, to find the periods that minimize the overall cost *globally*. This is achieved by solving the problem on a single-processor system which is K times faster ($K \leq M$), and then attempting partitioning with the periods obtained in that way. We call this scheme *Reduction-to-Single-Processor (RTSP)* technique. We propose two variants of this approach, one which gets the period

assignments by setting $K = M$, and another one which employs a binary search to find the most appropriate value for K.

Based on the Earliest-Deadline-First (EDF) scheduling on each processor [224], we perform an experimental evaluation of the local algorithms and the new *RTSP* approach. Our results indicate that *RTSP* can yield significant gains in minimizing the control cost compared to local algorithms, especially when the system is heavily loaded. We show that for most of the parameter spectrum, the control cost incurred by *RTSP* stays close to the theoretical lower bound. Our technique can be applied to any system with real-time control tasks where the cost is represented by a continuous and differentiable convex function. To the best of our knowledge, this is the first research effort that considers the period assignment problem on a multiprocessor real-time control system.

7.1.1 Models and Assumptions

In line with our general system model presented in Section 3.1, we consider a set of periodic real-time control tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$. However, this problem uses a different model than the rest of this dissertation– the set of real-time tasks are scheduled upon a *homogeneous* multiprocessor system that consists of M processors, denoted by the set $\Pi = \{\Pi_1, \ldots, \Pi_M\}$. We assume that the tasks are partitioned over the set of available processors, that is, migration of tasks from one processor to another is not allowed. The tasks allocated to a given processor are scheduled by the preemptive *Earliest-Deadline-First (EDF)* policy. It is known that the preemptive EDF is optimal for scheduling on uniprocessor systems, in that it can generate a feasible schedule as long as the total utilization of the tasks assigned to that CPU does not exceed 100% [224].

The worst-case execution time of task τ_i is denoted by C_i. The periods of the real-time tasks can be selected within a given range. Specifically, the *nominal* (minimum) period of τ_i is given by T_i^{min}; however its period T_i can be increased (at the cost of increased control cost), up to a limit T_i^{max}. T_i^{min} represents the minimum task period for which the

controller's performance is best – this is the point where the performance of the discrete-time optimal controller is identical with the performance of the continuous-time optimal controller [200]. As the task period increases, the difference between the discrete-time controller and continuous-time controller increases, and if it is larger than a certain upper bound (T_i^{max}), the stability can no longer be guaranteed. The ratio of the maximum period to the minimum period for a given control task τ_i is called the *elasticity factor* and is denoted by EF_i: $EF_i \triangleq \frac{T_i^{max}}{T_i^{min}}$. We consider implicit-deadline task sets where the relative deadline of each task is equal to its assigned period T_i.

The invocation frequency F_i of task τ_i is the inverse of its period, that is, $F_i = \frac{1}{T_i}$. The minimum and maximum task invocation frequencies are defined, respectively, as $F_i^{min} = 1/T_i^{max}$ and $F_i^{max} = 1/T_i^{min}$. The *utilization* of a task is defined as the ratio of its worst-case execution time to its assigned period. As the utilization is a function of the assigned period, we can define the minimum and maximum utilization of a task τ_i as $U_i^{min} = C_i/T_i^{max} = C_i * F_i^{min}$ and $U_i^{max} = C_i/T_i^{min} = C_i * F_i^{max}$, respectively.

The control cost performance of a task decreases with its invocation frequency. In literature, it is generally assumed that the control performance is a *convex* function of the invocation frequency [200]. The cost function we will use can be any generic convex function which we denote by $J(F)$, where F is the invocation frequency of the task. This function is specific to each task, and the task-specific constants can be used to formulate the function. Therefore, for a task τ_i, the cost function becomes $J_i(F_i)$. We require that $J_i(F_i)$ is convex, continuous, differentiable, and monotonically decreasing, as in [200]. The most common control cost function used in the real-time systems literature is the exponential decay function of the form $J_i(F_i) = \varphi_i \cdot e^{-\beta_i \cdot F_i}$ [200], where φ_i (the *weight*) and β_i are task specific constants. In the literature, other cost functions that incorporate additional factors (such as jitter) have been proposed (e.g., [227–229]), but in this study we are considering only the impact of the period assignment on the controller's performance.

Since the nominal (maximum) invocation frequency is the most preferable setting for

every task period, we assume that the cost for a task is 0 if it is running on its maximum frequency. Hence, $J_i(F_i^{max}) = 0 \ \forall i$. In other words, for the common exponential decay functions, we are using the form $J_i(F_i) = \varphi_i e^{-\beta_i F_i} - \varphi_i e^{-\beta_i F_i^{max}}$.

For a task set, the total (overall) cost will be the sum of the costs associated with each task.

$$J_{tot} = \sum_{\tau_i \epsilon \Gamma} J_i(F_i)$$

7.1.2 Minimizing Control Cost

7.1.2.1 Problem Definition

The problem addressed in this research, denoted by `MIN-COST-PARTITION`, can be stated as follows:

> Given M homogeneous processors and a set of periodic real-time tasks, how can we assign the task periods and partition the tasks among the processors such that:
>
> 1. the task set assigned to each processor can be scheduled in a feasible manner using the assigned frequencies, and,
>
> 2. the overall control cost across all processors is minimized.

This problem can be easily seen to be NP-Hard: Consider a special case where the allowable minimum and maximum periods are the same, i.e., $T_i^{min} = T_i^{max} \ \ \forall i$. In this case, the period of each is task is a fixed value, and the problem reduces to the problem of partitioning tasks with due dates on a multiprocessor platform, which is known to be NP-Hard in the strong sense [230]. Hence, `MIN-COST-PARTITION` is also NP-Hard in the strong sense.

Before discussing efficient heuristic approaches to address this problem, we briefly turn our attention to the single-processor case.

7.1.2.2 Optimization on a single processor with arbitrary convex control cost functions

The problem of minimizing the overall cost on a single processor has been considered in the seminal paper by Seto et al. [200]. We re-visit the same problem, and show how a more general solution can be obtained for arbitrary convex cost functions as long as the function is continuous and differentiable (the seminal work in [200] considered only exponential decay functions). We also incorporate lower bounds on the allowed task periods to indicate the task's preferred (nominal/minimum) periods. Finally, we show that this problem can be solved in polynomial-time (in the number of tasks), for arbitrary convex functions.

As stated previously, EDF is used as the scheduling algorithm for each processor. The schedulability condition for EDF is simple: the total utilization of the task set should not exceed 1.0. We can now formally define the problem on a single processor (denoted by `UNI-MIN-COST`) as follows:

$$\begin{aligned} \underset{F_1,..,F_n}{\text{minimize}} \quad & \sum_{\tau_i \in \Gamma} J_i(F_i) \\ \text{subject to:} \quad & \sum_{\tau_i \in \Gamma} C_i * F_i \leq 1.0 \\ & F_i^{min} \leq F_i \leq F_i^{max} \end{aligned} \tag{7.1}$$

where $J_i(F_i)$ is a continuous, differentiable, and monotonically decreasing convex function.

We now define $R_i(F_i) = -J_i(F_i)$. Note that $R_i(F_i)$ is a concave, continuous, and monotonically increasing function. Then, we define a new variable y_i such that $F_i = y_i + F_i^{min}$ and a new function $Q_i(y_i)$ such that

$$Q_i(y_i) = R_i(F_i) = R_i(y_i + F_i^{min})$$

Observe that $Q_i()$ is effectively a concave and non-decreasing *reward* function indicating

the controller's performance. The transformed problem is defined as:

$$\begin{aligned} \underset{y_1,..,y_n}{\text{maximize}} \quad & \sum_{\tau_i \in \Gamma} Q_i(y_i) \\ \text{subject to:} \quad & \sum_{\tau_i \in \Gamma} C_i * y_i \leq B \\ & 0 \leq y_i \leq F_i^{max} - F_i^{min} \\ \text{where} \quad & B = 1.0 - \sum_{\tau_i \in \Gamma} C_i * F_i^{min} \end{aligned} \tag{7.2}$$

Observe that $\sum_{\tau_i \in T} C_i * y_i$ has to be bounded by the constant B. If B is large enough, then clearly assigning $y_i = F_i^{max} - F_i^{min}\ \forall i$ would also maximize the objective function due to the non-decreasing nature of the reward function. Otherwise, this quantity B should be used in its entirety since the total reward never decreases by doing so. In this case, we obtain a constrained concave (non-linear) optimization problem with upper and lower bounds.

$$\begin{aligned} \underset{y_1,..,y_n}{\text{maximize}} \quad & \sum_{\tau_i \in \Gamma} Q_i(y_i) \\ \text{subject to:} \quad & \sum_{\tau_i \in \Gamma} C_i * y_i = B \\ & 0 \leq y_i \leq F_i^{max} - F_i^{min} \\ \text{where} \quad & B = 1.0 - \sum_{\tau_i \in \Gamma} C_i * F_i^{min} \end{aligned} \tag{7.3}$$

This problem turns out to be an instance of the general reward maximization problem considered in [231]. In fact, Aydin et al. [231] developed an iterative solution that considers the unconstrained optimization problem without the lower and upper bounds, and then iteratively adjusts the solution in time $O(n^2 \log n)$ to solve the constrained optimization

problem. We refer the reader to [231] for full details. Hence, `UNI-MIN-COST` can be also solved in time $O(n^2 \log n)$ for arbitrary convex functions.

7.1.2.3 Optimization on a multiprocessor platform

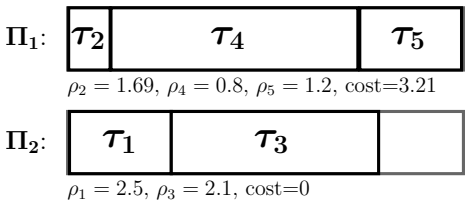
As shown in Section 7.1.2.1, overall control cost minimization on a multiprocessor setting with partitioned approach is, in general, intractable[1]. The problem has two important components, assigning periods to minimize the cost, and generating a feasible partitioning. While there are well-known effective partitioning algorithms (such as First-Fit and Worst-Fit), as the following example illustrates, using a different approach may significantly reduce the overall cost.

Table 7.1: Task set for Example 1

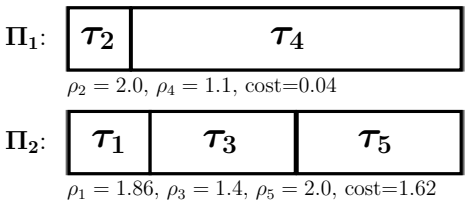
Name	β_i	φ_i	$F_i^{min}[Hz]$	$F_i^{max}[Hz]$	$C_i(ms)$
τ_1	0.3	4.42	1.7	2.5	105
τ_2	0.4	9.68	1.3	2.0	45
τ_3	0.6	3.56	1.4	2.1	260
τ_4	0.7	1.42	0.8	1.2	825
τ_5	0.8	9.86	1.2	2.5	220

Example 1. Table 7.1 shows a task set with 5 tasks ($\tau_1 - \tau_5$) that are to be scheduled on a system with two identical processors Π_1 and Π_2. The given task set can be partitioned among the processors and then the `UNI-MIN-COST` algorithm can be used to compute the periods to minimize the cost on each processor. However, different partitionings will result in different values of *overall cost*, as can be seen in Figure 7.1. If we set all the task utilizations to their minimum possible values (by using the maximum periods), and then apply a well-known partitioning heuristic, e.g., First-Fit-Decreasing (FFD), we obtain the

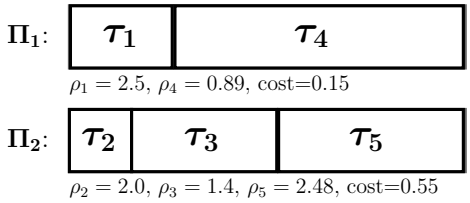
[1]It is worth mentioning that the optimal periods could be obtained for *global scheduling* by replacing the overall utilization bound 1.0 by M in Equation (7.1), and then using an optimal global scheduler such as *PFair* [37]. However, the focus of this research is the partitioned approaches which are known to have lower run-time overhead.



(a) Partitioning 1 using FFD/BFD, total cost = 3.21



(b) Partitioning 2 using WFD, total cost = 1.66



(c) Partitioning 3, total cost = 0.70

Figure 7.1: Partitioning Options for Example 1

partitioning in Figure 7.1a. After optimizing the frequencies on each individual processors with `UNI-MIN-COST`, we get a total overall cost of 3.21. Note that, if we had used Best-Fit-Decreasing (BFD), instead of FFD, we would obtain the same partitioning, incurring the same total cost. When Worst-Fit-Decreasing (WFD) is used as the partitioning heuristic, we found the partitioning in Figure 7.1b, yielding a total cost of 1.66, providing 48% improvement over FFD. However, there is a completely different partitioning, shown in Figure 7.1c, giving an overall cost of 0.70, and 58% improvement over WFD in terms of minimizing cost. In fact, this partitioning is produced by the RTSP* algorithm that we will develop in the next Section.

7.1.3 Proposed Algorithms

7.1.3.1 Local period assignment algorithms

For a given processor and its assigned task set, `UNI-MIN-COST` provides the optimal invocation frequencies for all tasks within their respective upper and lower bounds. Hence, an intuitive way to address the control cost minimization on a multiprocessor platform is to

perform task partitioning first using some well-known partitioning heuristic, and then call `UNI-MIN-COST` in order to assign periods for minimizing the cost *locally* on each processor.

Some well-known partitioning heuristics are *First-Fit*, *Worst-Fit*, and *Best-Fit*. In order to make the partitioning problem easier, respective minimum frequencies (maximum periods) are *tentatively* assigned to each task before partitioning is performed. It is well-known that in general, the performance of the partitioning heuristics improves if the tasks are ordered according to decreasing utilization [40]. This yields heuristics such as *First-Fit-Decreasing (FFD)*, *Worst-Fit-Decreasing (WFD)*, *Best-Fit-Decreasing (BFD)*. After the partitioning phase is over, the final period values to minimize the cost for each processor are obtained separately. This gives us a family of algorithms (that we call *local* algorithms); the algorithms differ only in the special partitioning heuristic used in the initial step. Depending on the specific partitioning algorithm used, we obtain different local algorithms called, for example, *WFD-Local, FFD-Local, BFD-Local*. Hence, a local period assignment algorithm has the following two steps:

1. Partition the given task set among the processors, by tentatively assigning $T_i = T_i^{max}$ to all tasks, and using a well-known heuristic such as FFD or WFD,

2. Invoke `UNI-MIN-COST` to assign optimal task period values in order to minimize cost on each processor, separately.

Note that if we do not have a feasible schedule after Step 1 (with maximum periods), we can declare that the task set is not schedulable using the local approach and the given heuristic. Otherwise, we call `UNI-MIN-COST` on each processor individually to adjust the periods of all tasks to optimal values locally. As it can be easily seen, the complexity of the local algorithms is determined primarily by Step 2, which is $O(M \cdot n^2 \cdot \log n)$.

7.1.3.2 Reduction to Single Processor (RTSP) Technique

Despite their intuitive nature, the local period assignment algorithms suffer from an obvious deficiency: The partitioning step is performed in a way which is completely incognizant of the specific cost functions. This is important because at the end of the partitioning, for example, tasks with very steep cost functions may be allocated to the same processor, limiting the potential of cost minimization on that specific processor through the invocation of `UNI-MIN-COST` algorithm.

An alternative approach that we consider is to *reverse* the order of steps undertaken by the local algorithms: If we can make the (near-)optimal period assignments in the first step by considering the *entire* task set, that is very likely to reduce the overall cost. This is, of course, based on the assumption that a feasible partitioning will be found with those suggested period assignments.

Specifically, in this approach, we first consider a *single-processor* Π_0 which is M times faster than each of the individual (unit-speed) processors in the original problem. Observe that Π_0 offers an aggregate computational capacity which is the same as the total computational capacity of all (unit-speed) processors in Π. The entire task set Γ is assigned to the faster processor Π_0, and `UNI-MIN-COST` is applied to get the optimal periods for all tasks. After that, one of the partitioning heuristics, (e.g., FFD, WFD, or BFD) can be used to partition the task set upon the original processor set Π. Note that, instead of using maximum periods for each task, RTSP uses the optimal periods while making partitioning decisions.

It is possible that, due to the inherent difficulty of partitioning problem, we will not be able to allocate some tasks with the assigned optimal periods. During the partitioning phase, the tasks that cannot be assigned to any processor are put in a list of *unassigned-tasks*. Next, this list is ordered according to decreasing value of utilization ($C_i * F_i$). From this list, tasks are picked one after the other and put to the processor that has the least value of *normalized local cost*, which is given by $(\sum_{\tau_i \in \Gamma_j} J_i(F_i) / \sum_{\tau_i \in \Gamma_j} J_i(F_i^{min}))$, where Γ_j is the

set of tasks already assigned to the j^{th} processor. Finally, `UNI-MIN-COST` is called on each processor to adjust the frequency for each task to preserve the feasibility with minimum cost increase. Algorithm 5 presents the pseudocode for RTSP.

Complexity: It can be seen that the complexity of the RTSP algorithm is dominated by the for loop in lines 20-24. In that loop, the algorithm `UNI-MIN-COST` is invoked separately for each of the M processors. Given that the number of tasks on each processor is bounded by n and, the complexity of `UNI-MIN-COST` is $O(n^2 \cdot \log n)$, the overall complexity of the algorithm is $O(M \cdot n^2 \cdot \log n)$, which is the same as that of the local algorithms.

Algorithm 5 RTSP

1: **Input:** Task set (Γ), Processor Set (Π)
2: **Output:** Task assignments and frequencies
3: $speed\_up \leftarrow M$
4: **for all** $\tau_i \in \Gamma$ **do**
5: $\quad C_i \leftarrow C_i / speed\_up$
6: $\quad F_i \leftarrow F_i^{min}$
7: **end for**
8: **if** $\sum \frac{C_i}{F_i} > 1$ **then**
9: $\quad$ **return** $failure$
10: **end if**
11: Call `UNI-MIN-COST` to get optimal frequencies $\{F_i^*\}$ for all tasks in Γ
12: **for all** $\tau_i \in \Gamma$ **do**
13: $\quad C_i \leftarrow C_i * speed\_up$
14: **end for**
15: Apply *partitioning scheme*, (e.g., `FFD`) to partition Γ upon Π with the $\{F_i^*\}$ frequencies, put infeasible tasks to *unassigned_list*
16: Order *unassigned_list* by decreasing value of $C_i * F_i$
17: **for all** $\tau_i \in$ *unassigned_list* **do**
18: $\quad$ Assign τ_i to the processor with the *least* value of $\sum_{\tau_i \epsilon \Gamma_j} J_i(F_i) / \sum_{\tau_i \epsilon \Gamma_j} J_i(F_i^{min})$, where Γ_j is the set of tasks already assigned to the j^{th} processor
19: **end for**
20: **for all** processors $\Pi_i \in \Pi$ **do**
21: $\quad$ Call `UNI-MIN-COST` to get optimal frequencies $\{F_i^*\}$ for all tasks on processor Π_i
22: $\quad$ **if** `UNI-MIN-COST` fails to return frequencies **then**
23: $\quad\quad$ **return** $failure$
24: $\quad$ **end if**
25: **end for**
26: **return** task-to-processor assignments $\{\Gamma_i^*\}$ and frequency assignments $\{F_i^*\}$

RTSP:* The RTSP scheme has a shortcoming which is immediately noticeable: the total utilization of the task set after the optimal period assignment on processor Π_0 is typically very close to M, the number of available unit-speed processors in the original problem. In other words, we attempt to partition a task set with total load equal to the available multiprocessor computing capacity. This is the *hardest* any partitioning problem can become. As a result, the list of *unassigned-tasks* is almost never empty; and those tasks are allocated to some processors using the *minimum normalized-cost* heuristic, before invoking `UNI-MIN-COST` on those processors and potentially modifying the originally assigned optimal period values to preserve the feasibility.

To overcome this shortcoming, we propose an improved version of RTSP, called RTSP*. Similar to RTSP, RTSP* also assumes a hypothetical processor which is x times faster than a unit-speed processor. But now, x is chosen as the largest value $\leq M$ for which the task set can be feasibly partitioned among the M processors, with the suggested periods. After the partitioning has been completed, `UNI-MIN-COST` is called individually on each processor to further reduce the cost, in case there is some idle capacity on certain processors. The value of x is determined using binary search. The search space ranges from $L = \sum_{\tau_i \in \Gamma} U_i^{min}$ to the total capacity of the system, M. The value of the lower bound L is determined by considering that even if all the tasks are assigned their maximum periods (minimum utilizations), they cannot be possibly scheduled on a platform which offers a computational capacity less than the total utilization $\sum_{\tau_i \in \Gamma} U_i^{min}$. The binary search algorithm continues to shrink the search space until the difference between the upper and lower bound comes within a pre-specified error-value, ε. Algorithm 6 presents the pseudocode. In fact, in Example 1, the best partitioning shown in Figure 7.1c was obtained using RTSP*.

Complexity: In each of the binary search iterations from line 6 to line 36, first `UNI-MIN-COST` is called for the entire task set, which takes time $O(n^2 \log n)$ for a given *speed_up* value. To calculate the number of iterations that the binary search takes before it converges to a value within ε of the actual value (or, fails), we divide the speedup value range into equal chunks, each of size ε, which is a configurable search parameter. Let the

total number of chunks be k, then $k = (M - \sum_{\tau_i \epsilon \Gamma} U_i^{min})/\varepsilon$, so the binary search would complete in at most $O(\log k)$ iterations. By factoring also the complexity of FFD in line 18, the overall complexity of RTSP* is found as $O(M \cdot n^2 \log n \cdot \log k)$ which is slightly higher than that of RTSP.

7.1.4 Evaluations and Discussions

We developed a discrete-event simulator to evaluate the proposed schemes. By varying various system parameters, the proposed schemes are evaluated through extensive simulations with synthetic tasks. In the simulator, the following schemes are implemented.

WFD-local: Based on their minimum activation rates, tasks are first partitioned to CPUs according to the well-known heuristic WFD (Worst-Fit Decreasing). Then, the `UNI-MIN-COST` problem is solved to minimize the cost on each CPU. We note that we also implemented other local algorithms such as FFD and BFD; but they were consistently outperformed by WFD which tends to distribute tasks more evenly across processors and thereby providing better opportunities for minimizing the total cost. Hence, in the evaluation section, we consider WFD as the representative of the local algorithms.

RTSP: The proposed *Reduction-to-Single-Processor (RTSP)* scheme, where a hypothetical processor with the speed of M is adopted to first optimize the activation rates for tasks. Then, based on the resulting optimal activation rates, tasks are partitioned upon M unit-speed CPUs according to the well-known heuristics (such as FFD, BFD and WFD). In what follows, we show only the results for FFD when used in conjunction with RTSP, because BFD and WFD perform similar or worse. Again, once tasks are mapped to CPUs, the activation rates for tasks are further adjusted by solving the `UNI-MIN-COST` problem on each CPU (See Algorithm 5).

RTSP*: This is the enhanced RTSP scheme. The speed of the hypothetical processor that ensures the feasible partitioning of all tasks is determined through binary search. See Algorithm 6 for details. Again, we use FFD to partition tasks. For binary search error threshold, we adopted $\varepsilon = 0.01$, which leads to acceptable running time and reasonable

Algorithm 6 RTSP*

Input: Task set (Γ), Processor Set (Π)
Output: Task assignments and frequencies
$upper \leftarrow M$
$lower \leftarrow \sum_{\tau_i \epsilon \Gamma} U_i^{min}$
$speed\_up \leftarrow lower$
loop
  for all $\tau_i \in \Gamma$ **do**
    $C_i \leftarrow C_i / speed\_up$
    $F_i \leftarrow F_i^{min}$
  end for
  if $\sum \frac{C_i}{F_i} > 1$ **then**
    return *failure*
  end if
  Call `UNI-MIN-COST` to get optimal frequencies $\{F_i^*\}$ for all tasks in Γ
  for all $\tau_i \in \Gamma$ **do**
    $C_i \leftarrow C_i * speed\_up$
  end for
  Apply *partitioning scheme*, (e.g., `FFD`) to partition Γ upon Π with the $\{F_i^*\}$ frequencies
  if a feasible partitioning is obtained **then**
    $lower \leftarrow speed\_up$
    if $upper - lower \leq \varepsilon$ **then**
      for all processors $\Pi_i \in \Pi$ **do**
        Call `UNI-MIN-COST` to get optimal frequencies $\{F_i^*\}$ for all tasks on processor Π_i
      end for
      return *task-to-processor assignments*
    else
      $speed\_up \leftarrow (upper + lower)/2$
    end if
  else
    if $speed\_up = lower$ **then**
      return *failure*
    end if
    $upper \leftarrow speed\_up$
    $speed\_up \leftarrow (upper + lower)/2$
  end if
end loop

accuracy for the results.

Bound: By assuming that all tasks run on a single hypothetical processor with the speed of M, we obtain the optimal activation rates of tasks by solving the `UNI-MIN-COST` problem,

which lead to the lowest possible total cost for all tasks. Basically, *Bound* can be considered as a yardstick algorithm to illustrate how well other schemes perform; because it does not consider the challenges of partitioning while keeping total computational capacity the same as the original M-processor system. No feasible partitioning on an M-processor system can yield an overall cost lower than the one provided by *Bound*, in other words, it represents the *lower bound* on the total cost that *any* algorithm can have.

Simulation Settings. In the simulations, we vary the following system parameters: the number of tasks in each task set n, the number of CPUs M, and tasks' elasticity factors. As the computational capacity of a multiprocessor system increases with the number of processors M, we represent the system load as relative to the maximum computational capacity on a given multiprocessor platform. This quantity representing the system load is called *normalized utilization* ($U_{tot}^{\mathcal{N}}$) , and is defined as the ratio of the total utilization to the number of processors, that is, $U_{tot}^{\mathcal{N}} = \frac{\sum U_i^{max}}{M}$.

For a given set of n, M, and $U_{tot}^{\mathcal{N}}$, the synthetic tasks are generated as follows. First, the total system utilization is found as $U_{tot} = U_{tot}^{\mathcal{N}} * M$. Then, the *RandFixedSum* algorithm in [214] is adopted to generate the maximum utilization U_i^{max} for each task τ_i such that: (1) the utilization is randomly chosen between 0 and 1 following a uniform distribution; and (2) the summation of the utilization values for all tasks in the set is exactly equal to the total utilization U_{tot}. Next, the minimum period T_i^{min} for each task τ_i is randomly selected within the range of $[10, 100]$ following a log-uniform distribution. A uniform elasticity factor, EF, is chosen for all tasks in a task set, where $T_i^{max} = EF \cdot T_i^{min}$. The inverse of minimum and maximum periods give maximum and minimum task frequencies (F_i^{max} and F_i^{min}), respectively. Finally, *worst-case execution time (WCET)* for each task can be calculated based on its minimum period and maximum utilization as $C_i = U_i^{max} * T_i^{min}$.

In our evaluations, we use the exponential decay functions [200] that have the form of $J_i(F_i) = \varphi_i e^{-\beta_i F_i} - \varphi_i e^{-\beta_i F_i^{max}}$. The constant term $\varphi_i e^{-\beta_i F_i^{max}}$ guarantees that $J_i(F_i^{max}) = 0$. We distinguish four different cost functions.

- **Type-0:** For all tasks in a task set, φ_i is set to 1, and β_i is set to 0.1 statically (*Uniform* cost functions).

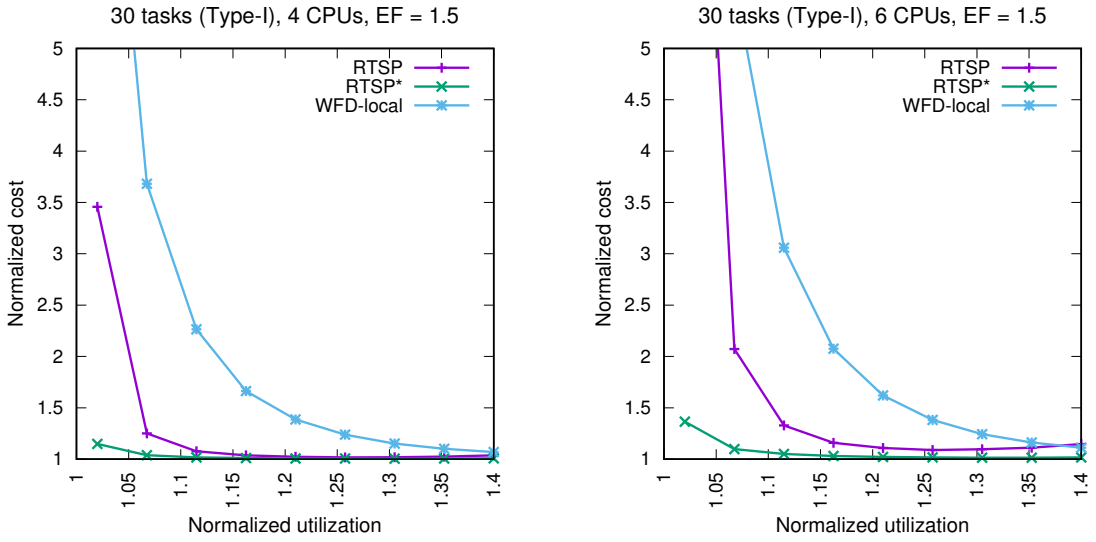
- **Type-1:** For any task in a task set, the *weight* φ is chosen randomly from a uniform distribution between 1 to 10. β is set to a constant value of 0.1 for all tasks (*Uniform weight - different decay parameter* cost functions).

- **Type-2:** For all tasks in a task set, the *weight* φ is statically set to 1, while β is randomly chosen within the range of $(0.0, 0.25]$ following a uniform distribution (*Different weight - uniform decay parameter* cost functions).

- **Type-3:** For any task in a task set, the *weight* φ is randomly chosen between 1 to 10, and β between 0.0 to 0.25, both following a uniform distribution (*heterogeneous* cost functions).

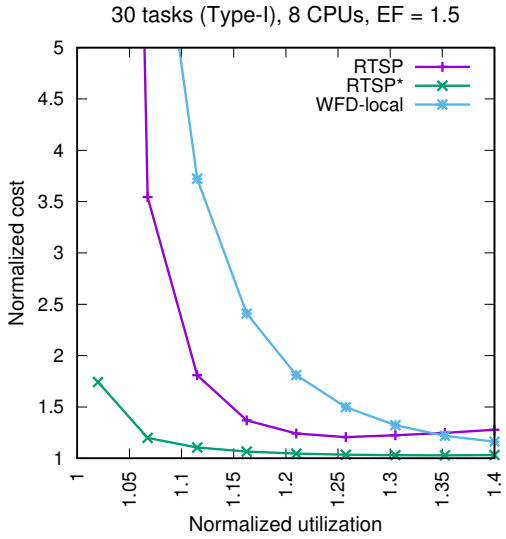
We first report results obtained with Type-1 cost functions in the evaluations. Later we comment on results with other types of cost functions. The performance of each scheme is reported as the *normalized cost*, which is defined as the ratio of the total cost incurred by the scheme to the total cost of *Bound*. Recall that *Bound* represents the scheme that assigns the periods by fully utilizing the total computational capacity of all M processors by ignoring the partitioning aspect; hence, it represents the upper bound on the performance of any partitioning-based algorithm. By definition, the normalized cost can never be smaller than 1.0. For each data point in the result plots, 25,000 synthetic task sets are generated and the average result of these task sets is reported.

Impact of System Utilization. Figure 7.2 shows the achieved normalized cost as a function of the utilization under different schemes for systems with different number of CPUs. We can see that the performance levels of both RTSP and RTSP* are better than that of WFD-local. The reason is that, compared to RTSP and RTSP* where tasks are mapped to processors based on their global optimal activation rates, WFD-local maps tasks to CPUs based on their minimum activation rates, which limits the opportunities for tasks



(a) 4 CPUs

(b) 6 CPUs



(c) 8 CPUs

Figure 7.2: Normalized cost for different schemes with varying system utilization; $EF = 1.5$

to exploit the entire system computing capacity for optimal activation rates. RTSP, while doing better than WFD in general, is outperformed by RTSP*. This comes from the fact that RTSP, in order to schedule the infeasible tasks, allocates them on certain processors based on a heuristic rule, and the activation rates of all tasks on those processors need to

be reduced, increasing the overall cost.

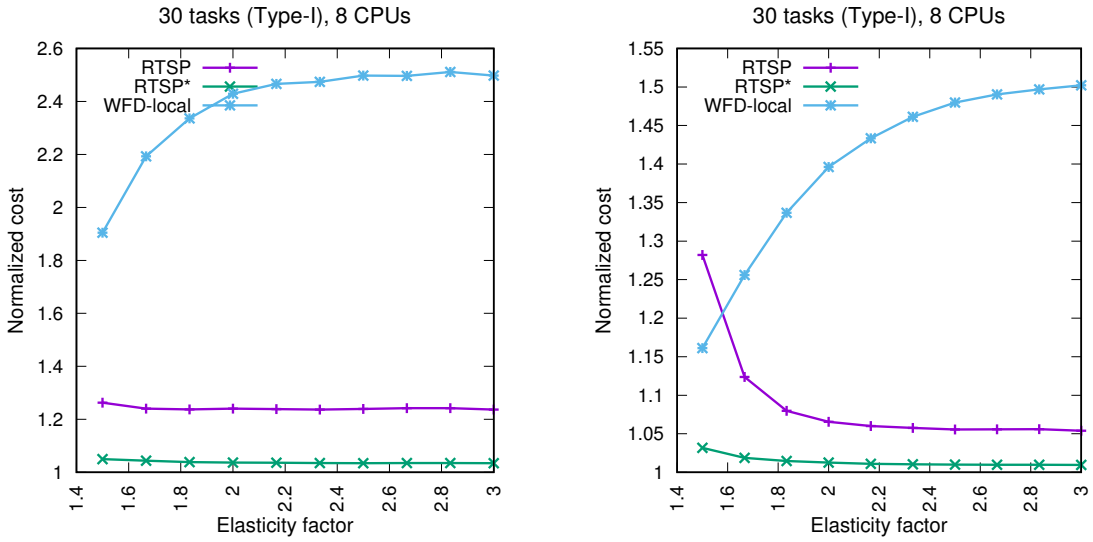
As another observation we can see that as the the system load (i.e., the *normalized utilization*) increases, the *normalized cost* for all schemes gradually approaches to the level of *Bound*, even though different schemes converge at different rates. In contrast, with modest load (e.g., less than 1.1), the normalized cost of the schemes (with respect to *Bound*) increases sharply.

This is because, when the normalized utilization exceeds 1.0 only by a small margin, *Bound* is able to guarantee the feasibility by increasing the periods only by very small amounts, yielding a total cost very close to zero. While the *absolute* cost of other schemes is also quite low in that region, the *normalized cost* increases given that *Bound*'s cost is close to zero. As the utilization increases the cost incurred by *Bound* also increases quickly and the normalized cost of the schemes improves.

With more (e.g., 8) processors in the system, there will be fewer number of tasks per CPU, and the performance difference of the schemes becomes much more pronounced. However, the performance of RTSP* is, even at medium loads, is very close to *Bound*; even on 8-processor systems, the difference is less than 20% as soon as the load exceeds 1.07, and becomes less than 5% at the increased load values.

Impact of Elasticity Factor. Next we consider the impact of different elasticity factors, which indicate the flexibility for assigning tasks' activation rates (i.e., periods), on the normalized cost of the system. Figures 7.3a and 7.3b show the results for two different system loads $U_{tot}^{\mathcal{N}} = 1.2$ and $U_{tot}^{\mathcal{N}} = 1.4$, respectively.

As the elasticity factor increases, we have more flexibility in terms of assigning larger periods, and the *absolute total cost* for all schemes tends to decrease. We can see that RTSP and RTSP* are, in most cases, able to maintain their normalized cost performance with increased elasticity factor; implying that the cost improvement due to the increased maximum period values is reflected in those schemes *in the same proportion* as in *Bound*. In fact, the *absolute cost* of WFD-Local monotonically decreases with increasing elasticity factor as well; however, the rate of decrase is *smaller* compared to that of *Bound* – that is



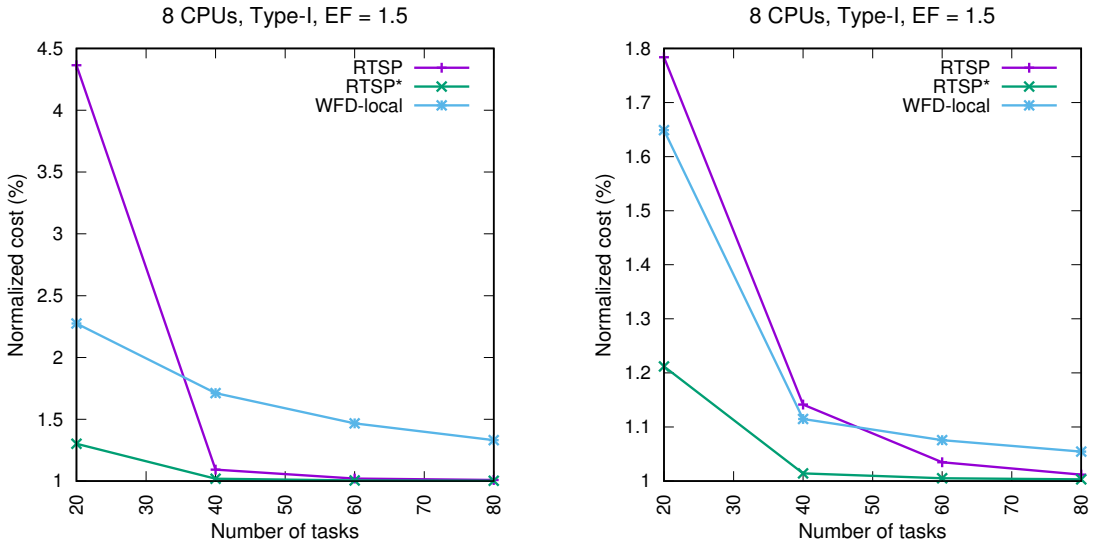
(a) $U_{tot}^{\mathcal{N}} = 1.2$ (b) $U_{tot}^{\mathcal{N}} = 1.4$

Figure 7.3: Impact of the elasticity factor

why we are observing an increase in *normalized cost.* in the plots.

We note that for small elasticity factor (e.g., $EF < 1.5$ for the case of $U_{tot}^{\mathcal{N}} = 1.4$), RTSP can perform slightly worse than WFD-local. The reason is that, for such inflexible task sets, RTSP cannot take much advantage of the global optimality and the WFD-local scheme tends to perform well. But, by virtue of using the globally assigned activation rates and the exact speedup ratio for making partitioning decisions, RTSP* can still outperform WFD-local. These plots again confirm that RTSP* stays within 5% of *Bound* for most of the spectrum. RTSP, performing close to *Bound* most of the time, drifts away for small EF values.

Impact of Number of Tasks. Figures 7.4a and 7.4b further show the normalized cost of tasks under different schemes when the number of tasks varies for systems with 8 CPUs. Here, we can see that, for a given system load ($U_{tot}^{\mathcal{N}} = 1.2$ or $U_{tot}^{\mathcal{N}} = 1.4$), the normalized cost for tasks under all schemes decreases as the number of tasks increases. The reason is that, with more tasks, the size (i.e., utilization) of each task gets smaller. With the smaller



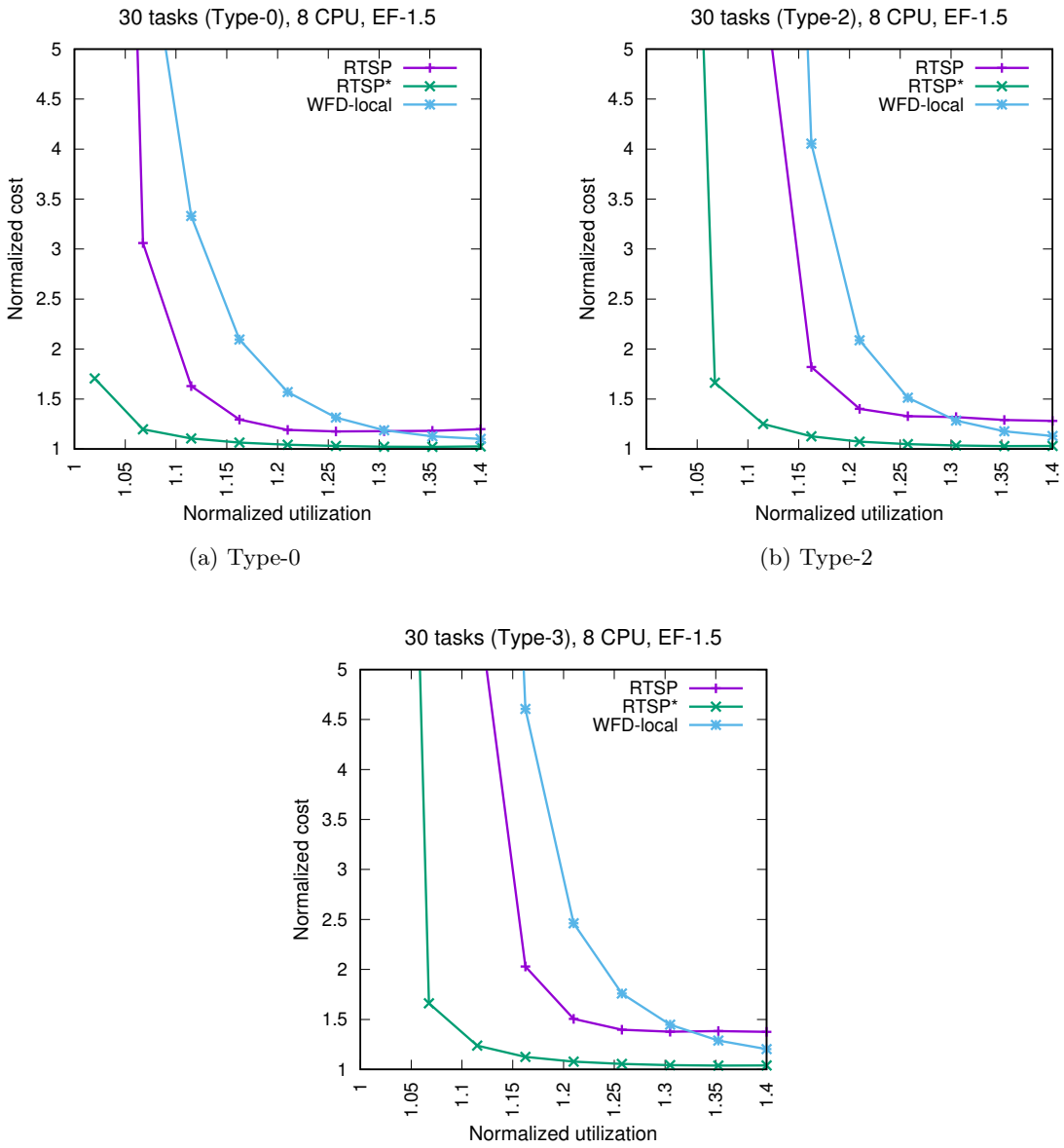
(a) $U_{tot}^{\mathcal{N}} = 1.2$ (b) $U_{tot}^{\mathcal{N}} = 1.4$

Figure 7.4: Impact of the number of tasks

granularity of task sizes, partitioning tasks to CPUs becomes relatively easier, even with small periods. When there are more than 60 tasks on a system with 8 CPUs, each CPU, on the average, can get more than 7 tasks, which enables RTSP and RTSP* to get very close to *Bound* (within 2%).

On the other hand, when there are fewer number of tasks, both RTSP and RTSP* start to drift away from *Bound*, but they still perform better than WFD-local when the system has more than 40 tasks. However, when the system has only 20 tasks, where each CPU has around 2-3 tasks, RTSP can perform even worse than WFD-local for the case of $U_{tot} = 1.2$ due to the increased granularity of tasks. For the case of slightly higher system load ($U_{tot} = 1.4$), WFD-local can perform as good as RTSP for 30 tasks (Figure 7.4b). Therefore, we can conclude that, the granularity of tasks affects the performance of RTSP the most, followed by WFD-local and RTSP*. However, RTSP* is quite close to *Bound*, except for the case of small number of tasks (20) when the difference is maximum (around 20%).

Impact of Cost Functions. When tasks have different types of cost functions, Figure 7.5



(a) Type-0

(b) Type-2

(c) Type-3

Figure 7.5: Impact of utilization on different types of task sets (30 tasks) on 8 CPUs

shows the normalized cost for tasks under different schemes when $EF = 1.5$, with varying system load. For all the types of cost functions considered in the evaluations, RTSP* performs the best and has performance level close to that of *Bound* (except when $U_{tot} \leq 1.1$). Moreover, the different types of exponential cost functions have similar impacts on the

performance of the proposed schemes regarding the normalized cost of tasks. In particular, as shown in Figure 7.2 for Type-1 tasks, RTSP is gradually outperformed by WFD with the increased load, with Type-0, Type-2 and Type-3 cost functions as well.

Comparison with the Optimal Solution. An interesting question is the relative performance of RTSP schemes with respect to the optimal *partitioning-based* algorithm. Obviously, since the problem is in general intractable, the optimal solution cannot be obtained in polynomial time. However, we implemented an exhaustive-search based solution that computes the optimal solution by enumerating all possible task-to-processor assignments, computing the best frequency assignments for each possible partitioning, and then picking up the best solution at the end. Due to the prohibitive computation time, the experiments were performed only for small number of tasks (6-12) running on relatively small number of processors (4). Each data points we show represents the average of 500 runs.

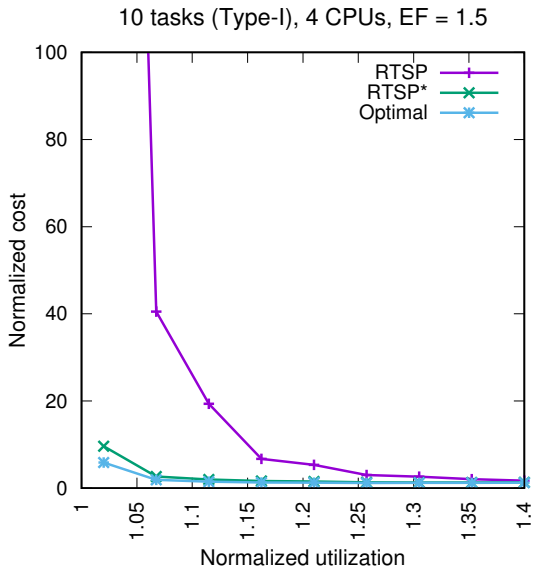


Figure 7.6: Impact of Utilization - Comparison to Exhaustive-Search Based Optimal Solution

In Figures 7.6 and 7.7, we show the impact of the utilization and the impact of the

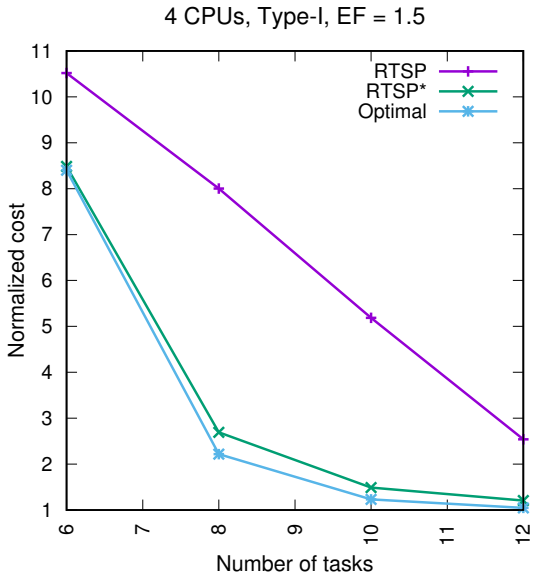


Figure 7.7: Impact of number of tasks - Comparison to Exhaustive-Search Based Optimal Solution; $U_{tot}^{\mathcal{N}} = 1.2$

number of tasks, respectively, on RTSP, RTSP* and Optimal scheme, with respect to *Bound*. It can be observed that RTSP* follows very closely the Optimal scheme even for medium normalized utilization and small number of tasks. It is also worth observing that even the performance of Optimal partitioning scheme starts to deviate significantly from that of *Bound* at low load or small number of tasks cases.

7.1.5 Concluding Remarks

Period assignment to real-time control tasks can make important difference in terms of control performance. While several studies considered the problem for uniprocessor settings and scheduling models, in this work, we considered multiprocessor platforms that are increasingly used in view of the performance requirements. We assumed partitioned EDF scheduling and arbitrary convex control cost functions.

Finding optimal solution to this problem on a multiprocessor platform is, in general, computationally intractable. We identified *local* period assignment algorithms as the ones

that first partition the tasks and then optimize the periods on each processor locally. To address the limitations of the local approaches, we proposed the Reduction-to-Single-Processor (RTSP) technique which first assigns tentative optimal periods by considering all the tasks at once on a hypothetical faster single processor, and then attempts partitioning with those period values. We developed two variants of the algorithm that differ on the way they determine the speedup factor for the hypothetical faster single processor. Our experimental results indicate that the RTSP technique generally outperforms the local algorithms, and follows the theoretical upper bound on the control performance closely.

Chapter 8: Conclusions

In this chapter, we summarize the main contributions of this dissertation and offer a few future research directions. The main theme of this research was the minimization of energy consumption for reliable real-time systems implemented upon heterogeneous dual-core systems. We also studied maximizing quality-of-control for homogeneous multiprocessor control systems.

8.1 Summary of the Dissertation's Contributions

The research addressed in this dissertation has two main parts. The first is the energy-aware fault-tolerant scheduling of real-time tasks, and the second is the quality-of-control maximization for multiprocessor control systems. For the first part, we addressed the three interconnected dimensions of the general problem, i.) energy management, ii.) fault tolerance, and iii.) strict timing guarantees. We developed several algorithms and techniques which achieve good performance under a wide range of system configurations and application models. We demonstrated the effectiveness of our techniques by using extensive simulation studies with synthetic task sets. The second part of the dissertation addressed digital control systems implemented upon homogeneous multiprocessors and opted to maximize its quality-of-control by making task partitioning and period assignment decisions. Extensive simulation experiments were conducted to test the effectiveness of the algorithms we proposed.

8.1.1 Energy-Aware Fault-Tolerant Scheduling of Real-Time Tasks

We developed energy-aware scheduling algorithms for fault-tolerant real-time systems implemented on single-ISA heterogeneous dual-core systems. We considered three different

application task models separately and proposed techniques for energy-efficient and reliable operation, as described below:

- *Independent frame-based tasks.* In this problem, we addressed the scheduling of a given set of independent real-time tasks satisfying their deadline and reliability constraints, while also minimizing the overall energy consumption of the system. In one sub-problem, we designated one processing core as the "primary" core and the other core as the "spare" core (called "standby-sparing" configuration). A spare copy of each task is created and allocated to the spare core. Moreover, the primary core can use DVFS to select an operating frequency that can save energy, and the spare core is equipped with DPM in order to put the core into a low-power state. Under such settings, we developed algorithms to minimize the overall energy consumption by determining: i) which core should be designated as the primary and spare, and, ii) what processing frequency assignments should be made to the tasks on the primary core. In the second direction, we also addressed minimizing energy consumption in the context of *mixed primary/backup (MPB) scheduling* in which, each processing cores are allocated a "mix" of primary and backup copies of tasks, with the constraint that the primary and backup copies of the same task must be placed on separate cores. In order to minimize the overall energy consumption in such a setting, we developed several algorithms to determine: i) the allocation of tasks under reliability constraints, and, ii) the processing frequency assignment to individual tasks. We presented extensive simulation studies which validate our findings.

- *Frame-based tasks with precedence constraints.* In this problem, we considered real-time tasks with precedence constraints and fault tolerance requirements, and developed energy-aware scheduling algorithms on heterogeneous dual-core systems using the MPB approach. Similar to the previous problem, each task has a backup copy placed on the alternate core and both cores support DVFS and DPM; however, tasks

have additional constraints due to their data-dependencies between them, which govern the execution order of the tasks. We developed a technique called CPSS along with other algorithms which can minimize the energy consumption by determining how to schedule the primary and backup tasks (along with execution frequencies for primary tasks) such that all timing and reliability guarantees are met. Extensive simulation studies were performed in order to verify the performance of the proposed algorithms.

- *General periodic tasks.* In this problem, we considered general periodic real-time tasks with fault tolerance requirements. Again, we used the MPB approach and assumed that both of the processing cores were equipped with DVFS and DPM for energy savings, however, in this problem we generalized our real-time task model such that tasks can have arbitrary periods. Each periodic task is provisioned with a backup copy on the alternate core to satisfy reliability constraints. We considered fixed-priority systems in which we assign an execution priority to each task offline (before execution). To manage the overall energy consumption, we used two main mechanisms: i.) the primary tasks were executed at low voltage/frequency levels using DVFS, and, ii.) the backup copies were delayed to the extent it is possible to enable their cancellation in case the primary completes without a fault. In this setting, the problem was to determine the task allocation, priority assignment, execution frequency assignment and backup delaying mechanism which yields the minimal energy consumption for the overall system. We conducted extensive simulation studies which show that our main proposed algorithm performs very close to the optimal theoretical limit.

8.1.2 Quality-of-Control Management via Period Assignment

Our objective in this problem was to maximize the overall quality-of-control, by means of allocating tasks to processors and choosing a suitable invocation rate (period) for each of the control tasks. In this research, we considered a homogeneous multiprocessor system. The quality-of-control of a task is defined by a concave relationship to the invocation rate

(period) of each real-time control task. We proposed a family of heuristics (based on reduction from a multiprocessor system to a more powerful single-processor system) which maximize the overall quality-of-control of the system while guaranteeing a feasible schedule by determining: i) the task to processor allocation, and, ii) the invocation-rate (or, period) of each control task. We used various types of convex cost functions to mimic various application models and conducted simulation experiments which demonstrate considerable improvement in the quality-of-control for a wide range of digital control systems.

8.2 Future Work

Embedded systems have increased presence in almost every aspect of our lives, and some of them exhibit real-time operation and safety-criticality features. Moreover, energy efficiency is a quite common design and operation requirement. This dissertation effort explored a number of open research problems that considered the subtle interplay among reliability and energy-efficiency requirements of real-time embedded systems implemented upon heterogeneous dual-core systems. We can offer the following future directions to extend our research presented in this dissertation.

8.2.1 Energy-Aware Fault-Tolerant Scheduling on Heterogeneous Cluster Based Multicores

We addressed energy-aware scheduling of heterogeneous multicores with fault tolerance and real-time guarantees, and to the best of our knowledge, this has been the first research of its kind. As the first steps, we developed task partitioning and scheduling algorithms on a heterogeneous platform consisting of only two processing cores. However, in practice, many heterogeneous multicore processors are cluster based, where all cores in a given cluster share the same characteristics and different clusters may have a wide range of varying power-performance characteristics (e.g., Apple M1 and Samsung Exynos 2100 Big-Little clusters.) In order to develop practical applications on such cluster based multicores, one needs to develop energy-efficient scheduling algorithms for systems with for more than two cores.

Our algorithms can be used as a starting point of such studies and can be extended to support heterogeneous cluster based multicores. One straightforward, but not necessarily optimal, approach would be to use mixed-primary/backup scheduling with Big and Little clusters, where we allocate each pair of primary and backup copies to different type of clusters, and exploit DVFS and DPM to save energy, while respecting the hard deadlines of real-time applications. However, more detailed studies are needed to assess the performance of this approach and investigate more sophisticated techniques.

8.2.2 Energy-Aware Fault-Tolerant Scheduling of Dynamic-Priority Periodic Tasks

While we addressed the problem of energy-efficient scheduling of general periodic tasks on heterogeneous multicores with fault tolerance, our system model assumed a fixed-priority system, meaning the execution priorities of various periodic tasks are assigned statically, and they remain fixed throughout runtime. However, it is known that dynamic-priority systems, where task priorities can vary at run-time, in general, are capable of better using the processors' computational capacity. This implies that dynamic-priority systems could be exploited to produce scheduling algorithms which are even more energy-efficient than our proposed framework. Further analysis is warranted to assess whether extra complexity which often accompanies dynamic-priority systems' design and operation is justifiable with respect to the additional energy savings it might offer.

Bibliography

Bibliography

[1] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[2] R. Xu, D. Mossé, and R. Melhem, "Minimizing expected energy consumption in real-time systems through dynamic voltage scaling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 4, p. 9, 2007.

[3] C. Rusu, R. Melhem, and D. Mossé, "Maximizing rewards for real-time applications with energy constraints," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 2, no. 4, pp. 537–559, 2003.

[4] V. Devadas and H. Aydin, "On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 31–44, 2010.

[5] J. W. S. Liu, *Real-Time Systems*. Pearson Education, 2000. [Online]. Available: https://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/8177585754

[6] A. Roy, H. Aydin, and D. Zhu, "Energy-aware standby-sparing on heterogeneous multicore systems," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[7] A. Roy, H. Aydin, and D. Zhu, "Energy-efficient primary/backup scheduling techniques for heterogeneous multicore systems," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2017, pp. 1–8.

[8] A. Roy, H. Aydin, and D. Zhu, "Energy-efficient fault tolerance for real-time tasks with precedence constraints on heterogeneous multicore systems," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2019, pp. 1–8.

[9] A. Roy, H. Aydin, and D. Zhu, "Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems," in *2020 Eleventh International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2020, pp. 1–8.

[10] A. Roy, H. Aydin, and D. Zhu, "Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems," *Sustainable Computing: Informatics and Systems*, vol. 29, p. 100474, 2021.

[11] A. Roy, H. Aydin, and D. Zhu, "On task period assignment in multiprocessor real-time control systems," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. ACM, 2016, pp. 151–160.

[12] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[13] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Real Time Systems Symposium, 1989., Proceedings.* IEEE, 1989, pp. 166–171.

[14] N. Fisher and S. Baruah, "Rate-monotonic scheduling," in *Encyclopedia of Algorithms*. Springer, 2008, pp. 1–99.

[15] N. C. Audsley, A. Burns, M. Richardson, and A. Wellings, "Deadline monotonic scheduling," 1990.

[16] J. P. Lehoczky, "Enhanced aperiodic responsiveness in hard real-time environment," in *Proceedings of IEEE Real-Time Systems Symposium'87*, 1987, pp. 261–270.

[17] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[18] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[19] M. Spuri and G. C. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling." in *RTSS*, 1994, pp. 2–11.

[20] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, no. 2, pp. 179–210, 1996.

[21] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Carnegie-Mellon University Pittsburg PA School of Computer Science, Tech. Rep., 1993.

[22] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications." in *ICMCS*, 1994, pp. 90–99.

[23] X. Qin, H. Jiang, and D. R. Swanson, "An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems," in *Parallel Processing, 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 360–368.

[24] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Computing*, vol. 32, no. 5-6, pp. 331–356, 2006.

[25] Q. Zheng and B. Veeravalli, "On the design of communication-aware fault-tolerant scheduling algorithms for precedence constrained tasks in grid computing systems with dedicated communication devices," *Journal of Parallel and Distributed Computing*, vol. 69, no. 3, pp. 282–294, 2009.

[26] A. Benoit, M. Hakem, and Y. Robert, "Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems," *Parallel Computing*, vol. 35, no. 2, pp. 83–108, 2009.

[27] B. Zhao, H. Aydin, and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 2, p. 23, 2013.

[28] L. Wang, G. Von Laszewski, J. Dayal, and F. Wang, "Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 368–377.

[29] Y. C. Lee and A. Y. Zomaya, "Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling," in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 2009, pp. 92–99.

[30] E. W. Dijkstra, "Cooperating sequential processes," in *The origin of concurrent programming*. Springer, 1968, pp. 65–138.

[31] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[32] P. B. Hansen, *Operating system principles*. Prentice-Hall, Inc., 1973.

[33] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[34] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*. IEEE, 1990, pp. 191–200.

[35] J. B. Goodenough and L. Sha, *The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks*. ACM, 1988, vol. 8, no. 7.

[36] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[37] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[38] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 101–110.

[39] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[40] J. M. López, J. L. Díaz, and D. F. García, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Systems*, vol. 28, no. 1, pp. 39–68, 2004.

[41] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," *Handbook on scheduling algorithms, methods, and models*, pp. 30–1, 2004.

[42] S. Baruah, "Partitioned edf scheduling: a closer look," *Real-Time Systems*, vol. 49, no. 6, pp. 715–729, 2013.

[43] I. Present, "Cramming more components onto integrated circuits," *Readings in computer architecture*, vol. 56, 2000.

[44] T. Mitra, "Heterogeneous multi-core architectures," *Information and Media Technologies*, vol. 10, no. 3, pp. 383–394, 2015.

[45] A. Vajda, "Multi-core and many-core processor architectures," in *Programming Many-Core Chips*. Springer, 2011, pp. 9–43.

[46] S. Pasricha and N. Dutt, "Orb: An on-chip optical ring bus communication architecture for multi-processor systems-on-chip," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. IEEE Computer Society Press, 2008, pp. 789–794.

[47] A. Jantsch, H. Tenhunen *et al.*, *Networks on chip*. Springer, 2003, vol. 396.

[48] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, "Multi-core cache hierarchies," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–153, 2011.

[49] S. Devadas, "Toward a coherent multicore memory model," *Computer*, vol. 46, no. 10, pp. 30–31, 2013.

[50] R. Manikantan, K. Rajan, and R. Govindarajan, "Nucache: An efficient multicore cache organization based on next-use distance," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 243–253.

[51] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 45–54.

[52] V. Devadas and H. Aydin, "Coordinated power management of periodic real-time tasks on chip multiprocessors," in *Green Computing Conference, 2010 International*. IEEE, 2010, pp. 61–72.

[53] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 329–342, 2012.

[54] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 190–197.

[55] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1131–1136.

[56] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–10.

[57] T. Mitra, T. S. Muthukaruppan, A. Pathania, M. Pricopi, V. Venkataramani, and S. Vishin, "Power management of asymmetric multi-cores in the dark silicon era," in *The Dark Side of Silicon*. Springer, 2017, pp. 159–189.

[58] J. Spasic, D. Liu, and T. Stefanov, "Energy-efficient mapping of real-time applications on heterogeneous mpsocs using task replication," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2016, p. 28.

[59] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. IEEE, 2004, pp. 64–75.

[60] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[61] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Méhaut, "Performance/energy trade-off in scientific computing: the case of arm big. little and intel sandy bridge," *IET Computers & Digital Techniques*, vol. 9, no. 1, pp. 27–35, 2014.

[62] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little technology," *Samsung White Paper*, 2012.

[63] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee, "Power-aware task scheduling for big. little mobile processor," in *SoC Design Conference (ISOCC), 2013 International*. IEEE, 2013, pp. 208–212.

[64] S. Kamdar and N. Kamdar, "big. little architecture: Heterogeneous multicore processing," *International Journal of Computer Applications*, vol. 119, no. 1, 2015.

[65] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, vol. 17, 2011.

[66] S. Variable, "A multi-core cpu architecture for low power and high performance," *Whitepaper-http://www.nvidia.com*, 2011.

[67] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "Sparta: runtime task allocation for energy efficient heterogeneous many-cores," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2016, p. 27.

[68] J. Yun, J. Park, and W. Baek, "Hars: A heterogeneity-aware runtime system for self-adaptive multithreaded applications," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[69] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2013, pp. 1–9.

[70] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 161–176, 2014.

[71] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 45, 2016.

[72] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 1995, pp. 374–382.

[73] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *Real-Time Systems, 13th Euromicro Conference on, 2001.* IEEE, 2001, pp. 225–232.

[74] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 4, p. 31, 2009.

[75] M. A. Awan and S. M. Petters, "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems," in *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*. IEEE, 2011, pp. 92–101.

[76] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo, "Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*. IEEE, 2006, pp. 408–417.

[77] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *computer*, vol. 36, no. 12, pp. 68–75, 2003.

[78] J.-J. Chen and T.-W. Kuo, "Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor," *ACM SIGPLAN Notices*, vol. 41, no. 7, pp. 153–162, 2006.

[79] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 8, no. 3, pp. 299–316, 2000.

[80] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 89–102.

[81] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584–600, 2004.

[82] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 1. ACM, 2001, pp. 50–61.

[83] H. Aydin, V. Devadas, and D. Zhu, "System-level energy management for periodic real-time tasks," in *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 313–322.

[84] D. Zhu, R. Melhem, and D. Mossé, "The effects of energy management on reliability in real-time embedded systems," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* IEEE, 2004, pp. 35–40.

[85] S. Saewong and R. Rajkumar, "Practical voltage-scaling for fixed-priority rt-systems," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE.* IEEE, 2003, pp. 106–114.

[86] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: a survey," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, p. 7, 2016.

[87] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1.* IEEE Computer Society, 2003, p. 10918.

[88] W. Kim, J. Kim, and S. L. Min, "Preemption-aware dynamic voltage scaling in hard real-time systems," in *Proceedings of the 2004 international symposium on Low power electronics and design.* ACM, 2004, pp. 393–398.

[89] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling and power management for portable systems," in *Proceedings of the 38th annual Design Automation Conference.* ACM, 2001, pp. 524–529.

[90] M. Kim and S. Ha, "Hybrid run-time power management technique for real-time embedded system with voltage scalable processor," *ACM SIGPLAN Notices*, vol. 36, no. 8, pp. 11–19, 2001.

[91] J. Zhuo, C. Chakrabarti, and N. Chang, "Energy management of dvs-dpm enabled embedded systems powered by fuel cell-battery hybrid source," in *Proceedings of the 2007 international symposium on Low power electronics and design.* ACM, 2007, pp. 322–327.

[92] H. Cheng and S. Goddard, "Sys-edf: a system-wide energy-efficient scheduling algorithm for hard real-time systems," *International Journal of Embedded Systems*, vol. 4, no. 2, pp. 141–151, 2009.

[93] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 2008, pp. 34–44.

[94] V. Devadas and H. Aydin, "Dfr-edf: A unified energy management framework for real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 121–130.

[95] V. Devadas and H. Aydin, "On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 31–44, 2012.

[96] R. Jejurikar and R. Gupta, "Procrastination scheduling in fixed priority real-time systems," *ACM Sigplan Notices*, vol. 39, no. 7, pp. 57–66, 2004.

[97] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005, pp. 111–116.

[98] G. Quan, L. Niu, X. S. Hu, and B. Mochocki, "Fixed priority scheduling for reducing overall energy on variable voltage processors," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*. IEEE, 2004, pp. 309–318.

[99] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2009, pp. 193–202.

[100] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing for fixed-priority real-time task sets," *Journal of Sustainable Computing*, vol. 6, pp. 81–93, 2015.

[101] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[102] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 9–pp.

[103] G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada, "Practical energy-aware scheduling for real-time multiprocessor systems," in *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on*. IEEE, 2009, pp. 383–392.

[104] D. Zhu, R. Melhem, and B. R. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 686–700, 2003.

[105] J. Lu and Y. Guo, "Energy-aware fixed-priority multi-core scheduling for real-time systems," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, vol. 1. IEEE, 2011, pp. 277–281.

[106] H. Xu, F. Kong, and Q. Deng, "Energy minimizing for parallel real-time tasks based on level-packing," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 98–103.

[107] G. Chen, K. Huang, and A. Knoll, "Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 111, 2014.

[108] S. Pagani and J.-J. Chen, "Energy efficiency analysis for the single frequency approximation (sfa) scheme," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, p. 158, 2014.

[109] M. E. Gerards, J. L. Hurink, and J. Kuper, "On the interplay between global dvfs and scheduling tasks with precedence constraints," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1742–1754, 2015.

[110] K. Srinivasan and K. S. Chatha, "Integer linear programming and heuristic techniques for system-level low power scheduling on multiprocessor architectures under throughput constraints," *INTEGRATION, the VLSI journal*, vol. 40, no. 3, pp. 326–354, 2007.

[111] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 81–92.

[112] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "Oscar api for real-time low-power multicores and its performance on multicores and smp servers." in *LCPC*, vol. 5898. Springer, 2009, pp. 188–202.

[113] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 125–138.

[114] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara, "Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 184–198.

[115] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 23–32.

[116] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2012, pp. 345–350.

[117] D. K. Pradhan, *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[118] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 658–671, 1982.

[119] R. K. Iyer, D. J. Rossetti, and M.-C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 3, pp. 214–237, 1986.

[120] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[121] F. Many and D. Doose, "Scheduling analysis under fault bursts," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 113–122.

[122] M. A. Haque, H. Aydin, and D. Zhu, "Real-time scheduling under fault bursts with multiple recovery strategy," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 63–74.

[123] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich *et al.*, "Design and architectures for dependable embedded systems," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*. IEEE, 2011, pp. 69–78.

[124] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll, "Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*. IEEE, 2011, pp. 247–256.

[125] H. Aydin, "Exact fault-sensitive feasibility analysis of real-time tasks," *IEEE Transactions on Computers*, vol. 56, no. 10, 2007.

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

[127] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 813–825, 2017.

[128] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 3, pp. 389–402, 2009.

[129] H. Aydin, R. Melhem, and D. Mossé, "Tolerating faults while maximizing reward," in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000.* IEEE, 2000, pp. 219–226.

[130] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "Software-implemented hardware error detection: Costs and gains," in *Dependability (DEPEND), 2010 Third International Conference on.* IEEE, 2010, pp. 51–57.

[131] J. Huang, K. Huang, A. Raabe, C. Buckl, and A. Knoll, "Towards fault-tolerant embedded systems with imperfect fault detection," in *Proceedings of the 49th Annual Design Automation Conference.* ACM, 2012, pp. 188–196.

[132] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerant scheduling on a hard real-time multiprocessor system," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International.* IEEE, 1994, pp. 775–782.

[133] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272–284, 1997.

[134] Y. Guo, D. Zhu, H. Aydin, J.-J. Han, and L. T. Yang, "Exploiting primary/backup mechanism for energy efficiency in dependable real-time systems," *Journal of Systems Architecture*, vol. 78, pp. 68–80, 2017.

[135] A. Acharya and B. Badrinath, "Checkpointing distributed applications on mobile computers," in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on.* IEEE, 1994, pp. 73–80.

[136] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[137] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[138] H. Higaki and M. Takizawa, "Checkpoint-recovery protocol for reliable mobile systems," in *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on.* IEEE, 1998, pp. 93–99.

[139] K. G. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal checkpointing of real-time tasks," *IEEE Transactions on computers*, vol. 100, no. 11, pp. 1328–1341, 1987.

[140] E. Gelenbe, "On the optimum checkpoint interval," *Journal of the ACM (JACM)*, vol. 26, no. 2, pp. 259–270, 1979.

[141] R. Melhem, D. Mossé, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 217–231, 2004.

[142] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *Services (SERVICES), 2011 IEEE World Congress on*. IEEE, 2011, pp. 280–287.

[143] N. LEVESON, "Software fault tolerance-the case for forward recovery," in *4th Computers in Aerospace Conference*, 1983, p. 2327.

[144] S. Jajodia, C. D. McCollum, and P. Ammann, "Trusted recovery," *Communications of the ACM*, vol. 42, no. 7, pp. 71–75, 1999.

[145] J. Long, W. K. Fuchs, and J. A. Abraham, "Forward recovery using checkpointing in parallel systems." in *ICPP (1)*, 1990, pp. 272–275.

[146] W. Torres-Pomales, "Software fault tolerance: A tutorial," 2000.

[147] N. G. Leveson, "Software safety: Why, what, and how," *ACM Computing Surveys (CSUR)*, vol. 18, no. 2, pp. 125–163, 1986.

[148] A. Avizienis, "The methodology of n-version programming," *Software fault tolerance*, vol. 3, pp. 23–46, 1995.

[149] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[150] M. R. Lyu and Y.-T. He, "Improving the n-version programming process through the evolution of a design paradigm," *IEEE Transactions on Reliability*, vol. 42, no. 2, pp. 179–189, 1993.

[151] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an n-version software experiment," *IEEE Transactions on software engineering*, vol. 16, no. 2, pp. 238–247, 1990.

[152] J. C. Knight and N. Leveson, "A large scale experiment in n-version programming," in *NASA. Goddard Space Flight Center Proceedings of the Ninth Annual Software Engineering Workshop p 86-99(SEE N 86-19967 10-61)*, 1984.

[153] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance*, vol. 3, pp. 1–22, 1995.

[154] M. R. Lyu *et al.*, "Handbook of software reliability engineering," 1996.

[155] K. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE transactions on Computers*, vol. 38, no. 5, pp. 626–636, 1989.

[156] H. D. Patterson and R. Thompson, "Recovery of inter-block information when block sizes are unequal," *Biometrika*, vol. 58, no. 3, pp. 545–554, 1971.

[157] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems," in *Design and Test Workshop (IDT), 2011 IEEE 6th International*. IEEE, 2011, pp. 12–17.

[158] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant network-on-chip multiprocessors," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*. ACM, 2011, pp. 129–136.

[159] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson, "Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 121–130.

[160] J. H. Collet, P. Zajac, M. Psarakis, and D. Gizopoulos, "Chip self-organization and fault tolerance in massively defective multicore arrays," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 2, pp. 207–217, 2011.

[161] Y. Guo, D. Zhu, and H. Aydin, "Reliability-aware power management for parallel real-time applications with precedence constraints," in *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE, 2011, pp. 1–8.

[162] Y. Guo, D. Zhu, and H. Aydin, "Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2013, pp. 62–71.

[163] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 786–796.

[164] Y. Guo, H. Su, D. Zhu, and H. Aydin, "Preference-oriented real-time scheduling and its application in fault-tolerant systems," *Journal of Systems Architecture*, vol. 61, no. 2, pp. 127–139, 2015.

[165] Q. Han, T. Wang, and G. Quan, "Enhanced fault-tolerant fixed-priority scheduling of hard real-time tasks on multi-core platforms," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*. IEEE, 2015, pp. 21–30.

[166] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 564–571.

[167] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, "Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*. IEEE, 2010, pp. 434–441.

[168] C. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 48, 2014.

[169] R. Al-Omari, A. K. Somani, and G. Manimaran, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*. IEEE, 2001, pp. 8–pp.

[170] R. Al-Omari, A. K. Somani, and G. Manimaran, "Efficient overloading techniques for primary-backup scheduling in real-time systems," *Journal of Parallel and Distributed Computing*, vol. 64, no. 5, pp. 629–648, 2004.

[171] D. Zhu, R. Melhem, D. Mossé, and E. Elnozahy, "Analysis of an energy efficient optimistic tmr scheme," in *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*. IEEE, 2004, pp. 559–568.

[172] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles, "Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 233–238.

[173] D. Zhu, X. Qi, and H. Aydin, "Priority-monotonic energy management for real-time systems with reliability requirements," in *Computer Design, 2007. ICCD 2007. 25th International Conference on*. IEEE, 2007, pp. 629–635.

[174] D. Zhu and H. Aydin, "Reliability-aware energy management for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1382–1397, 2009.

[175] D. Zhu, H. Aydin, and J.-J. Chen, "Optimistic reliability aware energy management for real-time tasks with probabilistic execution times," in *Real-Time Systems Symposium, 2008*. IEEE, 2008, pp. 313–322.

[176] B. Zhao, H. Aydin, and D. Zhu, "Enhanced reliability-aware power management through shared recovery technique," in *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*. IEEE, 2009, pp. 63–70.

[177] B. Zhao, H. Aydin, and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 381–386.

[178] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012, pp. 285–294.

[179] O. S. Unsal, I. Koren, and C. M. Krishna, "Towards energy-aware software-based fault tolerance in real-time systems," in *Low Power Electronics and Design, 2002. ISLPED'02. Proceedings of the 2002 International Symposium on*. IEEE, 2002, pp. 124–129.

[180] S. Aminzadeh and A. Ejlali, "A comparative study of system-level energy management methods for fault-tolerant hard real-time systems," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1288–1299, 2011.

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

[182] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms," in *2013 International Green Computing Conference Proceedings*. IEEE, 2013, pp. 1–11.

[183] S. Gopalakrishnan and M. Caccamo, "Task partitioning with replication upon heterogeneous multiprocessor systems," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*. IEEE, 2006, pp. 199–207.

[184] W. Luo, F. Yang, L. Pang, and X. Qin, "Fault-tolerant scheduling based on periodic tasks for heterogeneous systems," in *International Conference on Autonomic and Trusted Computing*. Springer, 2006, pp. 571–580.

[185] X. Zhu, X. Qin, and M. Qiu, "Qos-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters," *IEEE transactions on Computers*, vol. 60, no. 6, pp. 800–812, 2011.

[186] H. Xu, R. Li, C. Pan, and K. Li, "Minimizing energy consumption with reliability goal on heterogeneous embedded systems," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 44–57, 2019.

[187] R. Devaraj, "A solution to drawbacks in capturing execution requirements on heterogeneous platforms," *The Journal of Supercomputing*, pp. 1–16, 2020.

[188] T. Li, T. Zhang, G. Yu, J. Song, and J. Fan, "Minimizing temperature and energy of real-time applications with precedence constraints on heterogeneous mpsoc systems," *Journal of Systems Architecture*, vol. 98, pp. 79–91, 2019.

[189] J. Zhou, M. Zhang, J. Sun, T. Wang, X. Zhou, and S. Hu, "Drheft: Deadline-constrained reliability-aware heft algorithm for real-time heterogeneous mpsoc systems," *IEEE Transactions on Reliability*, pp. 1–12, 2020.

[190] Y. Liu, J. Liu, Z. Zhu, C. Deng, Z. Ren, and X. Xu, "Adaptive fault-tolerant scheduling in heterogeneous real-time systems," in *2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2019, pp. 982–987.

[191] S. Bansal, R. K. Bansal, and K. Arora, "Energy-cognizant scheduling for preference-oriented fixed-priority real-time tasks," *Journal of Systems Architecture*, vol. 108, p. Article No. 101743, 2020.

[192] R. Begam, Q. Xia, D. Zhu, and H. Aydin, "Preference-oriented fixed-priority scheduling for periodic real-time tasks," *Journal of Systems Architecture*, vol. 69, pp. 1–14, 2016.

[193] M. Zhao, D. Liu, X. Jiang, W. Liu, G. Xue, C. Xie, Y. Yang, and Z. Guo, "CASS: Criticality-aware standby-sparing for real-time systems," *Journal of Systems Architecture*, vol. 100, p. Article No. 101661, 2019.

[194] S. Safari, S. Hessabi, and G. Ershadi, "LESS-MICS: A low energy standby-sparing scheme for mixed-criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–10, 2020.

[195] N. Kumar, J. Mayank, and A. Mondal, "Reliability aware energy optimized scheduling of non-preemptive periodic real-time tasks on heterogeneous multiprocessor system," *IEEE Transactions on Parallel and Distributed Systems*, pp. 871–885, 2019.

[196] P. Martí, J. M. Fuertes, G. Fohler, and K. Ramamritham, "Improving quality-of-control using flexible timing constraints: metric and scheduling," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE.* IEEE, 2002, pp. 91–100.

[197] P. Martí, J. Yépez, M. Velasco, R. Villà, and J. M. Fuertes, "Managing quality-of-control in network-based control systems by controller and message scheduling co-design," *IEEE transactions on Industrial Electronics*, vol. 51, no. 6, pp. 1159–1167, 2004.

[198] G. Buttazzo, M. Velasco, and P. Marti, "Quality-of-control management in overloaded real-time systems," *IEEE Transactions on Computers*, vol. 56, no. 2, pp. 253–266, 2007.

[199] G. Buttazzo, M. Velasco, P. Marti, and G. Fohler, "Managing quality-of-control performance under overload conditions," in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on.* IEEE, 2004, pp. 53–60.

[200] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On task schedulability in real-time control systems," in *Real-Time Systems Symposium, 1996., 17th IEEE.* IEEE, 1996, pp. 13–21.

[201] E. Bini and M. D. Natale, "Optimal task rate selection in fixed priority systems," in *Real-Time Systems Symposium. 26th International.* IEEE, 2005.

[202] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *Computers, IEEE Transactions on*, vol. 51, no. 3, pp. 289–302, 2002.

[203] M. Marinoni and G. Buttazzo, "Elastic dvs management in processors with discrete voltage/frequency modes," *Industrial Informatics, IEEE Transactions on*, vol. 3, no. 1, pp. 51–62, 2007.

[204] T. Chantem, X. Wang, M. D. Lemmon, and X. S. Hu, "Period and deadline selection for schedulability in real-time systems," in *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on.* IEEE, 2008, pp. 168–177.

[205] T. Chantem, X. S. Hu, and M. D. Lemmon, "Generalized elastic scheduling for real-time tasks," *Computers, IEEE Transactions on*, vol. 58, no. 4, pp. 480–495, 2009.

[206] Y. Wu, G. Buttazzo, E. Bini, and A. Cervin, "Parameter selection for real-time controllers in resource-constrained systems," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 4, pp. 610–620, 2010.

[207] D. Fontantelli, L. Palopoli, and L. Greco, "Optimal cpu allocation to a set of control tasks with soft real–time execution constraints," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*. ACM, 2013, pp. 233–242.

[208] Y.-C. Tian and L. Gui, "Qoc elastic scheduling for real-time control systems," *Real-Time Systems*, vol. 47, no. 6, pp. 534–561, 2011.

[209] J. Kim, K. Lakshmanan, and R. R. Rajkumar, "Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems," in *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. IEEE Computer Society, 2012.

[210] G. C. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive tasks: Model, analysis, and design issues," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[211] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. IEEE, 1997, pp. 298–307.

[212] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[213] B. R. Borgerson and R. F. Freitas, "A reliability model for gracefully degrading and standby-sparing systems," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 517–525, 1975.

[214] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings of the Int. WS on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.

[215] S. K. Baruah, "A general model for recurring real-time tasks," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE, 1998, pp. 114–122.

[216] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 302–315, 2011.

[217] A. Bhuiyan, Z. Guo, A. Saifullah, N. Guan, and H. Xiong, "Energy-efficient real-time scheduling of DAG tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 5, pp. 1–25, 2018.

[218] K. D. Cooper, P. J. Schielke, and D. Subramanian, "An experimental evaluation of list scheduling," *TR98, 326*, 1998.

[219] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Trans. on software engineering, 15(12)*, 1989.

[220] Y. Liu, B. Veeravalli, and S. Viswanathan, "Novel critical-path based low-energy scheduling algorithms for heterogeneous multiprocessor real-time embedded systems," in *2007 International Conference on Parallel and Distributed Systems*. IEEE, 2007, pp. 1–8.

[221] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE'98)*. IEEE, 1998, pp. 97–101.

[222] R. Davis and A. Wellings, "Dual priority scheduling," in *Proceedings 16th IEEE Real-Time Systems Symposium*. IEEE, 1995, pp. 100–109.

[223] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[224] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[225] N. C. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[226] N. C. Audsley, *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[227] B. Lincoln and A. Cervin, "Jitterbug: A tool for analysis of real-time control performance," in *Proceedings of the 41st Conference on Decision and Control*. IEEE, 2002.

[228] S. M. Melzer and B. C. Kuo, "Sampling period sensitivity of the optimal sampled data linear regulator," *Automatica*, vol. 7, no. 3, pp. 367–370, 1971.

[229] E. Bini and A. Cervin, "Delay-aware period assignment in control systems," in *Real-Time Systems Symposium*. IEEE, 2008.

[230] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[231] H. Aydin, R. Melhem, D. Mosse, and P. Mejía-Alvarez, "Optimal reward-based scheduling for periodic real-time tasks," *Computers, IEEE Transactions on*, vol. 50, no. 2, pp. 111–130, 2001.

Curriculum Vitae

Abhishek Roy received his Bachelor of Science degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh in 2009. He was employed as a software engineer for several years. He received his Master degree in Computer Science from George Mason University in 2018. His research interests focus on low-power computing, fault tolerance and real-time embedded systems.