Towards Energy-Efficient Real-Time Computing in Embedded Systems

Ashraf Suyyagh

Department of Electrical and Computer Engineering
Faculty of Engineering, McGill University
Montreal, Canada

October 2018

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dedication

For my beloved parents Emad and Lina,
For my sisters Diana and Nancy,

In memory of my closest friend and mentor Brian Brice,
September 25th, 1946 - August 7th, 2018

Acknowledgments

"And not in utter loneliness to live,
Myself at last did to the Devil give!"
Goethe, Faust Part II

Throughout the time of doing my graduate studies, and as years went by, I came to think of myself as *Faust*. With no clear path to the end, it has always felt like I have sold my soul to the devil and the time to recollect is ticking. Albeit, this contract is real, and the consequences are dire. Despite this anxious feeling, I am blessed to have known remarkable people who shared this very long journey with me, personally and professionally. These people either directly or indirectly made the journey at times tolerable, at others enjoyable. I would like to take the chance to thank them.

First and foremost, I dearly thank my supervisor Prof. Zeljko Zilic, for his continued support, standing by my side, and whenever I doubted myself, he never stopped believing in me. I am greatly indebted to him and thankful for allowing me to learn and share the knowledge of modern embedded systems and ARM technologies. I would also like to thank the University of Jordan for their financial help in my first three years of my program, without which, I would not have started my PhD. I also extend my thanks to my professors and colleagues at the Department of Computer Engineering at the University of Jordan for their support and patience. I would like to thank Prof. Juan Carlos Sáez Alcaide for his generous help and time in helping me use the PMCTrack tool in my research. A multitude of thanks goes to the legions of unsung contributors over all online forums who lent their time and knowledge to help me tackle technical issues related to Linux and *bash* scripting.

To the members of the Integrated Microsystems Laboratory (in no particular order) I want to thank, Ben Nahill, Steve Ding, and Andrey Tolstikhin with whom I had insightful and inspirational research discussions that enriched my technical knowledge. To Jason Tong, Bojan Mihajlović, Omid Sarbishei, Majid Janidarmian, Atena Roshan Fekr, Dimitrios Stamoulis, Ari Ramdial, Amir Shahshahani and his wife Sadaf, and Omar Abdelfattah for their unwavering friendship, continued support, and for making my stay in the lab fun and enjoyable. To Chuansheng Dong, George Gal, and Atousa Assadi for the good times we had together. I would also like to acknowledge Alexandre Courtemanche, Harsh Aurora and Loren Peter Lugosch whom I had the pleasure of teaching the microprocessor systems course with. To my current lab colleagues, Anastasios Alexandridis, Junchao Wang, Pavel Sinha, and Farimah Poursafaei, though our time together was short, I extend my best wishes for your success. Warm thanks go to Aileen Roungou for her friendship and help in editing my French abstract translation.

On a personal note, a special thanks goes to Prof. Andraws Sweidan who has always been like a father figure to me, and his family whose kindness will forever be cherished. To my uncle Radwan to whom I am indebted for helping me overcome initial obstacles. To my friend Osaid Matar, his wife Heba Alqub and their son Laith, the good times we have had shall be remembered fondly. To Tammy Hurst for taking her time to introduce me to Montreal, and the opportunities to explore the city and get out of my comfort zone. To Audrey-Ann Raymond, your presence in my life enriched me and helped me grow personally.

To my friends in Jordan and over the world, Enas Jaara, Adiy Tweissi, Fadi Ghawanmeh, Ahmad Souri, Diana Nassar, Alia Al-Bashir, Hamza Nassar, and Imen Ben Khalid who tried to the best of their abilities to stay in touch despite the distance, their busy schedule, and time difference.

To the memory of Brian Brice, my mentor, and my closest friend. I shall ever be thankful to the daily coffee hour during which we discussed our many shared interests. Your wisdom, unique perspective on various issues, and advice have had a major influence on me. To the countless hours we spent discussing Shakespeare, English history, Greek tragedy, the operas of Wagner and Verdi and the works of Gilbert and Sullivan, Russian literature and world politics. The last months of your worldly existence brought both joyful and painful moments alike. May your soul rest in peace. To the friends of Brian and members of his Carmina Choir, especially Danielle Pullen and her husband Jack Denison, Dr. Gillian Bramwell, Van Lam, and Catherine Pycock, thanks for your support after Brian's passing.

Last but not least, I want to thank my parents and sisters for their unconditional love, support, and encouragement. I can never find the right words to give them their due for the sacrifices they have made and the stress and pain my path have caused them.

Abstract

Modern embedded real-time systems are increasingly interconnected with a multitude of sensory devices, other embedded systems, and the cloud. The adoption of high-end embedded single core processors and multiprocessors emanates from complex application processing requirements. Timeliness, safety, and deterministic systems are long-standing design and operational requirements of embedded systems. Recently, mobility, energy-efficiency, and heat dissipation are equally crucial design requirements in applications including autonomous mobile robots, wearable devices, and sensor networks. High-end embedded processors employ energy-reduction techniques like Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM). An effective energy-management strategy simultaneously exploits hardware- and software-level energy-reduction techniques.

Initially, this thesis addresses the issue of energy-reduction on DVFS-capable single core systems with peripheral devices. We consider a system-wide minimization problem where we concurrently consider DVFS and DPM. Given that the frequency to task assignment is an NP-hard problem, we appropriate and adapt two metaheuristics in our approach to frequency assignment; namely the differential evolution and genetic algorithms. We analyze the performance of the metaheuristics given various initial conditions and show in our simulations that our approach yields better results than two well-known heuristics.

Further, even though discrete-time simulators are sufficient for analyzing real-time schedule feasibility, they fall short when evaluating energy-efficient scheduling. This is due to incorrect processor modeling (i.e. due to IP rights, complex processor designs) and the inability to capture realistic task behavior. The literature often presents case studies on real hardware to corroborate simulations. However, these approaches are often ambiguous. We propose a methodology that facilitates evaluating real-time systems on real hardware using available embedded benchmarks as system tasks at various system load points. Similar to software simulations, our methodology tackles the issue of examining the system at different utilization points. We build on previous work that estimates task WCET, generates task periods, and assigns task utilizations. The three parameters are interlocked, which limits the flexibility of changing one without affecting the others. We propose a set of efficient algorithms that pair tasks with bounded or discrete periods to meet the total system utilization with minimal relative errors.

Finally, we address the issue of energy-efficient scheduling on clustered heterogeneous platforms. We focus on energy-efficient partitioning where task allocation to heterogeneous clusters directly impacts the total system energy. In this thesis, we couple the problem of energy-efficient partitioning on single-ISA heterogeneous platforms with task-aware scheduling. Tasks differ in their instruction mix, cache behavior, memory and I/O access, execution path, and active processing and SoC circuitry. This affects their power demand. We make further use of underlying frequency scaling hardware and sleep states to minimize the system energy. We propose two variants of our *Task and Cluster Heterogeneity Aware Partitioning* (TCHAP) algorithm targeting ARM big.LITTLE platforms. Based on our methodology for simulation on real hardware, we show that our algorithms achieve between 13% to 23% energy-reduction on average compared to a state-of-the-art schemes.

Abrégé

Les systèmes temps-réel embarqués modernes sont de plus en plus interconnectés avec une multitude d'appareils sensoriels, d'autres systèmes embarqués et du “cloud”. L'adoption de processeurs monocœurs et multicœurs haut de gamme dans les applications embarquées émane des exigences de traitement d'applications complexes. La pérennité, la sécurité et les systèmes déterministes sont des obligations de conception et de fonctionnement de longue date des systèmes embarqués. Récemment, la mobilité, l'efficacité énergétique et la dissipation de chaleur sont des exigences de conception tout aussi cruciales dans les applications comprenant les robots mobiles autonomes, les appareils portables et les réseaux sensoriels. Les processeurs embarqués haut de gamme utilisent des techniques de réduction d'énergie telles que le voltage dynamique, la mise à l'échelle de la fréquence (DVFS) et la gestion dynamique de la puissance (DPM). Une stratégie efficace de gestion de l'énergie exploite simultanément les techniques de réduction de l'énergie au niveau matériel et logiciel.

Initialement, cette thèse aborde le problème de la réduction de la consommation d'énergie sur les systèmes monocœurs avec matériel DVFS et périphériques. Nous considérons un problème de minimisation à l'échelle du système où nous examinerons simultanément DVFS et DPM. Étant donné que l'affectation de fréquence à la tâche est un problème NP-difficile, nous avons adaptés deux métaheuristiques à notre approche d'affectation de fréquence: l'évolution différentielle et l'algorithme génétique. Nous analyserons les performances des métaheuristiques en fonction de diverses conditions initiales. Nous montrerons dans nos simulations que notre approche donne de meilleurs résultats que deux heuristiques bien connues.

En outre, bien que les simulateurs à temps-discret sont suffisants pour analyser la faisabilité des emplois du temps en temps réel, ils sont insuffisants pour évaluer des emplois du temps qui économisent l'énergie. Cela est dû à une modélisation incorrecte de processeurs (en raison des droits de propriétés intellectuelles, conceptions de processeurs complexes) et à l'incapacité de capturer un comportement réaliste des tâches. La littérature présente souvent des études de cas de “real-hardware” pour corroborer des simulations. Cependant, ces approches sont souvent ambiguës. Nous proposons une méthodologie qui facilite le portage de simulations sur la ”real-hardware” a l'aide des ”benchmarks” embarqués en tant que tâches système.

Comme pour les simulations logicielles, notre méthodologie aborde la question de l'examen du système à différents points d'utilisation. Nous nous appuyons sur des travaux antérieurs qui évaluent la WCET des tâches, génèrent des périodes des tâches et attribuent des utilisations des tâches.

Les trois paramètres sont interconnectés, ce qui limite la possibilité de modifier l'un sans affecter les autres. Nous proposons un ensemble d'algorithmes efficaces qui associent des tâches à des périodes délimitées pour répondre à l'utilisation totale du système avec un minimum d'erreurs relatives.

Enfin, nous abordons la question de l'ordonnancement optimal énergétique sur des plateformes hétérogènes groupées. Nous nous concentrons sur la répartition énergétique efficace des tâches à des groupes hétérogènes ayant un impact direct sur l'énergie totale du système. Dans cette thèse, nous associerons le problème de répartition énergétique efficace sur des plateformes hétérogènes mono-ISA à un ordonnancement conscient de la tâche. Les tâches diffèrent dans la combinaison d'instructions, le comportement du cache, l'accès à la mémoire et aux I/O, la route d'exécution, et le circuit actif de SoC. Cela affecte leur demande de puissance. Nous utilisons davantage les fréquences de “l'hardware” et les états de veille pour minimiser l'énergie du système. Nous proposons deux variantes de notre algorithme de répartition conscient de l'hétérogénéité de tâches et de cluster (TCHAP) ciblant les plateformes ARM big.LITTLE. Nous montrons que nos algorithmes permettent de réduire de 13% à 23% la consommation moyenne d'énergie par rapport à des systèmes à la pointe de la technologie.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BET	Break Even Time
BF	Best-Fit
BFD	Best-Fit-Decreasing
CBS	Constant Bandwidth Server
CHP	Clustered Heterogeneous Processors
CMOS	Complimentary Metal-Oxide Semiconductor
COTS	Commercial Off-The-Shelf
CMP	Chip Multiprocessors
CS	Critical Frequency
DPM	Dynamic Power Management
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
EVT	Extreme Value Theory
FF	First-Fit
FFD	First-Fit-Decreasing
HCN	Highly Composite Number
HP	Hyper-Period
ILP	Integer Linear Programming
IP	Intellectual Property
ISA	Instruction Set Architecture
LLC	Last Level Cache
MAS	Multi-Agent Systems
MBPTA	Measurement Based Probability Timing Analysis
MOET	Maximum Observed Execution Time
NF	Next-Fit

NFD	Next-Fit-Decreasing
PLL	Phased Locked Loops
PMC	Performance Monitoring Counter
PMU	Performance Monitoring Unit
PTS	Preemption Threshold Scheduling
pWCET	Probabilistic Worst Case Execution Time
RM	Rate Monotonic
RMSE	Root Mean Square Error
SCB	Single Board Computer
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
SSE	Sums of Square Errors
TDP	Thermal Dissipation Power
TLM	Transaction-Level Modeling
WCEC	Worst Case Execution Cycles
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
WF	Worst-Fit
WFD	Worst-Fit-Decreasing

List of Algorithms

CS-DVS	Critical Frequency - Dynamic Frequency Scaling
SAFS	Simulated Annealing Frequency Scaling
GAFS	Genetic Algorithm Frequency Scaling
DEFS	Differential Algorithm Frequency Scaling
CPA-AU	Compute Propagate Adjust - Ascending Utilization
CPA-RU	Compute Propagate Adjust - Random Utilization
CPA-DU	Compute Propagate Adjust - Descending Utilization
CPA-MUP	Compute Propagate Adjust - Maximize Unique Periods
AICF	Assign Initial Clusters Frequencies
T-CAFE	Task-Aware Cluster Assignment Frequencies Exploration
TCHAP_I	Task and Cluster Heterogeneity Aware Partitioning (optimize for Idle)
TCHAP_S	Task and Cluster Heterogeneity Aware Partitioning (optimize for Sleep)
HIT-LTF	Heterogeneous Island- and Task-Aware Largest Task First

Chapter 1

Introduction

Real-time embedded systems are pervasive in industrial, automotive, avionics, and consumer applications. Meeting timing constraints is a top priority in real-time embedded systems. System reliability, safety, determinism, and operational predictability dominate throughout all design stages and decisions. During the past couple of decades, there has been numerous breakthroughs and advancements in material design, lithography, transistor design and miniaturization. This enabled packing more cores and peripheral devices into system-on-chip (SoC) integrated circuits (ICs) reducing the amount of physical space required in designing circuit boards. The manufacturing of portable devices relies heavily on our ability to build devices in small form factors. As such, mobility, energy-efficiency, and heat dissipation have equally become crucial design considerations. For example, portable embedded systems impose limitations on the ability to use passive or active heat management approaches (e.g. heat-sinks or cooling fans). Furthermore, system mobility necessitates the use of smaller size power supplies and allows for the use of battery-powered systems or systems with limited power supply (e.g., solar powered). This is quite a common design practice in applications including autonomous mobile robots (e.g., drones), wearable devices, and sensor networks.

Autonomous robots and drones have seen wide adoption in search and rescue applications in war-torn or catastrophe-struck areas around the world as well as civil applications. Wearable and sensory devices take a primary lead in health-care applications. Given the central and essential role these systems play in wide array of life-saving applications, the longevity, reliability of the system, and continued uninterrupted operation is of vast importance. This is especially true when it is difficult to access the system or replace the power source due to operational and environmental constraints. Higher energy consumption increases the overall system heat and the occurrence of thermal hot-spots. This adversely affects the system reliability and decreases battery life. For example, system failure rates can rise by a factor of two with each 15 degree Celsius rise in temperature [5]. Consequently, safe operation in critical embedded systems benefits from lower total system energy-consumption and further elicits the need for developing fault-tolerant, energy- (and thermally) aware schedulers that meet the system timing constraints. Despite the low energy of embedded systems in general, their proliferation and expansive use in billions of devices would collectively add up to a high energy footprint . This holds for either mobile embedded systems or those connected to the power grid. As a result, power management in embedded and real-time systems enables green computing and indirectly reduces carbon emissions and potentially has a positive environmental impact.

General-purpose processors, graphic units, configurable FPGA units or custom ASICs have a power budget that ranges between dozens of watts up to hundreds of Watts. This renders them effectively unsuitable for deployment in real-time embedded systems where portability and long system operational life is essential. Due to the stringent power constraints of modern embedded systems, specialized processors have been developed to meet functional, operational, timing, and power requirements. Low power embedded processors have a typical power budget of 1 - 2 Watts (e.g. TI AM437x Sitara Cortex-A9 processors [6]). On another hand, ultra-low power embedded processors have power ratings in the range of milliwatts (e.g. the STM32L5 series have a rating of 60μA/MHz in active mode running at 3.3V (typical 60mW) [7]). For high-performance multi-core embedded processors such as Samsung's Exynos 5422 ARM's big.LITTLE in Odroid-XU3 single board computers (SBC), they boast a power rating of merely 5 Watts. Figure 1.1 presents the Odroid-XU3 board which we use in this thesis. Such low-power embedded processors simplify heat dissipation management, and by extension maintain system reliability.

To improve the energy-efficiency of processors, dynamic power management techniques were among the first to make use of multiple sleep modes incorporated in processor designs. Sleep modes turn off inactive processor components and reduce static power; that is, power related to leakage current whenever logic components are active. As performance demands increased, both architectural innovations and a steady increase of processor frequency constituted a trend that dominated the market. This trend came at the cost of increased power consumption and power density. Dynamic power which is directly proportional to processors run-time voltage and frequency overcame static power in the total share of processor power. To mitigate this, frequency and voltage controller blocks were added to allow for run-time alteration of processor frequency. In general purpose computing, Intel and AMD introduced Intel SpeedStep and PowerNow! (later known as Optimized Power Management)

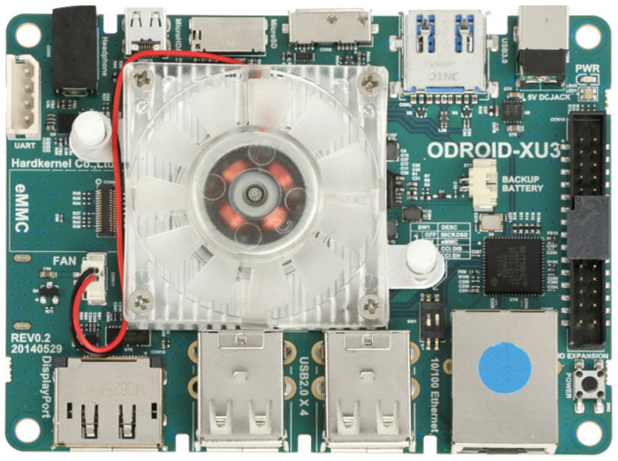


Figure 1.1 The Odroid-XU3 board featuring Samsung's Exynos 5422 ARM big.LITTLE processor

technologies, respectively. Both technologies allow for direct control of the system operational frequency by software. Each processor has a set of discrete set of frequencies and a firmware interface that allows for the selection of appropriate frequencies to balance between application performance demands and system energy-reduction. Hardware controllers also throttle run-time frequencies when overloaded cores reach high thermal levels at the risk of inability to dissipate heat efficiently.

The steady increase of processor frequencies led to processor designs reaching the speed wall due to overheating issues. As the power density increased, it has become difficult to dissipate the processor heat by conventional means. Instead, to sustain the demand for higher performance power, designs shifted towards packing multiple cores on the same chip. With continuous advancement in lithography, transistors designs, and transistor scaling, successive processor generations boasted an increase in core count over time. However, this new design trend tilted the share of total processor power consumption towards leakage current. This is due to the increased number of transistor count in many core designs, as well scaling down maximum processor frequency. As a result, dynamic power management techniques (i.e. sleep modes) had a surge of renewed interest. However, this time, power reduction techniques make concurrent use of both voltage and frequency scaling and sleep modes.

Within the context of real-time systems and real-time scheduling, the appropriation of sleep modes and frequency scaling is not as straightforward as in general purpose applications. Real-time workloads must never violate their deadlines. Coarse-grained (processor level) and fine-grained (task level) scaling down of processor frequency extends the task(s) execution times with higher probability of invalidating system feasibility. Furthermore, even though the prolonged task execution time decreases the dynamic energy consumption, it jointly increases the task leakage power. Homogeneous and heterogeneous multiprocessors and multicores add extra optimization dimensions to an already complex problem: task partitioning and the number of active cores. Energy-aware scheduling literature adapts to the advancements in the underlying hardware and offers new solutions to these challenging problems.

1.1 Problem Definition

Single core processors have for long dominated the embedded market. In 2016 and 2017, one market study [8] shows that single core microprocessors make for around 57% of all embedded projects. For example, over 200 companies have licensed the Cortex-M family of single core processors to date and shipped over 12.1 billion units [9]. Whereas low-power single core microcontrollers highly depend on sleep modes and lack the means to reduce dynamic energy at run-time, high-end single core embedded processors provide system designers with the necessary hardware and OS interfaces for tight control over the power budget. Furthermore, embedded systems are evolving into cyber-physical systems; that is, highly interconnected, tightly coupled systems with the physical world. The consolidation of the physical world into the interconnected embedded processing world requires the use of many SoC peripherals and external devices. Cyber-physical systems heavily rely on sensors, communication devices (e.g. bluetooth, Wi-Fi), and recently the cloud. These peripherals and devices claim significant portion of the system power profile and their share can no longer be excluded in energy minimization. As a result, system-wide power reduction becomes a significant challenge.

Within this context, few outstanding problems are apparent. In these systems, both leakage current dissipation and dynamic power energy consumption constitute almost equally high share of the total power consumption. It is not always clear how to effectively consolidate and couple the two main energy reduction approaches for maximum reduction of total system energy consumption. There is an intricate interplay between both techniques. Extensive use of dynamic frequency scaling reduces the available time of potential idle intervals which can be necessary to offset the increase of static power due to prolonged task execution. Conversely, extensive use of sleep modes imply that tasks will be running at higher frequencies where the effects of dynamic energy consumption can outweigh the benefits of deactivating the processing circuitry. Furthermore, the wake-up overhead due to “data destructive” sleep modes trigger successive operations in the underlying cache, bus, and memory subsystems in order to restore the lost system state. This induces further energy consumption. Additionally, in interconnected embedded systems, many tasks heavily rely on SoC peripherals and external devices to collect and store data, interact with the external word, or visualize system state to provide meaningful user data. In many systems, the energy profile of these devices is non-negligible. The literature is based on device models where the associated devices of system tasks are expected to be ready and available at all times the associated task is executing. Therefore, the use of task frequency scaling will lead to an increase of the power consumption of the device set. In this context, we can define the first problem as:

- How can we assign frequency scales to individual system tasks to attempt to minimize the overall system consumption taking into consideration the associated system devices? And how can we effectively integrate DPM/DFVS to achieve better energy reductions given the intricate interplay between the two techniques?

Secondly, a major issue in energy-aware scheduling problems is the efficacy of evaluation tools. These tools include in-house time-driven simulators, advanced community-supported open-source or proprietary simulators, and actual hardware. We present an abstract usage flow of these tools in Figure 1.2. Literature actively relies on simulation tools or mathematical formulations to evaluate real-time scheduling feasibility. These tools are developed by typical programming languages with the aid of extended classes and macros that enable event-driven simulation (e.g. SystemC). Such simulators are sufficient for capturing the temporal behaviour of system tasks and scheduling anomalies. However, these tools are limited in their capacity to faithfully model the energy-consumption of system tasks. Another popular approach in academia relies on far more sophisticated event driven or cycle-accurate simulators that present abstract functional and descriptive blocks of the complex hardware. Many also run operating system images and allow to run user tasks in emulation mode. One such popular tool is GEM5 [10]. GEM5 can also be interfaced with McPAT [11], which is another popular tool for power, area, and timing estimation. Despite the popularity of these tools and wide community support, the hardware models they employ remain not rigorously tested or validated against the hardware they represent. Recent studies show that these tools might be harmful to the research community and lead to results and conclusions which are based on complex models that most users do not understand fully [12, 13]. Many of these models are erroneous and inconsistent. For example, two models developed for ARM big.LITTLE had a timing discrepancy of -51% and 10% when validated against the hardware [12]. In another study, McPAT power estimates deviated considerably from actual power consumption due to abstraction error, modeling assumption error, input error, and coding error with the majority of errors attributed the first two sources [14]. Abstraction errors are due to only modelling common components across various supported architectures and disregarding the rest. Modeling assumption error is due to the inherent inability to faithfully model all various micro-architectural components in detail. Input errors are due to the vagueness of configuring architectural parameters whereas coding errors are attributed to human mistakes when coding the models.

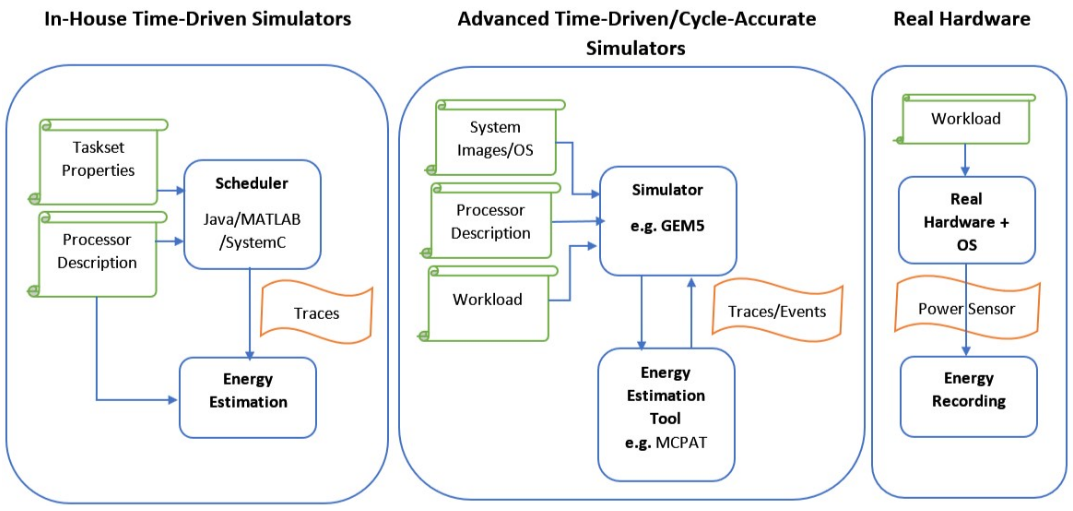


Figure 1.2 Evaluation platforms for embedded systems

As a result, direct evaluation on hardware remains the ideal option. This is quite common in general purpose computing or non-real time embedded systems research. However, direct evaluation on hardware is limited in real-time research to few case studies at certain points of the design and evaluation space. This is due to the arising challenges in constructing real tasksets for evaluation on hardware, especially when full-fledged evaluation at various system load points is desired.

The available taskset generation tools use timed loops, matrix operations, or employ functional code blocks to produce tasks that run up to a desired task worst-case execution time. This approach has the downside that the generated taskset might not be representative of real-life tasks. Additionally, given the reliance on code blocks and loops as building blocks, the generated tasks risk being repetitive in functionality and behaviour. Most importantly, these approaches mostly target single-core processors due to their simpler timing analysis. These tools strictly dictate that they do not work for multicore platforms [15].

Another alternative in generating tasksets is to rely on embedded benchmarks suites. These benchmarks arguably reflect some functionality of tasks found in industrial and commercial applications. Due to intellectual property constraints, the number of available benchmark suites is limited. Additionally, some researchers targeting specific platforms can infuse their own custom-designed in-house codes that take advantage of chipset features and levy up some embedded task functionality and realistic behaviour. In this case, we need to rely on the burdensome task of estimating task WCETs. Moreover, many use discrete bounded period sets either to limit system simulation time or to expose certain favourable characteristics (e.g. RM scheduling using harmonic periods reaches 100% utilization instead of less than 70% for non-harmonic periods [16]).

To reach solid conclusions about the algorithms under test, one needs to carry a comprehensive system evaluation across many points in the system design space. In real-time systems, one such evaluation criteria is the total system utilization (i.e. percentage of time the tasks occupy the processing elements). Real-time research tests the system under various load points ranging from low, to medium, to high utilization as system behaviour is influenced by the current task load.

The challenge is once we use discrete bounded or unbounded period sets, and tasks with estimated WCETs, it becomes apparent that the pairing problem between task WCETs, periods, and individual task utilization to meet total utilization becomes difficult. This is due to the fact that the three variables are interlocked. In real-time literature, when limited case studies on real hardware present results to corroborate the theoretical simulation results, in most cases, the methodology carried out for evaluation on hardware is ambiguous and restricted to a demonstrative case-study at an arbitrarily chosen load point. To the best of our knowledge, a methodology for the evaluation of embedded tasks on real hardware is missing. This thesis addresses the following questions pertaining to this issue:

- For any target single-core or multicore platform with either time-deterministic or time-randomized hardware features, what methodology should one follow in order to construct real-time tasksets that can be used to evaluate the system at different task load points?

- For a set of embedded tasks with estimated WCETs, and given a set of task utilizations that add up to the desired system load, how to find pairs of WCETs and periods from a discrete set to satisfy the task utilizations? In other words, how to pair these utilizations with any WCET such that the computed period is as close as possible to one of the periods in the discrete set?

- How can we generate taskset parameters pairings that respect the desired total system load?

Finally, embedded systems also harness the power of multiprocessor computing in demanding embedded applications. Homogeneous and heterogeneous platforms are ubiquitous. In 2015, they represented 30% of market share of embedded projects [8]. In these platforms, the energy-aware optimization problem takes extra optimization dimensions. In addition to the conventional problems of single core energy reduction, new challenges present themselves.

The first challenge is task-to-core affinity assignment. Different partitioning schemes affect both feasibility and energy consumption. The allocation of different tasks into different cores results in different partitioned schedules, task interactions, and thus different idle intervals. This directly affects the length of time a processor/core remains in a low-power sleep mode, or the available time that can be exploited for further down-scaling of processor/core frequency. The second challenge is the number of active cores. The system designer has multiple design choices. A possible approach is to run tasks on fewer cores at high frequencies and shutdown down inactive cores. This approach favors the reduction of the prominent effects of leakage current at the cost of higher dynamic energy consumption on fewer active cores. The second approach exploits all cores and extends task execution times as long as the schedule remains feasible. This approach favors reducing dynamic power consumption but increases the share of static power. There is no easy answer towards this problem. It highly depends on various factors. These factors include the partitioned scheduling policy under use, the taskset characteristics and interactions, the processor power profile, and shared resources.

When heterogeneous cores are considered, the power and performance profiles of the different cores or processors present another optimization parameter. In one scenario, heavy duty tasks can run on high-end processors at high frequencies resulting in bursts of execution with possibly ample idle time. This benefits in switching power-hungry processors into sleep modes for longer duration. Another scenario assigns heavy duty tasks onto energy-efficient cores. This scenario assumes that since the tasks utilize higher share than other system tasks, it is better to run them on energy-efficient processors. Frequency to task assignment further exacerbates the partitioning issue in heterogeneous systems. Performance-oriented cores running at lower speeds could potentially be more energy-efficient than low-end cores running at high frequencies. This class of scheduling problems is already known to be NP-Hard [17]. While real-time heterogeneous systems can use processing nodes of different architectures (e.g., different ISAs, GPUs, ASIC, and FPGAs), in embedded real-time computing, they mostly rely on single-ISA clustered heterogeneity in SoC form (e.g., ARM big.LITTLE SoCs). For embedded heterogeneous platforms, this thesis tries to answer these problems:

- In a single-ISA clustered heterogeneous multicore system, how to partition system tasks such that the overall processor power is reduced and the system remains feasible?
- How to couple task-awareness with cluster frequency selection and energy-efficient task partitioning?

1.2 Thesis Contributions and Collaborations

This section presents a summary of the contributions that we made in this research. We organize these contributions by the presented topics within this thesis. Most importantly, we highlight how we addressed the research questions and challenges that we presented in the previous section. We highlight the body of work that we have either previously published, or is at various points in the process of being considered for future publication.

- For the problem of frequency assignment of tasks with an associated device set, we propose utilizing two evolutionary-based algorithms based on the genetic algorithm *GAFS* and differential evolution *DEFS*. Given that the DVFS levels are discrete in all capable real-life processors, we implement the genetic algorithm and differential evolution algorithm as a discrete optimization problem to assign task frequencies. In lieu with other works, we have developed an in-house discrete time simulator based on transaction-level modelling in SystemC and C++. Our simulator includes modules for task generation and device association, the genetic and differential evolution algorithms, a scheduling module with feasibility checks, and energy reduction and computation. The simulator is able to compute the energy-consumption of any feasible taskset schedule and feeds it to the metaheuristic algorithms to use in the optimization problem. This work has resulted in the following publications:
    - **A. Suyyagh**, J. G. Tong and Z. Zilic, "Analysis of meta-heuristics performance in energy aware scheduling of real-time embedded systems," 2015 IEEE Conference on Applied Electrical Engineering and Computing Technologies (AEECT), Amman, 2015, pp. 1-6.
    - **A. Suyyagh**, J. G. Tong and Z. Zilic, Performance Evaluation of Meta-heuristics in Energy-Aware Real-Time Scheduling Problems," Jordanian Journal of Computers and Information Technology, vol. 2, no. 1, pp. 68–85, 2016.
- Our first novel contribution is that we have developed a methodology that facilitates porting real-time simulations onto real hardware using available embedded benchmarks

(or if desired, in-house developed codes) as system tasks. Our methodology allows for the evaluation of real-time tasks on real-hardware at various system utilization points. Our methodology builds on previous work in estimating task WCETs, generating taskset utilizations, and generating task periods. The second innovative contribution is a set of algorithms that efficiently pair and assign the tasks temporal properties to the benchmarks set and in-house codes (if any). The generated tasksets satisfy a desired total system utilization with minimal errors. Our methodology for generating tasksets is more realistic as opposed to taskset generators that construct tasks by burning processor cycles through loops, matrix operations, or functional code blocks. The added realism is fundamental for faithful evaluation of energy-aware scheduling algorithms. This work has resulted in the following publications:

  - **A. Suyyagh** and Z. Zilic, "A Methodology for Constructing Tasksets for the Evaluation of Real-Time Workloads on Embedded Hardware", ACM Transactions on Embedded Computing Systems, (Submitted May 2018 - Currently being revised by the authors)
  - **A. Suyyagh** and Z. Zilic, "Real-time benchmark set synthesis based on pWCET estimation and bounded hyper-periods," 2017 International Conference on Circuits, System and Simulation (ICCSS), London, 2017, pp. 129-133.

- We propose heuristics for partitioning embedded workloads on single-ISA clustered heterogeneous platforms. We designed two versions of our *TCHAP* algorithm to consider two cases. The first case is when the system is able to transition into low-power mode where the transition overhead does not affect the system timeliness. The second variant is when the system stays in idle mode and does not transition into sleep mode. The *TCHAP* algorithm relies on two other proposed algorithms: *T-CAFE*, and *ACIF*. The former adds task-awareness to the problem of heterogeneous task allocation, whereas the latter selects initial cluster frequencies that satisfy initial schedule feasibility. We target ARM's big.LITTLE architecture which to our knowledge is the most successful and pervasive clustered single-ISA heterogeneous architecture. This work has resulted in the following publication:

  - **A. Suyyagh** and Z. Zilic, "Energy and Task-Aware Partitioning on Single-ISA Clustered Heterogeneous Processors", IEEE Transactions on Parallel and Distributed Systems, (In Submission - September 2018)

Our extensive work, practical, and teaching experience based on ARM Cortex-M processors family partially influenced parts of this thesis. The working knowledge of industrial RTOS (CMSIS based on RTX5) assisted in the development and analysis of the scheduling algorithms used in this thesis. The academic experience in teaching real-time embedded systems culminated in the following publication:

- **A. Suyyagh**, B. Nahill, A. Courtemanche, E. Kirshin, Z. Zilic and B. Karajica, "Managing the microprocessor course scope expansion," 2013 IEEE International Conference on Microelectronic Systems Education (MSE), Austin, TX, 2013, pp. 36-39.

Lastly, as the single author of this thesis, I declare that I solely conducted the collection of measurements and analysis of the generated data. That the methodology and all algorithms presented in this thesis are novel ideas that the I designed, implemented, analyzed, and evaluated on my own. I claim the sole authorship of this thesis and the first authorship of all published works and all works in submission pertaining to this thesis. I acknowledge that Prof. Zeljko Zilic in his role as an academic supervisor has offered many suggestions and helped in editing all manuscripts submitted including this thesis. I also acknowledge that Dr. Jason Tong helped in editing the published manuscripts related to Chapter 3.

1.3 Thesis Organization

The organization of the entire thesis is as follows: Chapter 2 presents the necessary background, terminology, and definitions used throughout this thesis. Sections 2.4 through 2.6 present a brief survey of the previous work related to energy-aware scheduling on uniprocessor and multiprocessor platforms, the use of metaheuristics in scheduling problems, and the approaches of constructing tasksets for real-time systems simulation and evaluation. Chapter 3 covers the implementation and evaluation of a discrete genetic algorithm-based approach *GAFS* and a discrete differential evolution based-algorithm *DEFS* in frequency to task assignment in DVFS-capable uniprocessors with DPM support. Chapter 4 presents the proposed methodology for porting real-time simulations onto real hardware. It also presents the *CPA* family of algorithms that pair the taskset temporal properties and assigns them to the system tasks. Chapter 5 presents a comprehensive description and model of clustered single-ISA heterogeneous processors. It further proposes two versions of the *TCHAP* algorithm used for energy-aware partitioning on ARM big.LITTLE platforms. Finally, Chapter 6 summarizes the thesis and presents venues for future research that extends the themes and ideas presented in this work.

Chapter 2

Background and Literature Review

This chapter begins with a brief background on real-time systems and real-time scheduling theory. It introduces concepts and definitions which are used throughout this thesis. Subsequently, this chapter overviews the processor power model commonly used in literature. It further covers the major hierarchical techniques employed by both industry and literature to reduce power consumption. Essentially, the chapter presents a survey of the most important research pertaining to energy-aware scheduling on unipocessors, homogeneous and heterogeneous multiprocessor systems. The chapter provides a brief overview on the use of metaheuristics in scheduling theory, and on the different techniques that construct real-time tasksets. Finally, the chapter concludes with a discussion of Performance Monitoring Units and associated tools.

2.1 Real-Time Systems Concepts

A real-time system is a system that not only depends on the correct functional output of its underlying tasks, but crucially on the timeliness of the tasks. That is, tasks should start at specific times and deliver their output within specific time frames called task deadlines. Real-time systems must be highly responsive to external stimuli, predictable, and have a deterministic behavior for safe operation. Given that the timing latitude to react in response to external triggers is limited, both the physical components (i.e. real-time processors) and software components (i.e. Real-Time Operating System) must provide for the means to conform to the timing constraints of the system. For example, a real-time processor must have short interrupt latencies and real-time schedulers must schedule tasks such that none of the tasks misses a deadline.

Real-time systems can be classified according to their deadlines as hard, firm, or soft real-time systems. Hard real-time systems have serious consequences should deadlines be missed often resulting in fatalities, injuries, physical damage, or economic and ecological disaster. Hard real-time systems comprise most of automotive systems, avionics, defense and nuclear systems, or any system with safety critical applications.

Firm and soft real-time systems, on the other hand, can usually accommodate deadline misses. The repercussions manifest in the loss of quality rather than severe injuries or total system failure. In firm real-time systems, the system survives complete failure as long as the deadline misses are infrequent. For example, multimedia applications with firm timing constraints on processing video frames still run adequately if the system is not able to process some frames within the deadline. This would result in jittering or loss of video quality. However, the system will not fail unless most or all frames are not processed in time. Soft real-time systems allow for frequent deadline misses as long as tasks do present correct results after the deadline has passed even though the usefulness of the result degrades after the deadline. Some home-automation and IoT applications fall into this category. For example, a system that collects room temperature readings and adjusts the air conditioning level falls into this category. A lag in acquiring and processing temperature sensor readings would not incur a system failure.

2.1.1 Task Models, Parameter Definitions, and Characteristics

Real-time systems have been designed with various task models. However, regardless of the model, each task τ_i in the set of N real-time tasks $\Gamma = \{\tau_1, \tau_2 \ldots \tau_N\}$ is specified by a set of timing parameters which are illustrated in Figure. 2.1:

1. **Task release time or arrival time** (r_i) parameter denotes the triggering time for the task execution request. That is, when a task arrives and becomes known to the scheduler. The task release time should not be confused with the task execution start time (activation time) which is decided by the scheduler. When all system tasks have the same release time, this model is called *simultaneous triggering.* The tasks are said to be *in-phase.* When tasks first arrive to the system at different times, the model is therefore called *progressive triggering.*

2. **Task activation or start time** denotes the time at which the scheduler selects the task from the ready queue and releases it onto the processor to begin execution.

3. **Task finish time** is the time at which the task completely finishes execution and yields the processor back to the scheduler.

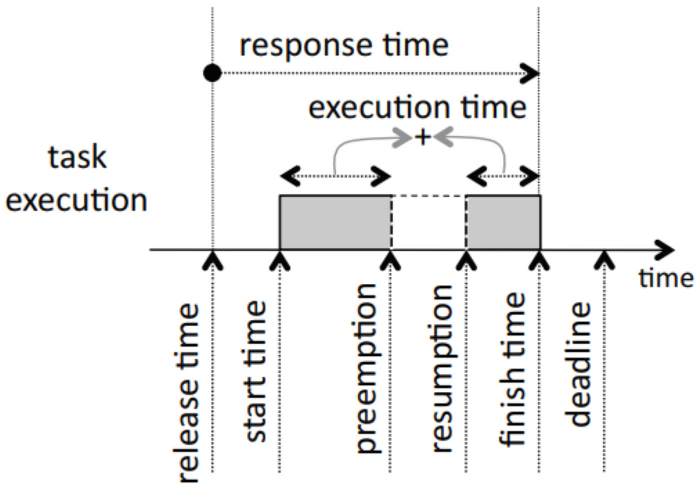


Figure 2.1 Summary of times associated with task execution [1]

4. **Task worst-case execution time (WCET) parameter** (c_i) denotes the task worst-case computation time required when the processor is fully allocated to its execution. This time does not necessarily equal the task finish time since the task might be interrupted and preempted few times during the course of its execution. The WCET only relates to the actual time spent on executing the task instructions and does not include the time when the task may be blocked or preempted. The notion of worst-case execution time stems from the fact that job releases will differ in their execution time. Tasks go through different computational flow paths in response to varying run-time inputs. The response to conditional statements and the length of loops might differ between task instances. The processor architecture and topology further affects the execution time (e.g. cache misses, no. of cache levels).

5. **Task period** (T_i): In the periodic task model, the task period denotes the regular interval between the arrival of the jobs of task τ_i.

6. **Task minimum inter-arrival time** (also T_i): In the sporadic model, this parameter denotes the minimum time interval between job releases of task τ_i.

7. **Task deadline** (d_i) denotes the maximum acceptable delay for task processing and providing correct results. In real-time systems, the task deadline is the most important parameter. It is worth mentioning that deadlines do not necessarily need to be short or enforce a sense of urgency. To clarify, the notion of a deadline is application dependant. Some deadlines are associated with fast response (e.g. car airbag controllers).

Others, for instance, can extend to minutes (e.g. some industrial temperature control applications). Task deadlines can be implicit, constrained, or unconstrained. In the implicit deadline model, task deadlines equal their periods or minimum inter-arrival time ($d_i = T_i$). An executing task instance must finish before the release of the next instance. A constrained deadline task has a deadline that is less or equal to its period or minimum inter-arrival time ($d_i \leq T_i$). The deadlines must be explicitly specified and known for each task τ_n. Implicit deadlines are therefore a special case of constrained deadlines. Unconstrained deadlines basically represent the general case where deadlines can be either less than, equal, or greater than task periods or minimum inter-arrival time ($d_i \leq T_i$ or $d_i > T_i$). Similar to the constrained deadline model, unconstrained deadlines must be explicitly specified.

8. **Task response time** is the time elapsed between the task release (arrival) time and its termination. The worst-case response time (WCRT) is the longest observed time from a job arrival to its completion.

9. **Task lateness** is the time difference between a finish time of a task and its deadline. In hard real-time systems, tasks that meet their deadline will have either zero or negative lateness. In soft-real-time system, it is possible to have positive lateness. Lateness is one criterion used to compare between scheduling algorithms. Many schedulers attempt to lower the maximum lateness of any taskset.

In real-time systems, the least common multiple of all task periods is called the *hyperperiod*($\mathbb{H}$). In mathematical notation, it is represented by Equation. 2.1:

$$\mathbb{H} = lcm(\ \forall \tau_n \in \Gamma\ ) \tag{2.1}$$

The quality and correctness of the scheduling depends on the exactness of these parameters. This is crucial for the determinism and safety of real-time systems. Furthermore, task switching (context switching), the scheduling overhead, and interrupt processing must not be neglected. During the design stage, these overheads must be analyzed and added to the task computation time.

In real-time systems, a task set has properties defined by one of these main task models:

1. **Periodic**: A periodic task is an infinite sequence of task instances (a.k.a jobs) whose arrival time is fixed in regular periods. The notation $\tau_{n,m}$ denotes the m^{th} instance (job) of task τ_n. In this model, tasks can be initially released at the same time (simultaneously triggered) or at different times (progressively triggered).

2. **Frame-based**: This model is a special case of the periodic task model where a set of tasks share the share the same release time ($t = 0$), the same period (called the *frame*), and the same deadline which is equal to the period. The length of the period is henceforth called the frame length. In each frame, tasks are executed in a predetermined fixed order.

3. **Sporadic**: This model is similar to the periodic task model with the main difference that task instances do not necessarily arrive at regular intervals. In this case, instead of periods, tasks have a minimum inter-arrival time parameter between task instances. Periodic tasks are a special case of sporadic tasks.

4. **Aperiodic**: This model is considered the hardest to schedule in real-time systems simply because tasks have no regular period or minimum inter-arrival time. Consequently, with no understanding of the job release behavior, the scheduling problem becomes a *best-effort* policy. That is; tasks are scheduled when there is time.

Tasks can be further classified as independent or dependent tasks. As opposed to dependant tasks, independent tasks have no precedence constraints nor a sequential order of execution to adhere to. For example, the process of acquiring a sensor's data, filtering the data stream, and processing it can be modeled by a set of small successive dependent tasks that must execute in a timely order. Alternatively, the whole process can be contained in one larger task that is independent of other system tasks.

In a real-time system, Equation 2.2 computes the processor utilization of any task:

$$u_i = \frac{c_i}{T_i} \tag{2.2}$$

where u_i is the task utilization, c_i is the worst-case execution time, and T_i the task period. For N periodic tasks, the total system utilization is given by Equation 2.3:

$$U = \sum_{i=1}^{N} \frac{c_i}{T_i} = \sum_{i=1}^{N} u_i \tag{2.3}$$

The processor load factor U^L (a.k.a density) is given by Equation 2.4. In implicit deadline periodic systems, the processor load and utilization are the same:

$$U^L = \sum_{i=1}^{N} \frac{c_i}{d_i} \tag{2.4}$$

For a system with support for frequency scaling and with a discrete frequency set $f = f_1, f_2, \ldots f_{max}$, the set $\lambda = \lambda_1, \lambda_2, \ldots \lambda_{max}$ denotes the scaling factors corresponding to the discrete frequency set f. The sets f and λ are of equal size. Equation 2.5 defines the frequency scaling factor as:

$$\lambda_i = \frac{f_{max}}{f_i} \tag{2.5}$$

When task execution times are assumed to scale linearly with frequency, the task utilization is updated to reflect this change in execution time. Equation 2.6 expresses the new utilization as:

$$u_i = \frac{\lambda_i . c_i}{T_i} \tag{2.6}$$

2.1.2 WCET Estimation

Estimating Worst-Case Execution Times (WCET) of real-time tasks is an arduous endeavour of extreme importance in time-critical and real-time systems. Given the different system initial conditions and the different task inputs throughout the system life-time, the execution time of any task varies. Figure 2.2 shows a distribution of task execution times for one possible task and the associated terminology. Most of the time, the task will run for shorter lengths than its WCET. Given the possibly large space of possible execution times of a task, there are no guarantees that one can observe the best-case execution time (BCET) or worst-case execution time (WCET) of any task under consideration.

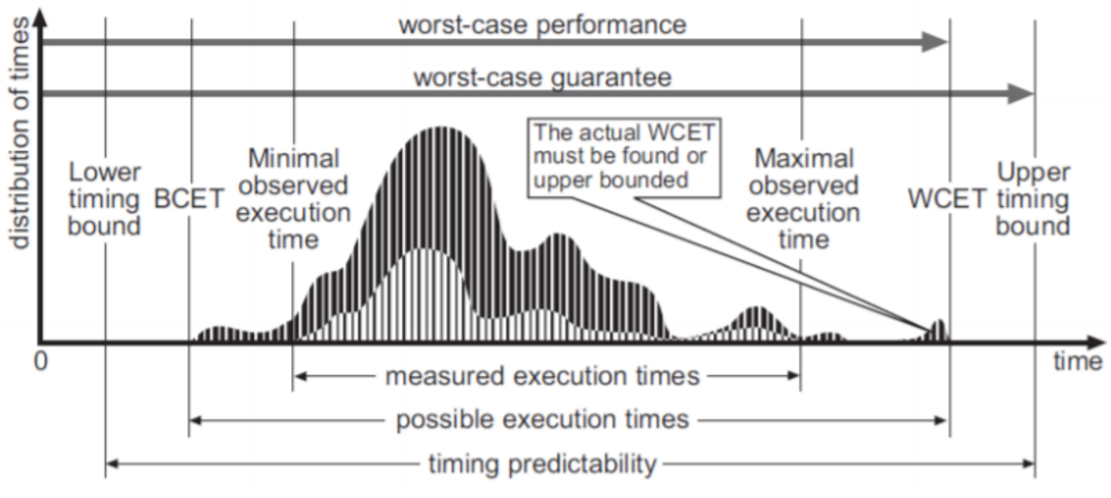


Figure 2.2 Timing analysis terminology of a task [2]

Various techniques exist that attempt to place lower and upper-bounds for the execution time of any task. The devised techniques largely fall under three approaches: static-based, measurement-based, and hybrid analysis. Each of the approaches can be carried in either deterministic or probabilistic fashion. Abella et al. [18] provides a survey of WCET analysis methods. Regardless of the method used to perform the timing analysis of system tasks, two criteria are essential in the of evaluation WCET estimation methods: *safety* and *precision*. Safety means that the obtained estimate must be higher than the WCET. Precision considers the closeness to the exact value of the WCET. The following sections briefly summarize the main approaches, their pitfalls, and strengths.

2.1.2.1 Static Timing Analysis (STA)

Static WCET analysis is based on analyzing the assembler code or machine level of the program and constructing a path-flow model. This requires user understanding and annotation of the code (e.g. specify number of loop iterations). Static techniques require an exhaustive analysis of all input values to the task, or a reduced set of values based on safe abstraction. Static WCET Analysis uses the path flow model in conjunction with a timing model of the target hardware platform. The accuracy of the timing model is fundamental. Recent advances in modern hardware and the introduction of complex features to levy up performance has complicated the low-level timing analysis procedure. Building accurate static analysis tools which incorporate knowledge of system-wide timing interaction (i.e. between processors, memory and I/O) has been increasingly challenging, especially that the hardware description in the abstract processor model must be highly detailed. The limited disclosure of technical and timing specifications has vastly exacerbated the issue. Common static WCET analysis tools include OTAWA [19], aiT [20], Heptane [21] and Bound-T [22]. However; the lack of support for many modern and multicore architectures (an open challenge), the cost of some of these tools, and the lengthy analysis time for numerous tasks makes them less of an option for a timely setup of a simulation platform. Yet, these tools provide tight and safe bounds on WCET and remain standard tools in industry.

2.1.2.2 Measurement-Based Timing Analysis (MBTA)

Measurement-based WCET estimation demands understanding of the task source code and potentially a large set of various inputs and initial conditions of the hardware (e.g cache state) that include the path which leads to the WCET. This is practically unfeasible for large systems. This approach requires accurate measurement procedures in the sense that the tools used to measure the time are not intrusive and should not introduce measurement errors. Despite the ability of Performance Monitoring Counters (PMCs) (refer to Section 2.6.1) to count the task execution cycles, they inherently pollute the measurements with the overhead used by the driver API that accesses them. The effect of the overhead on the measurement highly depends on the hardware and driver API, though it is often quite minimal. Underestimating the time affects the trustworthiness of the measurements, while overestimating the running time affects the tightness of the WCET bound. To measure execution times on embedded processors not equipped with performance counters, users rely on setting up a couple of flags. The first at the start of execution of the task, and the other right before the termination point. The flag signal goes through GPIO pins to oscilloscopes or logic analyzers where the measured time between the two flags denotes the execution time. Given the collected measurements, there is no clear consensus on how the WCET is determined. The maximum observed execution time does not necessarily suggest that it is the actual WCET. A practical approach is to add a safety margin that should be pessimistic yet tight. Typically, a certain percentage is added to the highest measured execution time (e.g. 10% to 20%). There is no scientific justification behind the criteria to select these percentage margins except that they work in most-cases!

2.1.2.3 Probabilistic Timing Analysis (PTA)

STA, MBTA, and their hybrid approaches can be either deterministic (DTA) or probabilistic (PTA). Deterministic approaches always provide a single WCET estimate for a set of given inputs and initial hardware conditions. They highly rely on determinism in the hardware design (e.g. fixed number of cycles for bus arbitration, fixed number of cycles to service interrupts ... etc). However, modern complex hardware designs are not inherently deterministic (e.g. shared caches and resources adds further interaction complexity). Probabilistic Timing Analysis (PTA) applies to both time-deterministic and time-randomized software and hardware resources. In contrast to a single WCET estimate, PTA computes a probability distribution function (*pda*) that encompasses a range of WCETs. The distribution features a prominent tail that extends towards long execution times. A single WCET is derived from the distribution at a certain probability which corresponds to the overall probability of failure according to a certain safety standard. As such, the WCET value is called pWCET to denote that it has been derived through probabilistic means.

When constructing the distribution function for pWCET estimation, the timing variable must be independent and identically distributed (*IID*). As such, a main requirement prior to using PTA approaches is randomization to ensure the *IID* property. For example, processors with caches that employ least recently used (LRU) replacement policy break the independence property as successive runs of the program are history dependent and blocks of code will always be placed/found at the same cache blocks addresses. In this case, software randomization is needed prior to using PTA approaches. Before each run, the task objects are randomly placed in memory in such a way that they would be mapped to different cache blocks during successive runs. On the other hand, a notable example of hardware randomization is using caches with Random Replacement policy (RR). These caches randomly map memory blocks to cache blocks eliminating the setup overhead of random object placement in memory. ARM Cortex-R and Cortex-A processors employ last level caches (LLC) that use random replacement policy.

PTA has been applied to both static timing analysis (SPTA) [23, 24] and measurement-based timing analysis (MBPTA) [25, 26, 27, 28]. The rationale behind adopting probabilistic WCET estimation (pWCET) is mainly the challenges of static WCET tools. Further, measurement-based techniques require extensive exploration time and initial test patterns than their probabilistic counterparts. Extreme Value Theory (EVT) is often used to conduct MBPTA. In contrast to central limit theorem where the focus is towards the mean of the distribution, EVT is concerned with the extremes of the distribution. Various applications employ EVT such as meteorology [29], finance [30], and astronomy [31]. Literature has covered few models to apply EVT for probabilistic WCET estimation. The Generalized Extreme Value (GEV) and the Gumbel distributions are notable examples [32, 33, 34, 35]. The Gumbel model is one of the models that are included in the GEV which also considers Weibull and Fréchet models.

2.1.3 Real-Time Scheduling

Scheduling is the core of any real-time operating systems. A scheduling decision comprises three main decision parts: timing, ordering, and assignment. Timing relates to the time at which a task executes. Ordering relates to which order any processor should execute the set of tasks in its scheduling queue. In multiprocessor and multicore platform, assignment is concerned by which processor runs which task. It is essential that we define the concept of a feasible schedule within the context of real-time scheduling:

Definition 2.1.1. Feasibility: A feasible schedule of taskset Γ is a timing schedule in which all tasks $\tau_n \in \Gamma$ meet their respective deadlines.

The notion of *feasibility* of a taskset Γ is independent of any particular scheduling algorithm. In contrast, the notion of taskset *schedulability* and *schedulability test* are defined relative to a scheduling algorithm:

Definition 2.1.2. Schedulability: A task set Γ is said to be schedulable by a given scheduling algorithm if the schedule generated by the scheduling algorithm respects all the deadlines of the tasks in Γ.

Definition 2.1.3. Schedulability test: For a taskset Γ scheduled by a real-time scheduling algorithm on platform Π, a schedulability test is a test on Γ determining if Γ is schedulable by the scheduling algorithm on Π.

A real-time schedule optimality is defined by the notions of *feasibility* and *schedulability*. In simple terms, an optimal scheduling algorithm is an algorithm which respects all task deadlines for any taskset for which a scheduling solution exists.

Definition 2.1.4. Scheduler optimality: A scheduler that yields a feasible schedule for any task set under a set of constraints (i.e. task model) for which there is a feasible schedule is said to be *optimal with respect to feasibility.*

Therefore, the comparison and analytical study of any scheduling algorithm comprises two parts:

- "The optimality of the algorithm in the sense that no other algorithm of the same class (e.g. fixed or variable priority, refer to Section 2.1.3.3) can schedule a task set that cannot be scheduled by the studied algorithm [36]". "A scheduling algorithm that delivers a feasible schedule whenever processor utilization is less than or equal to 100% is obviously optimal with respect to feasibility. It only fails to deliver a feasible schedule in circumstances where all scheduling algorithms will fail to deliver a feasible schedule. [1]".

- "The off-line schedulability test associated with an algorithm, allowing a check of whether a task set is schedulable without building the entire execution sequence over the scheduling period [36]"

There are different ways to group and classify scheduling algorithms. For instance, schedulers can be offline or online schedulers, fixed-priority or dynamic-priority schedulers, non-preemptive, limited- or fully-preemptive. The following sections present a brief discussion on each.

2.1.3.1 Offline and Online Schedulers

Schedulers are classified as online or offline depending on the time the scheduler makes it decisions:

- **Offline scheduling**: In offline scheduling, the taskset is fixed and known *a priori* in order to be able to calculate task activation times at design time. The scheduler executes over the entire taskset without actual task activation. The resulting schedule is stored in an *activation* table. At run time, instead of a scheduler, a task *dispatcher* consults the task activation table to decide which task to execute next given the pre-generated task schedule. This fully-static scheduling approach makes the three scheduling decisions at design time. Even though this approach reduces the scheduling overhead and is independent of the scheduling algorithm complexity, it is quite inflexible to operational changes. Furthermore, it is hard to realize on most modern microprocessor systems due to the difficulty in precisely predicting task execution times. Execution times are data-dependent and task behavior can be affected by the interaction with other tasks in the system. Another version of offline scheduling is a static order scheduler that decides processor-task affinity and task ordering at design time, yet decides the precise task activation time at run-time.

- **Online scheduling**: Online scheduling is a fully dynamic scheduling where all scheduling decisions take place at run-time. The scheduler runs whenever a new task enters the processor task ready queue, or a running task terminates, or when it blocks on waiting for a mutex to be released or an I/O operation to complete. However; it is worth noting that a static assignment scheduler where only task to processor assignment takes place offline, yet task ordering and activation at run time is also considered online scheduling. In online scheduling, tasks have either fixed priorities that are assigned to the tasks before their activation and never change throughout the system run-time, or dynamic priorities that change depending on tasks timing parameters through the course of system execution.

2.1.3.2 Real-Time Scheduling Preemption Levels

Schedulers can be non-preemptive, fully-preemptive, or have limited preemption capability. Non-preemptive schedulers allow every task to run to completion without any interruptions despite the fact that higher priority tasks are available in the task ready queue, or that the task is waiting on mutexes or I/O. The simplicity of the approach though appealing introduces higher risks of task deadline misses due to delayed task response time.

Fully preemptive schedulers can make scheduling decisions during the run time of any given task. It does not necessarily wait until executing tasks terminate or block. The scheduler can run periodically at specified intervals to make new scheduling decisions. It can be further triggered by new task arrivals in the task queue. Preemptive schedulers suspend currently running tasks in favor of higher priority tasks. Compared to non-preemptive scheduling, preemptive schedulers are highly responsive and reduce the possibility of missing deadlines. However, they incur scheduling and context switching overheads. Frequent context switching increases the traffic in the underlying memory and bus subsystems and consequently the energy consumption. For example, some data blocks related to the preempted and preempting tasks could possibly map to the same cache lines resulting in cache misses. Cache block replacement and fetching data and/or instructions from main memory often entails large energy and timing overheads. The work of [37] studied the relationship between preemption and cache behavior under fixed priority scheduling. Furthermore, the work of [38, 39] shows that the incorporation of DVFS hardware (see Section 2.3.3) increased the number of task preemptions by 500% and memory access by 55% in the worst case.

Preempting a lower priority task might not be always necessary in order retain schedule feasibility. Limited preemption is a hybrid implementation between fully-preemptive and non-preemptive scheduling. In some cases, the limited preemption model is able to schedule tasks sets which are deemed unfeasible under both non-preemptive and fully-preemptive scheduling [40]. Furthermore, limited preemption could reduce the number of preemptions by up to 90% while retaining schedule feasibility[41]. This effectively reduces task synchronization overhead and the required stack memory; a critical resource in memory constrained embedded systems[42].

A popular and simple approach to limited preemption scheduling is Preemption Threshold Scheduling (PTS) [40, 43]. In PTS, tasks have preemption levels and preemption thresholds. It is no longer sufficient for a task to have higher priority than another task in order to preempt it, but also it should have a preemption level that is higher than the running task preemption threshold. When preemption thresholds are set to the maximum, no task preempts another effectively rendering PTS to be non-preemptive. In contrast, when preemption threshold are set to the lowest possible threshold, PTS becomes fully-preemptive scheduling. As such, both non-preemptive and fully-preemptive scheduling are special cases of the PTS limited-preemption model. The survey of Buttazzo et al. [41] covers other techniques to implement limited preemption models and provides a comparative performance analysis. ThreadX [44] is one commercial RTOS that implements the PTS form of limited preemption.

2.1.3.3 Fixed and Dynamic Priority Schedulers

Priority-based scheduling assigns numbers to the tasks called task priorities. A real-time scheduler will choose the task with the highest priority to execute first. Priorities can be either fixed and remain constant for all task instances throughout the system operational time or can change dynamically. In some scheduling algorithms, system designers must specify the task priority levels at design time and ensure that the system demonstrates correct and safe behavior all the time for all possible cases. For example, this approach is used when the scheduler uses Round-Robin scheduling where tasks alternate in filling execution time slots.

The most prominent fixed-priority scheduling algorithms are the Rate Monotonic (RM) and Deadline Monotonic (DM, a.k.a inverse deadline) algorithms. In a periodic and fully-preemptive model with independent tasks, the RM algorithm schedules tasks according to their arrival rate. Tasks with the shortest periods are given the highest priority. To demonstrate, a simple form of the RM scheduling problem considers two tasks $\Gamma = \{\tau_1, \tau_2\}$, with worst-case execution times $\{c_1, c_2\}$, and periods $\{T_1, T_2\}$. The execution time c_2 of task τ_2 is longer than the period T_1 of task τ_1. Thus, if these two tasks are to execute on the same processor/core, then it is obvious that a non-preemptive scheduler will not result in a feasible schedule. However, a fully-preemptive RM scheduler will. Task τ_1 will be assigned higher priority since $T_1 < T_2$ and is able to preempt τ_2 whenever it is executing. Figure 2.3 illustrates the preceding example.

The rate monotonic scheduler is optimal with respect to feasibility for the above model among all fixed-priority uniprocssor schedulers [45]. It is worth noting that the RM algorithm cannot achieve 100% utilization. The Liu and Layland bound [45] given in Equation 2.7 provides a sufficient schedulability condition for RM scheduling:

$$U = \sum_{i=1}^{N} \frac{c_i}{T_i} \leq N.(2^{\frac{1}{N}} - 1) \tag{2.7}$$

where U is the processor utilization, N is the number of tasks in the taskset, c_i and T_i are the task WCET and period, respectively. For a system with a large number of tasks N, the upper bound converges to $\ln(2) = 0.6931$. That is, for a schedule to remain feasible under RM scheduling, the processor must remain idle for about 31% of the time when the number of tasks is large enough.

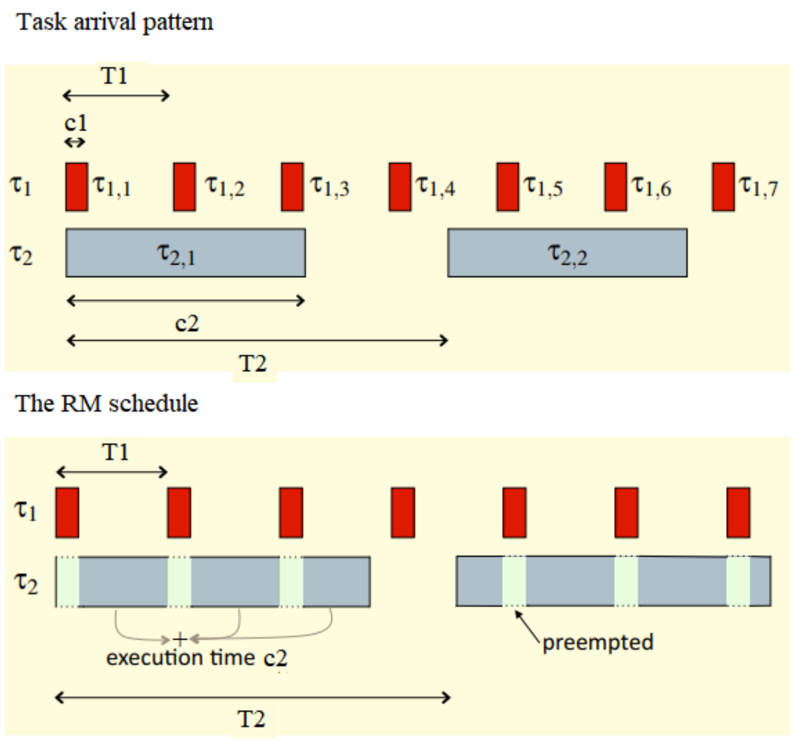


Figure 2.3 RM scheduling of two tasks τ_1, τ_2 where task periods $T_1 < T_2$ [1]

Rate Monotonic scheduling only works for an implicit deadline model. For an explicit deadline model where the condition that tasks deadlines equal their periods is relaxed, the deadline monotonic scheduling must be used instead. In a similar fashion to RM scheduling, deadline monotonic assigns task priorities to tasks according to their relative deadline. Tasks with the shortest relative deadlines are assigned higher priorities. Within the set of scheduling algorithms that target tasksets where tasks have deadlines shorter than their respective periods, deadline monotonic scheduling is optimal with respect to feasibility. A sufficient schedulability condition for the inverse deadline scheduling algorithm is given by Equation 2.8:

$$U = \sum_{i=1}^{N} \frac{c_i}{d_i} \leq N.(2^{\frac{1}{N}} - 1) \tag{2.8}$$

where U is the processor utilization, N is the number of tasks in the taskset, c_i and d_i are the task WCET and deadline, respectively.

In dynamic priority class of scheduling algorithms, the Earliest Deadline First (EDF) is the most important algorithm. The EDF algorithm applies to both independent periodic and sporadic tasks under the fully-preemptive model. The EDF strategy is based on *absolute* task deadlines. From a set of available in the ready queue, EDF assigns the task with the earliest deadline the highest priority and selects it for execution. Given that the set of tasks in the task ready queue varies during the system operation, so does the absolute deadlines, and consequently the task priorities will vary accordingly. That is, the task instances $\tau_{n,1}, \tau_{n,2}, \ldots$ will not necessarily have the same priority. If two tasks have the same deadline, their relative order is of no consequence. EDF is optimal with respect to feasibility. Equation 2.9 sets a necessary and sufficient condition for EDF schedulability for tasks with implicit deadlines:

$$\sum_{i=1}^{N} \frac{c_i}{T_i} \leq 1 \tag{2.9}$$

where N is the number of tasks in the taskset, c_i and T_i are the task WCET and period, respectively.

Aside from its optimality, a major advantage of EDF is that it allows for full utilization of the processor time towards useful computation. It further reduces the taskset maximum lateness compared to RM scheduling.

2.1.3.4 Real-Time Scheduling in Multiprocessor Systems

Traditionally, there are two main approaches for multiprocessor scheduling with a third hybrid approach that combines between the two. Figure 2.4 illustrates the different multiprocessor scheduling techniques.

- **Global scheduling**: This approach utilizes one global scheduler and one shared task queue for all tasks in the system. The scheduler decides the job-to-processor assignment at run-time. Task instances can be run on any core depending on global scheduling priorities at the scheduling instance. As such, tasks freely migrate between the cores or processors. Due to task migration, global scheduling can better achieve online load balancing and offers lower average response time than partitioned global scheduling. However, task migration comes with overhead costs and the loss of cache affinity.

- **Partitioned scheduling**: In this approach, each core or processor has its own scheduler and independent task queue. Task to processor affinity decisions are a design time parameter and thus determined *a priori*. Despite the extra cost in implementing separate schedulers, partitioned scheduling allows the treatment of each core or processor as a uniprocessor core. Hence, uniprocessor scheduling algorithms such as RM

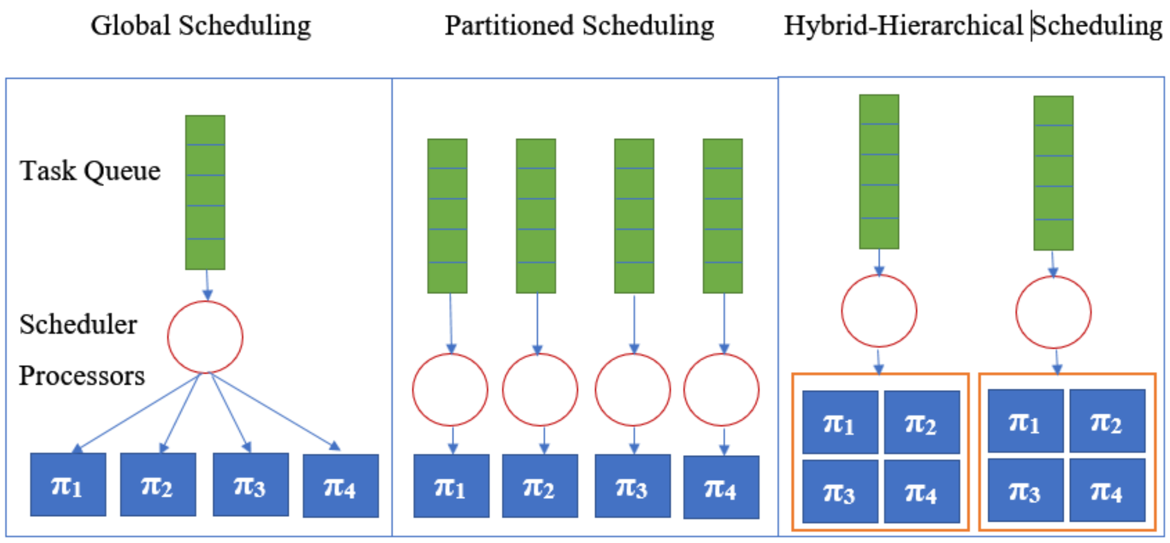


Figure 2.4 The different approaches for scheduling on multiprocessor systems

and EDF are readily applicable. Each scheduler runs on the subset of tasks associated with the core/processor. Partitioning further allows for the use of well-known single processor feasibility checks and analysis. However; task migration is not allowed. Task to processor affinity remains unaltered throughout the system run-time.

- **Hybrid Approaches**: One hybrid approach called *Semi-partitioned scheduling* offers a trade-off between partitioned and global scheduling. It relaxes the constraint that no tasks or jobs migrate after partitioning therefore enabling limited task migration with the goal of improving system utilization. In another approach called *Hierarchical scheduling*, tasks are partitioned among different processing groups (i.e. multiprocessors, clusters .. etc) and each group has a global scheduler.

Optimal scheduling algorithms for uniprocessor systems do not readily extend to global multiprocessor scheduling. In a result known as the “Dhall’s effect” [46], Dhall and Liu demonstrated that global RM (G-RM) and global EDF (G-EDF) schedulers are not optimal with respect to feasibility in global multiprocessor scheduling. However, this is an issue directly related to high utilization [47]. Compared to single processor scheduling, and in the general case, no online global scheduling algorithm exists that is optimal with respect to feasibility. This result is stated in Theorem 1 [36, 48].

Theorem 1. An online algorithm which builds a feasible schedule for any set of tasks with deadlines with m processors ($m \geq 2$) cannot exist. [1]

A necessary condition for multiprocessor schedulability is given by Equation 2.10. This simply means that the total utilization must not exceed the system capacity:

$$\sum_{j=1}^{m}\sum_{i=1}^{N}\frac{c_i}{T_i} \leq m \tag{2.10}$$

where m is the number of multiprocessors/cores in the system, N is the number of tasks in the taskset, c_i and T_i are the task WCET and period, respectively.

In consequence of the above findings, the research community has for long favored the partitioning scheme. Unfortunately, the partitioning scheme is limited by the performance of the allocation algorithm. Task partitioning is considered a *bin packing* problem. The problem of bin packing is concerned with filling a finite number of bins of a given capacity by a set of objects of different volumes to minimize number of bins used (or other criteria). This problem is known to be NP-Hard [49]. However, there exists a set of efficient bin-packing heuristics:

1. **Worst-Fit** assigns each task to the processor with the highest unused utilization (most remaining capacity).

2. **Best-Fit** attempts to assign the task to the most loaded processor as long as it has remaining capacity. Otherwise, it moves to the next loaded processor and so on.

3. **Next-Fit** starts assigning tasks to processors in sequential order of processors. It begins with the first processor until it is full or if the task does not fit in the remaining capacity of the processor. In this case, this processor is considered “closed”. The heuristic proceeds with assigning tasks to the next sequential processor. The next-fit heuristic never revisits “closed” processors even if there exists tasks that fit their remaining capacity. This heuristic only has to keep track of the remaining capacity of the currently “open” processor. However, it is clear that the next-fit heuristic is inefficient in term of maximizing utilization.

4. **First-Fit** is similar to the next-fit heuristic with the main difference that it does not “close” processors. It attempts to fill the processors in sequential order from the beginning thus capitalizing on any remaining capacity that fits the task.

A minor variation of the above bin-packing heuristics orders the tasks in descending order of task utilization. These are commonly known as Worst-Fit-Decreasing (WFD), Best-Fit-Decreasing (BFD), Next-Fit-Decreasing (NFD), and First-Fit-Decreasing (FFD).

2.1.4 Commercial Real-Time Operating Systems and Real-Time Linux

Numerous RTOSes are available for deployment in industrial and commercial applications either under proprietary or free public license. Notable mentions are Windows CE [50], ThreadX [44], ARM's RTX Keil [51], MicroC/OS-II [52], FreeRTOS and SafeRTOS [53]. In automotive industry and under the AUTomotive Open System ARchitecture $AUTOSAR$ standard, the real-time scheduler assigns higher task priorities to high-rate tasks. This approach is similar to the Rate Monotonic (RM) scheduling algorithm. VxWorks [54] is a proprietary RTOS that features a priority-based preemptive scheduler and a fully-preemptive round-robin scheduler. VxWorks is heavily used in spacecrafts and avionics. Notable deployments include the Curiosity rover, Phoenix Mars lander, SpaceX Dragon, James Webb Space Telescope, and Boeing 787 Dreamliner.

Mainstream Linux distributions has no native support for real-time scheduling. The *Linux_RT_PREEMPT* patch enables kernel preemption and real-time API to allow for real-time support. Linux introduced a scheduler *SCHED_DEADLINE* based on Earliest Deadline First and Constant Bandwidth Server (CBS) starting with kernel version 3.14 [55]. Commercial Real-Time Linux or Unix-like examples include LynxOS [56] and RTLinux [57]. Similar to the non-linux based RTOSes, RTLinux is based on first come first serve (FIFO) scheduler with support for priority. LynxOS scheduler features four scheduling policies, FIFO, Priority Quantum, Round-Robin, and non-preemptive. These RTOS solutions provide predictable and deterministic behavior rather than optimal scheduling. The underlying schedulers do not guarantee hard-real time operation. It is the responsibility of the designer to analyze the schedules through offline timing tools to ensure feasibility.

In literature, given that the focus is on optimal scheduling algorithms, the Linux Test-bed for Multiprocessor Scheduling in Real-Time Systems (LITMUS^{RT}) [58, 59] provides a useful experimental platform for applied real-time systems research. It modifies the Linux kernel through patches to add support for periodic and sporadic task models as well as the numerous scheduling algorithms used by the research community. These include Partitioned EDF with synchronization support (PSN-EDF), Global EDF with synchronization support (GSN-EDF), Clustered EDF (C-EDF), and Partitioned Fixed-Priority (P-FP) among few others. LITMUS^{RT} is actively maintained, widely used in literature, and offers the opportunity for reproducing and comparing results [60, 61, 62, 63, 64, 65].

2.2 Processor Power Modelling

Modern processors have three modes of operation: *active, idle*, and at least one *sleep mode* level. *Active* mode is the mode in which the processor is executing system tasks. In this mode, the frequency of the processor is assumed to not vary throughout the run-time of the task. In the *idle* mode, the processor is powered on but no tasks are running. Sleep modes denote various states where the processor (its various internal components) are powered off. Processors could have multiple sleep modes which offer different power savings by powering off various components in the microprocessor. The deeper the sleep mode is, the more internal processor components are powered down. Depending on the depth of the sleep mode, the internal state of the inactive components can be either retained (drowsy state), or be lost (destructive). Waking up from deeper sleep mode states where internal state and register data has been lost incurs timing and power overheads.

Processor power consumption is modelled by a set of equations which have been repeatedly verified by SPICE simulation and adopted in literature [66, 67]. In the active mode, Equation 2.11 gives the total processor power as:

$$P_{active} = P_{Dynamic} + P_{Static} + P_{Independent} \tag{2.11}$$

where $P_{Dynamic}$ is the power consumed by the transistors during switching activity and is proportional to the run-time voltage and frequency. P_{Static} is the power consumed due to leakage current, and $P_{Independent}$ is the power consumed by various processor components not directly related to task processing (i.e. power regulators and Phased Locked Loops (PLLs)).

The dynamic power for a given frequency and voltage level is given by Equation 2.12:

$$P_{Dynamic} = C_{eff} V_{dd}^2 f \tag{2.12}$$

where C_{eff} is the effective transistor switching capacitance, V_{dd} is the supply voltage and f is the clock frequency. Equation 2.13 demonstrates the relationship between the operating frequency, supply voltage, and threshold voltage :

$$f = \frac{(V_{dd} - V_{th})^{\gamma}}{L_d K_6} \tag{2.13}$$

where L_d is an estimated parameter related to the average logic depth of the critical path for all instructions supported by the processor, K_6 is a technology constant, and V_{th} is given by Equation 2.14:

$$V_{th} = V_{th1} - K_1.V_{dd} - K_2.V_{bs} \tag{2.14}$$

where V_{th1}, K_1, and K_2 are platform-specific technology constants and V_{bs} is the body bias voltage (i.e. voltage between transistor's body and source).

The static power is largely dominated by the effects of the subthreshold leakage current I_{subn} and the reverse bias junction current I_j. Sub-threshold leakage current results from current flow between source and drain terminals of circuit transistors. Equation 2.15 provides the means to compute the static power:

$$P_{Static} = L_g.(V_{dd}I_{subn} + |V_{bs}I_j|) \tag{2.15}$$

where L_g is the number of devices in the circuit. The subthreshold leakage current I_{subn} is given by Equation 2.16:

$$I_{subn} = K_3 e^{K_4 V_{dd}} e^{K_5 V_{bs}} \tag{2.16}$$

where K_3, K_4 and K_5 are technology constants. V_{bs} is constrained to have a value between 0 and -1V such that the junction leakage power would not override gains in lowering I_{subn}.

In the *idle* mode, the processor consumes considerable power yet much less than the *active* mode. *Sleep* modes consume much less power than the idle mode to virtually no power in a complete shutdown state.

Processors and devices should be transitioned from their idle states to a lower power state only when the transition is power-efficient. The decision to switch a processor or any peripheral device to a lower power sleep state is based on the break-even time t_{BET} parameter. The t_{BET} represents the minimum sleep time length that the processor is required to stay in at sleep mode in order to make the switch to a lower power sleep state power-efficient. Equation 2.17 computes the break even time for a processor [68]:

$$t_{BET}(CPU) = max(t_{sw}^{CPU}, \frac{E_{sw}^{CPU} - P_{sleep}^{CPU} \times t_{sw}^{CPU}}{P_{idle}^{CPU} - P_{sleep}^{CPU}}) \tag{2.17}$$

where t_{sw}^{CPU} and E_{sw}^{CPU} are the processor time and energy switching overhead from idle to sleep state, respectively. P_{sleep}^{CPU} and P_{idle}^{CPU} denote the processor power consumed in the sleep and idle states and $P_{idle}^{CPU} > P_{sleep}^{CPU}$.

Similarly, Equation 2.18 computes the break even time for any device dev_k.

$$t_{BET}(dev_k) = max(t_{sw}^{dev_k}, \frac{E_{sw}^{dev_k} - P_{sleep}^{dev_k} \times t_{sw}^{dev_k}}{P_{idle}^{dev_k} - P_{sleep}^{dev_k}}) \tag{2.18}$$

where $t_{sw}^{dev_k}$ and $E_{sw}^{dev_k}$ are the device time and energy switching overhead from idle to sleep state, respectively. $P_{sleep}^{dev_k}$ and $P_{idle}^{dev_k}$ denote the device power consumed in the sleep and idle states and $P_{idle}^{dev_k} > P_{sleep}^{dev_k}$

2.3 Power Reduction Techniques

In this section, we present some of the major approaches in the past few years that aim to reduce power consumption in modern processors. These approaches are hierarchical and span from transistor manufacturing technologies at the lowest level, through architectural innovations, to chipset technologies, and finally software-level approaches (e.g. energy-aware scheduling, compiler optimizations). Section 2.4 presents Energy-aware scheduling in more detail.

2.3.1 Silicon-Level Power Reduction

As CMOS technology keeps scaling down, the leakage current increases substantially, and by extension the static power. To curb the effects of static power, both industry and literature have brought forward recent innovations at the silicon level. In the past few years, newer transistor designs replaced the traditional planar FET transistors in CMOS circuits. Technically and commercially known as multigate and Tri-Gate (3D) transistors (a.k.a FinFET technology), these newer technologies reduced leakage current up to 90% and cut switching power by 50% [69]. Major foundries like TSMC, Samsung, and Intel produce chipsets with FinFet technology [70]. Figure 2.5 illustrates how the introduction of FinFET transistor technology has reversed the trend of leakage current taking a larger share of total circuit power as the process technology continues to shrink.

The International Roadmap for Devices and Systems (IRDS) expects CMOS technology to be nearing its scaling limit by mid 2020s (5nm and 3nm for FinFET) [71], and suggests future chips will use chip stacks and monolithic 3D technologies to maintain performance and power gains. Concurrently, research for candidate technologies continues. Examples include carbon nanotubes [72, 73, 74, 75], Tunnel FETs (TFET), nanosheet FET [76], and graphene based technologies [77].

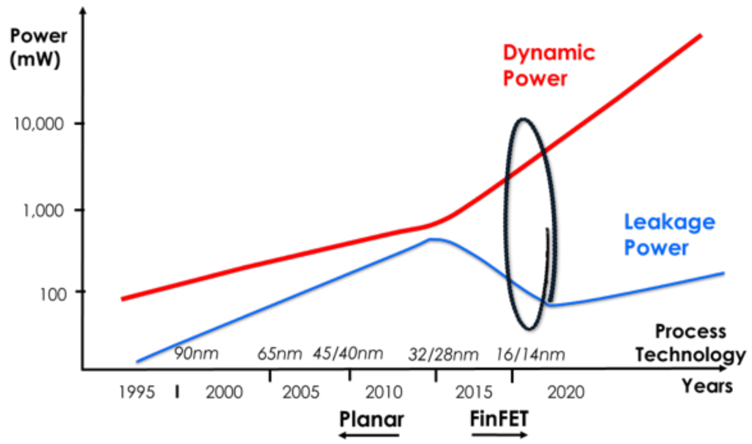


Figure 2.5 SoC Power Consumption Trends [3]

2.3.2 Architecture-Level Power Reduction

Architecture-level power reduction revisits traditional block designs in processor cores with the aim of reducing circuit power. The work of Mittal [78] surveys techniques for designing and managing CPU register files in modern processors, some of which cover power reduction techniques. Similar works address energy-efficient register file architecture in GPUs [79, 80]. Branch prediction (speculation) minimizes power by avoiding the impact of flushing pipelines executing the wrong execution sequence. Improvements to the design of branch predictors offer further power reductions [81]. Yet, these approaches remain in limited use.

Recently, single-ISA heterogeneous clustered multicore processors represent the major innovation in architecture-level power reduction. ARM's big.LITTLE [82, 83, 84] is the most successful commercial implementation of the clustered heterogeneous architecture. The traditional ARM's big.LITTLE architecture specifies two clusters. The first, termed *big*, has either two or four out-of-order high-end speculative cores aimed to deliver high application performance. The second cluster, termed *little*, has four in-order smaller cores which are energy-efficient compared to the *big* cluster. Each cluster has its own power and frequency controllers. In general-purpose computing scheduling, tasks migrate between clusters depending on application performance and power needs.

Some literature proposed pushing heterogeneity from processor level into the core level [85]. That is, instead of having separate big and little cores with separate caches and architectural components, each core will have two μEngines, one big and another little μEngine. The μEngines share the same cache and fetch units. This proposal allows for fine-grain task switching. That is, tasks could switch between μEngines at a granularity of hundreds and thousands of instructions instead of hundreds of thousands or millions of instructions. Since the μEngines are within the core and share processor blocks, such design consumes less switching overhead in terms of both time and energy.

2.3.3 Chipset-Level Power Reduction

Processor designs have taken advantage of Moore's law and the increased transistor density in many ways. These range from architectural innovations within the core design, to increasing the core count, and complete System on Chips (SoCs). SoCs are widely used in embedded systems [86, 87] and mobile devices. SoCs pack processors, peripherals (e.g. timers, ADCs, DACs), and various connectivity interfaces (e.g. SPI, USB, I^2C) on the same die. However, in a typical embedded application, some of these peripherals might go unused. Since it is necessary to curtail the overall system energy consumption, chipset designers utilize two major techniques for power reduction: power gating and clock gating.

In power gating [88, 89], processor cores, clusters, and peripherals have separate power regulators or power planes (voltage islands). This allows for turning off inactive cores or shut down unused or idle peripherals completely (e.g. ADCs). Power gating reduces leakage current yet introduces wake up latencies when these cores and peripherals activate[90]. In a typical SoC, clock generation, distribution, and clock buffers consume a large proportion of system power (15% - 45%) [91, 92]. To lower the power consumption of the clock-tree, clock gating enables system programmers to cut off the clock from inactive and unused SoC components (e.g. SoC timers).

A revolutionary step in power reduction is the incorporation of specialized hardware that dynamically controls the voltage and/or frequency of active cores at run-time. Operating systems can directly handle the DVFS-capable hardware through specialized drivers. *Ideal* DVFS processors expose an unrestricted continuous range of voltages and frequencies, a well-defined power-frequency relationship, and no speed change overhead. However, due to the extreme cost, complexity, and impracticality of ideal DVFS hardware, real-life implementations provide a limited set of voltages and frequencies and incur speed alteration cost. These systems are known as *non-deal* DVFS systems. In multicore processors, if the cores share the same power regulator, then their voltage and frequency scale globally. This model is called a dependent DVFS system. In independent DVFS multicore systems, each processor would have its own power regulator. In practice; however, in the majority of commercial multicore implementations, the cores within the same cluster share the power and DVFS hardware. Incorporation of independent voltage regulators and per-core DVFS hardware is restricted due to design complexity and increased chipset area. As the core count increases, per-core DVFS hardware introduces scalability and thermal dissipation issues [93].

2.4 Related Work to Energy-Aware Scheduling

Energy-Aware Scheduling couples the schedulability problem with that of energy-reduction. At its core, energy-aware scheduling is an optimization problem. Energy-Aware Scheduling is a software level approach to energy-reduction. Most commercial products attempt to offer solutions for non real-time schedulers, that is for the mainstream general computing platforms that power desktop and mobile devices. ARM and Linaro introduced EAS as a Linux Kernel enhancement to Linux power management. "EAS extends the Linux kernel scheduler to make it fully aware of the power/performance capabilities of the CPUs in the system, to optimize energy consumption for advanced multi-core SoCs including big.LITTLE" [94]. Whereas some commercial solutions are available for energy-aware schedulers for general purpose computing, energy-aware scheduling for real-time systems remains largely confined within the research community. In this section, we summarize the main energy-aware scheduling techniques for real-time uniprocessor and multiprocessor systems.

2.4.1 Energy-Aware Scheduling on Single Core Processors

Low power non-DVFS single core embedded processors mainly depend on utilizing the multiple sleep modes available to reduce power. High-end DVFS-capable processors highly rely on DVFS to amortize on energy costs. Integrated DVFS and DPM scheduling is an alternative to two phased approaches that start with task frequency assignment followed by DPM. Figure 2.6 illustrates the taxonomy of energy-aware scheduling on uniprocessors. Offline solutions make frequency assignment and sleep decisions at design time after thorough investigation of the task schedule while online solutions defer these decisions to execution time.

Early scheduling literature considered both *ideal* and *non-ideal* DVFS-enabled hardware [95, 96, 97, 98, 99]; however, these works discarded the effect of leakage power in the optimization problem. When leakage power is factored in, the work of Jejurikar and Gupta [100] showed that lower frequencies do not result in energy reductions. When both dynamic power and leakage power effects are simultaneously considered, they illustrated that the processor's *energy-per-cycle* metric takes a concave shape over the available DVFS supported frequencies. They introduced the concept of *critical frequency* at which both leakage power and dynamic power (i.e. *energy-per-cycle*) is minimum. However, despite the energy reduction

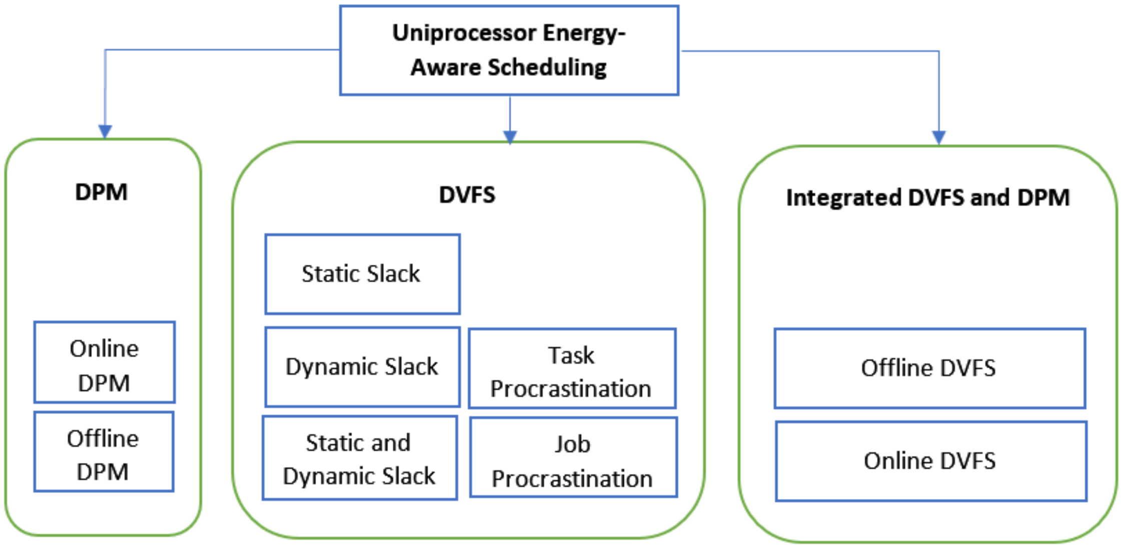


Figure 2.6 Taxonomy of Energy-Aware Scheduling on Uniprocessors

advantages at running the taskset at the critical speed, the schedule feasibility constraints might necessitate running the tasks at subsequent higher frequencies to meet task deadlines.

In any task schedule, the *slack time* is the time during which the processor is idle. The slack time can be used to reduce energy. The most common way is to stretch the execution time of tasks onto the slack by lowering the processor frequency. In such case, the slack is said to be reclaimed. Task reclamation algorithms can be static or dynamic.

Static algorithms are design time solutions. They are unaware of the actual task execution time. The slack reclamation is based on WCET. Therefore, the extra slack difference between the task actual execution time and its WCET goes unclaimed. In this category, Bini et al. [101] introduced the BBL algorithm for static slack reclamation for discrete frequency processors running EDF/RM schedulers and periodic and sporadic tasksets. Dynamic slack reclamation is an online algorithm. Therefore, it is able to capitalize on the extra slack generated when tasks finish early. The Bonus Sharing algorithm (BSDVFS) and its variant (BSDVFS*) [102] are prime examples of dynamic slack reclamation algorithms. The authors integrated their algorithms into the Erika RTOS [103].

A recent study by Bambagini et al. [104] showed that DPM algorithms work generally better than DVFS on actual hardware (due to the rise of static power share of the total power). As such, it might be beneficial to have as much slack as possible. This allows for keeping the processor in sleep mode for a longer time. Also, it potentially increases the idle interval such that it might exceed the break-even time (see Equation 2.17) thus making the switch to sleep mode energy-efficient.

To extend the slack time, whenever the processor is idling or in sleep mode, task procrastination techniques delay arriving tasks as much as possible without jeopardizing missing deadlines. Lee et al [105] were the first to introduce the concept of task procrastination. But their approach solely targets leakage power reduction on non-DVFS platforms. Jejurikar and Gupta [106] were the first to compute the procrastination delay for every system task based on task set utilization and EDF feasibility in a DVFS platform. The procrastination delay remains constant for every future task instance. The same authors extended their method in [100] by coupling it with slack reclamation for further energy reduction. Chen and Kuo [107] presented a similar approach for the rate monotonic scheduling policy.

The work of Niu and Quan [108] consider procrastination at the job level. Their DVSLK algorithm considers all job instances over the entire schedule. Their approach is computationally extensive. Pan and Lin [109] extended the DVSLK algorithm to take into account slack reclamation. Chen and Kuo [110] showed that many procrastination algorithms [106, 108] are greedy in the sense that they could sacrifice better energy reductions in the future in favour for less than optimal early savings. As a solution, they propose an algorithm called P-Procrastination that decides whether to procrastinate tasks early or in the future based on a threshold variable P. The above-mentioned works consider the periodic task model. Awan et al. [111] addressed the calculation of the procrastination interval for the sporadic task model.

System-level power-aware scheduling also received much attention especially in embedded systems applications with heavy peripheral devices use. As real-time tasks might use one or more devices, the literature assumes that these devices are ready at task start time and remain in an active state for the whole duration while the associated task is executing. This follows from the need to minimize blocking or waiting time of the tasks on devices to switch on. As task execution time is slowed down, then the expected time the associated devices remain in the active state is equally extended, thus consuming more energy. Consequently, system-level energy-aware scheduling algorithms attempt to minimize power across all system components. The work in [112] is the first to consider combined processor DVFS and devices Dynamic Power Management (DPM). Based on the notion of critical speed, they develop the CS-DVS algorithm which reduces total system power. This algorithm remains the most popular within the same class of algorithms. Similarly, [113] proposes a system-wide power-aware algorithm which assumes that some tasks have uninterruptable I/O access and therefore are non-preemptible. They further take device switching time into account. Their algorithm called SYS-EDF selects task slow-down factors that reduce system energy.

However, this works ignore the processor's leakage power.

Aydin et al. [114] present a system-wide solution that considers leakage power in an ideal DVFS-capable system. However, this model unrealistically assumes continuous voltage and frequency scales. For the rate monotonic policy, Niu [115] introduces a solution which couples the techniques of critical speed, dynamic slack reclamation, and DVS and DPM to minimize the system-wide energy of processor and peripheral devices. Their approach uses a two-phase DVFS/DPM approach instead of an integrated approach. In [116], the author shows that running tasks lower than the critical speed could possibly introduce energy saving in system-level scheduling. Soft real-time system-level scheduling is addressed in [117].

Devadas and Aydin [118] argue that the previous approaches which consider adjusting voltage scales in one phase followed by DPM optimization in the next phaze may not lead to optimal energy savings. Instead, they investigate the interplay between DVFS and DPM and propose a combined DVFS/DPM algorithm. The before-mentioned solutions [112, 113, 114, 116, 117, 118, 119, 120, 121, 122] are based on offline or semi-online energy-aware scheduling approaches and assume fixed task sets. The work of [123] is the first to implement an online system-level energy-aware scheduling. Their work is based on the simulated annealing metaheuristic. However, metaheuristics require a lengthy time to find a solution. In online scheduling, this increases the scheduling overhead to possibly a point that affects schedule feasibility.

2.4.2 Energy-Aware Scheduling on Homogeneous Multiprocessors

Energy-aware scheduling on multiprocessors considers additional optimization variables. These variables include task partitioning, the number of active cores, and processor heterogeneity. The majority of research considers homogeneous multiprocessors while some efforts attempt to generalize some solutions onto heterogeneous platforms. In general, the taxonomy of energy-aware scheduling on multiprocessors systems is based on the frequency scaling controllers. Figure 2.7 illustrates that research has divided energy-aware multiprocessor scheduling approaches into per-task DVFS, per-CPU DVFS, and clustered cores (a.k.a voltage islands). Per-task DVFS assigns frequencies at the granularity of tasks. Per-CPU DVFS assigns a fixed frequency for each processor/core in the platform. This frequency remains constant for all tasks. Lastly, in clustered multicore systems, a group of cores share the same DVFS hardware. In effect, the frequency scales globally for all cores.

The authors in [124] target frame-based tasks in a platform where all the cores share continuous voltage and frequency levels and ignore the effects of leakage power. Their work

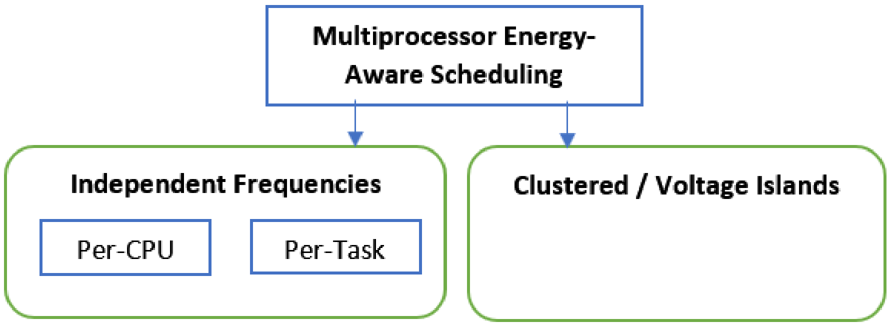

Multiprocessor Energy-Aware Scheduling
Independent Frequencies
Per-CPU
Per-Task
Clustered / Voltage Islands

Figure 2.7 Taxonomy of Energy-Aware Scheduling on Multiprocessors

uses the Largest-Task-First (LTF) for task assignment. No task migration is allowed. They prove that the problem is NP-hard and show that balancing the loads across the homogeneous cores yields the most energy-efficient schedule. For the same task model, the authors in [125] consider both dynamic and static energy on partitioned multicore platform. They find the critical speed which minimizes both dynamic and static energy then proceed with a binary search algorithms to determine the optimal number of active cores needed to schedule the taskset.

For periodic task systems, Aydin et al. [126] investigate the energy-aware partitioning using the First-Fit (FF), Best-Fit (BF), Next-Fit(NF), and Worst-Fit (WF) heuristics. They prove that the energy-aware partitioning problem under real-time constraints is NP-hard. They demonstrate that the Worst Fit algorithm achieves a balanced load and consequently minimum energy consumption when the task utilizations are known *a priori* and the frequency scale is continuous. Under discrete frequency scales, the authors in [127] show that the Worst Fit algorithm no longer provides a balanced workload. Instead they propose an Adaptive Minimal Bound First-Fit (AMBFF) algorithm which uses the First-Fit algorithm to balance the schedule.

The work of [128] considers Worst Fit heuristic for task partitioning. They further investigate the number of active cores needed through three proposed algorithms: an exhaustive search algorithm, a Greedy Load Balancing (GLB) algorithm which attempts to move tasks from the least loaded core to the second least loaded core as long as feasibility is maintained and expected energy is minimized, and a threshold-based algorithm that is similar to GLB but only moves tasks whenever the least loaded core is utilized under a certain predefined threshold. The authors further use the concept of instantaneous load to dynamically change the shared frequency. That is they reclaim the slack for the purpose of further scaling down task frequencies.

The above solutions in [126, 128] balance the load based on task WCETs. Given that the tasks will terminate earlier than their respective WCETs, the load partitioning will be rendered imbalanced. To reduce temporal imbalance, the author in [129] proposes an algorithm to migrate tasks during run-time. Furthermore, they propose a dynamic core activation algorithm that varies the number of active cores at run-time.

In global multiprocessor scheduling, the authors in [130] present LRE-TL, an algorithm based on LLREF [131] that considers static voltage and frequency assignments and attempts to minimize unnecessary migrations and preemptions. The authors in [132] use mathematical optimization to solve a periodic hard real-time task scheduling problem on homogeneous multiprocessor systems with DVFS capabilities. Their formulations address tasksets with implicit, constrained and arbitrary deadlines.

2.4.3 Energy-Aware Scheduling on Heterogeneous Multiprocessors

Energy-Aware task partitioning on heterogeneous platforms is an active field of research. Literature considers various platforms which includes different-ISA multiprocessors (e.g. ARM/Alpha, ARM/Intel or ARM/FPGA), single-ISA multiprocessors with different energy profiles (e.g. ARM Cortex-A/Cortex-M), and single-ISA clustered multi-core heterogeneous SoCs (ARM big.LITTLE).

For frame-based tasks where all real-time tasks share the same deadline, the work in [133] minimizes simultaneously the taskset make-span and energy consumption. They considered the min-max heuristic for the case of a processor supporting continuous voltage levels, and integer linear programming solution when it supports discrete voltage levels. However, they assumed all heterogeneous multiprocessors support the same voltage and frequency levels which is hard to enforce in practical systems. The work in [134] undertakes a similar approach. For periodic tasks, and for a system with unique multiprocessors (a set of multiprocessors with one of each type), [135] provides an approximation scheme using a dynamic programming solution. The authors extend their work by relaxing the processor uniquness restriction and allow for multiple processors of the same type in [136].

The paper [137] considers a load balancing algorithm to allocate non real-time jobs on heterogeneous nodes. In their work, the frequency of a processing node is a function of the total cycles of all jobs assigned to it. Petrucci et al. [138] present an ILP-based thread assignment that takes input from hardware performance counters that determine the performance characteristics of the running thread. They use the instructions per second and Last Level Cache misses as a measure of CPU load and memory bandwidth. They use them to optimize global thread to core assignment. However, their approach might not be suited for real-time systems and requires solving an ILP problem periodically, thus incurring significant scheduling overhead.

For periodic real-time systems, Alhamad and Gopalakrishnan [139] employ the dynamic programming approach to minimize the total system energy under QoS requirements. Kuo and Lu [140] propose a Best-Fit Descending algorithm to allocate tasks to multiprocessors. However, they consider a non-DVFS platform consisting of a set of unique and different heterogeneous multiprocessors. Zahaf et al. [141] propose a heuristic for parallelizing and allocating real-time threads on heterogeneous ARM big.LITTLE platforms. They consider partitioned Earliest Deadline Scheduling *EDF* and select a single frequency per cluster. Awan et al. [142] map tasks onto heterogeneous platforms using Least Loss Energy Density *LLED* algorithm. In LLED, tasks have favorite processors on which they are most energy-efficient to run. When tasks can not be assigned to their favorite processors, LLED maps tasks to less energy-efficient processors based on the difference of energy between lower to higher power processors. However, they consider non-DVFS cores. They further improve their work by proposing two-phase algorithms that assign sporadic real-time and best effort tasks onto a set of distinct heterogeneous multiprocessors with DVFS support [143]. The initial phase deals with assigning processor frequencies and minimizing active energy. The second phase attempts to enhance the ability of cores to go into energy efficient sleep states.

Metaheuristics for reducing energy consumption have been claimed to produce a near-optimal solution. For a partitioned scheduling on heterogeneous multiprocessors, [144] uses modified binary particle swarm optimization and ant colony optimization. In [145], the authors formulate a heterogeneous multi-core system as a Multilevel Generalized Assignment Problem (MGAP). They use an evolutionary algorithm based on the genetic algorithm to solve the energy minimization problem for their model.

In this thesis, we consider single-ISA clustered heterogeneous multi-core processors where each cluster constitutes a single voltage frequency island, using ARM's big.LITTLE is a prime example. Few works consider this model. The work in [146] proposes three mathematical optimization formulations based on a global scheduling scheme and a fluid model. They assume both ideal systems where a processor has a continuous frequency scale, and practical systems with discrete voltage and frequency levels. Colin et al. [4] show that aggressively assigning tasks to the more efficient cores is not optimal unless these cores are "negligibly cheap". Instead, they present a load distribution algorithm that caps task allotment on energy-efficient cores and attempts to balance the workload within a cluster. A similar work in [147] introduces the Equally Worst-Fit Decreasing Algorithm (EWFD) which aims to balance the workload within each island. However, both [4, 147] neglect that tasks do not consume the same power even if they run on the same core at the same voltage/frequency. Pagani et al. [148] mitigate this simplified assumption of the task power and energy models and introduce the Heterogeneous Island- and Task-Aware Largest Task First algorithm (HIT-LTF). This algorithm maps tasks to heterogeneous clusters according to the task power at the highest frequency supported by each cluster. Then it scales down the cluster frequency as long as feasibility is maintained. The energy-minimization technique they employ depends on the remaining capacity of the most loaded core within the cluster. HIT-LTF is the most recent work that we are aware of that uses the same model and platform as the work we present in Chapter 5.

2.5 Related Work on the Use of Metaheuristics in Scheduling Problems

Literature shows that many scheduling problems are NP-Hard [124, 126, 149] and are strongly believed that they cannot be solved to optimality within a polynomially bounded computation time. Since it is inefficient to use exact algorithms for this class of problems, literature trades optimality for efficiency. These efficient algorithms seek to obtain near optimal solutions within reasonable computation time. However, they do not guarantee optimal solutions. When these algorithms make use of observations, knowledge, and assumptions pertaining to the specific problem, they are called heuristics. On the other hand, metaheuristics are a generic class of heuristic algorithms. They can be adapted with few modifications and applied to a wide set of difficult optimization problems.

Scheduling literature employed metaheurisitcs for difficult optimization problems. The work of [150, 151, 152, 153] use the genetic algorithm (GA), while [154] apply ant colony optimization. Braun et al. [155] present a comparative study between different metaheurisitc and well-known partitioning heuristics for makespan minimization. In their work, the genetic algorithm and the min-min heuristic perform well among eleven algorithms. However, none of these works consider the energy-minimization problem. For energy-reduction through frequency assignment in DVFS-capabale uniprocessors, He and Mueller [123] use the simulated annealing algorithm. For complex large chip multiprocessors with multiple pre-defined voltage islands, the authors in [156] propose an approach to static task mapping based on the *Extremal Optimization* metaheurisitc. For energy-aware partitioning of tasks onto heterogeneous multiprocessors, Zhang et al. [157] use the particle swarm optimization and show that their approach consumes 40% - 50% energy less than a genetic algorithm and shuffled frog leaping partitioning schemes.

2.6 Related Work on Constructing Real-Time Tasksets

In preparing tasksets for the evaluation of real-time scheduling algorithms, most literature adopted a methodology that starts with computing task utilizations. To achieve a desired total system utilization, efficient statistical tools generate unbiased individual task utilizations. Uniprocessor systems use the UUniFast algorithm [158]. In the multiprocessor domain, the UUniFast-Discard algorithm extends the UUniFast algorithm [159]. UUniFast-Discard basically applies UUniFast algorithm and discards any unfeasible solutions. For a certain number of tasks and desired total utilization, the high discard ratio renders the algorithm inefficient. The *randfixedsum* algorithm [160] addresses the shortcomings of UUniFast-Discard and works for any number of tasks n and total utilization U. The work of [62, 161, 162] generates task periods from statistical distributions (e.g. uniform, log-uniform, bi-modal, exponential). Once task utilizations and periods are available, Equation 2.2 computes task worst case execution times.

Recent works use the above mentioned procedure in task set generation. For example, for General Purpose and Real-Time Multi-Agent Systems (MAS), the work of [163] presents a generator of task-sets and scenarios. Their work adds configurable parameters specific to MAS, and defines lower and upper bounds for tasks timing parameters. MCRTsim [164], an open source task scheduling simulator for real-time systems with uniprocessors, multiprocessors, and multi-core processors also uses the classical approach at the core of their task set generator.

In the case where researchers want to deploy the tasksets on a hardware platform, they synthetically construct the tasks to satisfy the task WCET parameter. In one approach, [165, 166, 167, 168], each task simply burns processor cycles through timed loops until the task reaches its assigned WCET. Vulgarakis et al. [169] use a similar approach for control applications on multi-core platforms. When simulating memory and cache access behavior, Lindsay et al. [170] construct tasks using matrix operations. Each task executes as one thread and has its own address space. This approach is direct and simple; however, an underlying taskset might not capture the realistic behavior of industrial and commercial workloads.

The COBRA framework [171] generates taskset parameters and synthesizes executables based on ten benchmarking codes from TACLeBench suite. However, their approach does not place bounds on task periods. This potentially leads to high simulation intervals. Additionally, their framework runs a selected subset of the ten core benchmarks inside loops until the task reaches a value close to its desired WCET. One downside of this approach is that each task in the final taskset (regardless of the task set size) is built from the same core codes. Therefore, there is no variance in the nature of the tasks.

The authors of [15] introduce TASKers: a whole-system generator for benchmarking real-time–systems. Their approach is similar to the COBRA framework in that they build tasks from synthetic building blocks. They use an iterative approach to reach to within 0.1% of the target WCET for the process to terminate. However, the TASKers generator only targets single core processors and operating systems that comply with the OSEK standard (i.e. uses fixed-priority scheduling). The tasks are assumed to run-to-completion in the absence of asynchronous interrupts and events.

2.6.1 Performance Monitoring Units (PMUs)

Most high-end processors are equipped with specialized hardware blocks dedicated to collecting run-time performance parameters. The PMU, not to be confused with Power Management Unit, has a set of registers that are used to count processor *events*. In multicore systems, each core has its own performance monitoring unit. Most processors report the *cycles* event which counts the number of cycles that the processor has executed. However, they differ in what other events they report. Commonly, this includes cache related events such as cache miss rates, cache write-backs, and cache refills, number of branch mispredictions, number of bus accesses or memory accesses. The PMU hardware is usually non-invasive with low overhead. However, the driver firmware which controls and accesses these registers does incur slight overhead at the software or OS level.

The number of dedicated monitoring registers is usually limited compared to the number of events supported. For example, ARM Cortex-A15 PMU provides six event counters in addition to the dedicated cycles counter. The Cortex-A15 PMU supports 67 events. Meanwhile, the Cortex-A7 PMU provides four counters in addition to the cycles counter. It supports 42 events [172]. User-configurable control registers specify which events to monitor at any time. Many PMU drivers (software) provide two means to profile the system. Say one wants to profile a schedule that runs with a hyper-period $\mathbb{H}$ of 5 seconds on a Cortex-A7 processor. Given that the Cortex-A7 dedicated PMU registers can be loaded four at a time, the first approach requires running the system for 55 seconds. In each hyper-period, the control registers are configured to monitor four new events. This approach provides accurate results for a given event. However, it is susceptible to variance between system runs between hyper-periods. The second approach supports event multiplexing. In one hyper-period, the PMU driver multiplexes the 42 events and relies on interpolation to fill the gap. Despite the efficiency of this approach, it is highly unreliable.

Under Windows operating system, Intel provides a tool called Intel ® Performance Counter Monitor for reporting processor events. In Linux, the *perf* tool is one of the most commonly used profiling tools and is natively supported by many Linux distributions. It supports many architectures including x86, PowerPC64, UltraSPARC (III and IV), ARM (v5, v6, v7) among others. However, this tools has few shortcomings. The documentation is lacking and does not document all events or explain their aliases. Secondly, the tool only runs from userspace. Finally, as of the time of writing this thesis, it has minimal support for heterogeneous processors such as big.LITTLE processors (can be only active for one cluster). The PMCTrack tool addresses the shortcomings of *perf*. "PMCTrack is an open-source OS-oriented performance monitoring tool for GNU/Linux. This performance tool has been specifically designed to aid kernel developers in implementing scheduling algorithms in Linux that leverage data from performance monitoring counters (PMCs) to perform optimization at run time. Despite being an OS-oriented tool, PMCTrack still allows gathering PMC values from user space, enabling kernel developers to carry out the necessary offline analysis and debugging to assist them during the scheduler design process. [173, 174]". PMCTrack supports ARM big.LITTLE platforms and provides specific Linux patches and drivers to the Odroid-XU3/XU4 evaluation boards we use in this thesis.

Chapter 3

Evaluation of Meta-Heuristics in Energy-Aware Real-Time Scheduling

3.1 Introduction

This chapter addresses the problem of DVFS and DPM assignment in a real-time system utilizing a non-ideal DVFS-enabled single core processor and where tasks are associated with a set of system devices. The chapter presents the motivation behind the work set forth herein and lists our main contributions. It follows by presenting the system task and power models. The chapter then provides a brief background on the metaheuristics we use followed by how we adapt and configure these algorithms to solve the system-wide reduction problem. The chapter presents the simulation methodology and discuses the results and observations we obtain from simulations. The chapter concludes with a brief summary.

3.2 Chapter Motivation and Contributions

The problem of optimal DVFS and DPM assignment is NP-hard [126]. As we mentioned in Section 2.4.1, many works approach the problem with mathematical optimization and heuristics. While the majority of literature focus on reducing processor power, fewer works consider system-wide energy reduction when devices are involved. Given the complexity of the problem, appropriating metaheuristics is appealing. For example, Hu and Mueller [175] applied the Simulated Annealing metaheuristic. To the best of our knowledge, while metaheuristics have been widely used in solving different scheduling problems as we mentioned in Section 2.5, no other work employed other metaheuristics for the system-wide energy reduction problem. For this reason, we propose using evolutionary algorithms for the problem of task frequency assignment and system energy reduction. We summarize our contributions in this chapter as follows:

1. We propose using discrete versions of the genetic algorithm and the differential evolution algorithm for finding a DVFS configuration that attempts to bring the system-wide energy to a value close enough to the optimal.

2. We offer a comparative study against the Simulated Annealing metaheuristic used by Hu and Mueller [175] and the CS-DVS algorithm [176] under the same system and task models.

3. Given that evolutionary algorithms are sensitive to initial conditions, we investigate the effects of varying the values of initialization variables on producing good enough energy savings. We present our rationalizations behind selecting the values for these parameters given the constraints of the problem and the experimental platform.

The chapter organization is as follows: Section 3.3 presents the system power and task models. Section 3.4 introduces the general concepts of the genetic algorithm, differential evolution, and simulated annealing metaheuristics. It proceeds with presenting the adaptations of these algorithms towards reducing overall system energy. The chapter presents the simulation platform in Section 3.6. The analysis and discussion of results follow in Section 3.7. A brief summary concludes the chapter in Section 3.8.

3.3 System Model

In this section, we present the processor, device, and real-time task model we use throughout this chapter.

3.3.1 System Power Model

We consider a DVFS-capable processor with active, idle, and low-power states. We further consider a set of devices dev_k where each device has an active and a sleep state. For both the processor and devices, we only assume one sleep state to keep the design space exploration manageable. In the active state, the processor is capable of executing tasks at any of the supported n-discrete frequencies f_i where $f_i < f_{i+1} < \cdots < f_n$. We normalize the frequency scales λ_i corresponding to f_i according to the highest system frequency f_n as in Equation 2.5. Switching from the processor active state to the lower power state s entails switching time and energy overheads, defined as t_{SW}^{CPU} and E_{SW}^{CPU}, respectively. The switching overheads represent both the switching overheads from active-to-low power state and vice versa. The power consumed while the processor is in the active state depends on the currently selected frequency which we denote as $P_{f_i}^{CPU}$. Similarly, the power consumed while the processor is in the idle state also depends on the currently selected frequency which we denote as $P_{idle_{f_i}}^{CPU}$. We denote the power consumed while the processor is in deep sleep state s as P_{sleep}^{CPU}. Similar to the processor model, we denote the device power consumed in active and low power states as $P_{active}^{dev_k}$ and $P_{sleep}^{dev_k}$, respectively. We represent device time and energy switching overheads from the device active to low power state ($s$witching $d$own) by $t_{sd}^{dev_k}$ and $E_{sd}^{dev_k}$, respectively. Similarly, we represent device time and energy switching overheads from the device low power to active state ($s$witching up) by $t_{su}^{dev_k}$ and $E_{su}^{dev_k}$, respectively.

3.3.2 Task Model

We assume a hard real-time system which consists of a taskset Γ of N tasks τ_i where $\Gamma = \{\tau_1, \tau_2, \ldots \tau_N\}$. We assume independent and periodic tasks τ_i with implicit deadlines. We represent each task τ_i by the tuple (c_i , d_i , T_i) denoting task worst-case execution time, deadline, and period, respectively. We assume an inter-task DVFS model where we assign each task a frequency f_i to execute at. We compute the hyper-period $\mathbb{H}$ of taskset Γ according to Equation 2.1. For each task τ_i, we assign a number of unique devices dev_k. Similar to previous work [175, 177], we assume an inter-task device scheduling model where devices are available (not shared), in active mode (already powered on), and run throughout the associated task run-time (consume power). We power down devices only when the associated task terminates on condition that it is not used by the subsequent task, and only when the transition to low power mode is energy-efficient (refer to Equation 2.18). We use a fully-preemptive EDF scheduler. We only consider task schedules which are already feasible under EDF when no frequency or voltage scaling techniques are employed. We start with this assumption because if the schedule is not feasible when tasks run at the highest processor supported speed and shortest execution times, they will not be feasible under any DVFS policy. We compute task utilization and scaled utilization under DVFS according to Equation 2.2 and Equation 2.6, respectively. Under EDF, a feasible schedule must satisfy Equation 2.9. In accordance with previous research [175, 176, 178], we assume a linear relationship between task execution time and frequency.

We present few definitions that we use throughout the remainder of the chapter:

Definition 3.3.1. A DVFS configuration is a configuration that includes a permutation of

DVFS frequency assignments of dimension N corresponding to the size of taskset Γ, and includes an energy cost variable as well as a set of status flags.

The flags in a DVFS configuration convey information on the feasibility of the configuration or control decision paths within the algorithm. The feasibility flag denotes if the scaled tasks violate EDF schedulability conditions under certain frequency assignments.

Definition 3.3.2. A feasible DVFS configuration is a configuration which satisfies EDF schedulability condition when we use DVFS frequency assignments.

Definition 3.3.3. We define a good-enough solution as a solution that is within 1% of the optimal value obtained through an exhaustive approach.

Definition 3.3.4. A G_{engh} configuration is a configuration which yields a good-enough solution.

3.4 Background

In this section, we provide an abstract introduction of the metaheuristics that we use or reference in this chapter. We discuss the Genetic Algorithm (GA), Differential Evolution (DE), and Simulated Annealing (SA).

3.4.1 Genetic Algorithm (GA)

The genetic algorithm (GA) is an optimization algorithm based on the principles of genetics. It mimics the process of natural selection and the survival of the fittest. The algorithm starts with a population of random solutions that evolve through time towards the global optimum. Each member in this population is called a chromosome. Each chromosome consists of a set of variables called genes. The number of genes (i.e. length of the chromosome) corresponds to the number of variables that need to be optimized.

The genetic algorithm passes through multiple iterations (a.k.a generations). In each iteration, the genetic algorithm evaluates the cost of each chromosome within the population by applying the chromosome genes (variables) into the objective function. The objective function (a.k.a fitness function) represents the problem which we aim to minimize or maximize. The algorithm sorts the chromosomes in terms of their cost either in ascending or descending order depending if the optimum it seeks is a global minimum or maximum, respectively. In one approach, the algorithm keeps half the chromosomes that have costs closer to the optimum and discards the other half.

The chromosomes that survive are called *parent* chromosomes. The genetic algorithm uses parent chromosomes to generate the other half of the population appropriately called the offspring, or children. The genetic algorithm selects pairs of parent chromosomes either randomly, or sequentially, or any other mechanism [179]. The algorithm applies an operation called crossover at which it exchanges genes between parents at pre-selected points. Each parent pair produces two new child chromosomes, each has part of one parent, and part of the other. Consequently, the populations size remains unchanged across all generations. The genetic algorithm mimics nature by introducing the *mutation* operation. It randomly chooses a certain percentage of genes from the population pool and assigns them new values (from within the accepted range the variable can take). This procedure helps in pushing the algorithm away from converging towards a local maxima or minima. The mutated set of parent and child chromosomes constitute a new population. The genetic algorithm repeats this procedure in every iteration over new populations. The algorithm executes either up to a desired number of iterations or when it stops producing better solutions. Figure 3.1 illustrates the genetic algorithm stages. There is no guarantee that the genetic algorithm will ever give the global optimum, but it can provide near-optimal solutions.

3.4.2 Differential Evolution (DE)

Differential evolution is an optimization algorithm which belongs to the same group of evolutionary algorithms as that of the genetic algorithm. However, differential evolution is founded on stochastic principles to find a solution. Similar to the genetic algorithm, the algorithm maintains a set of solutions called candidate vectors which evolve through iterative operations. Each vector is comprised of a set of variables that it applies into the objective function to calculate the cost of a solution. The length of the vector corresponds to the number of variables that need to be optimized.

The differential evolution algorithm passes through multiple iterations. In each iteration, for each vector in the population (called base or parent vector), a new vector is created. Differential evolution generates this new new vector from the addition of a scaled difference of two different candidates to another third candidate. The set of new candidates are called donor vectors. The algorithm generates a new vector called the trial vector from each base vector and its corresponding donor vector based on a certain probability. It measures the cost of the trial vector. The trial vector replaces its corresponding base vector only if it is closer to the optimum. Otherwise, the algorithm discards it.

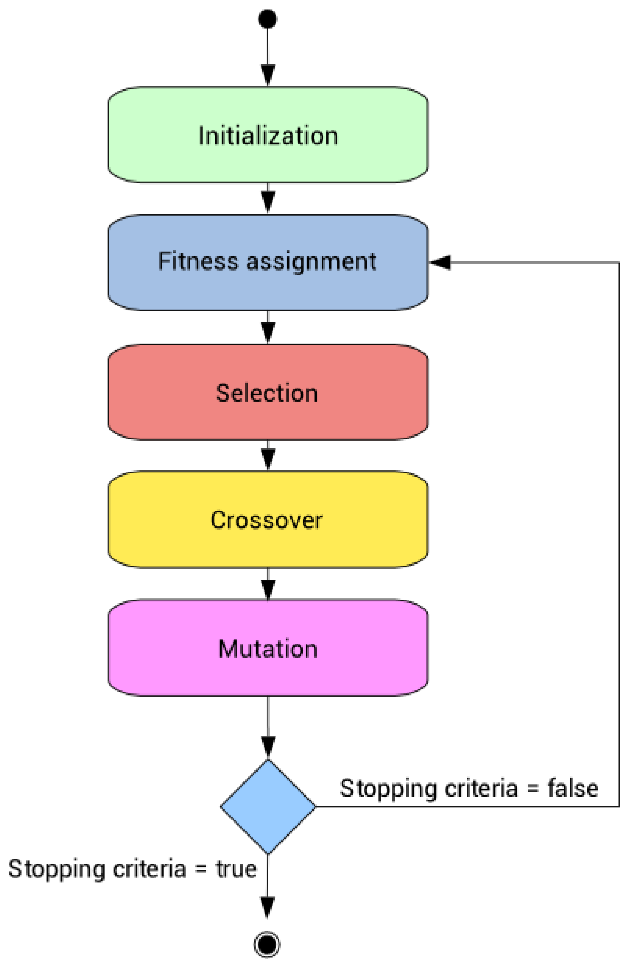


Figure 3.1 Stages of the genetic algorithm

3.4.3 Simulated Annealing (SA)

Simulated Annealing is based on an analogy to the heat-treatment of metals (a.k.a annealing). Annealing is a technique that involves the initial heating and then controlled cooling of a material to increase the size of its crystals and reduce their defects. Within this process, desirable properties possibly emerge such as hardness or flexibility. In simulated annealing optimization, when the "temperature" parameter is initially high, the algorithm has more freedom of random movement towards a solution. This allows the algorithm to sample the solution space widely. This further minimizes the possibility of getting stuck in a local optima (but not necessarily guarantees escaping it).

Simulated annealing starts with one solution and explores neighbouring solutions that move towards a global optimum. A neighbouring configuration is a configuration that differs from the current configuration by the value of one variable. A neighbouring solution that is closer to the global optimum always replaces the current solution. To avoid falling into a local minimum, the algorithm accepts a worse solution based on a certain acceptance probability. The rationale behind this is that worse solutions can potentially move the algorithm toward the global optimum.

3.5 Algorithms

In this section, we present our discrete implementation of the genetic algorithm and differential evolution for the frequency to task assignment problem. For the sake of completeness, we summarize the simulated annealing algorithm presented by Mueller [175] and the CS-DVS [112] at the end of this section.

3.5.1 Genetic Algorithm Frequency Scaling (GAFS)

Given our task and processor models, the total number of possible frequency to task permutations is n^N. The DVFS configuration comprises the chromosome that represents the frequency to task assignment to be solved. It also has a set of configuration flags. We denote the chromosome as $\mathbb{C}$ and its length of N genes corresponds to each system task τ_i. The value each gene takes is the index i of supported frequencies f_i with a possible range of $[1, n]$. This renders our algorithm a discrete integer genetic algorithm. Besides the feasibility flag and the chromosome, the DVFS configuration has two other flags. The first denotes if the chromosome is a parent chromosome. The second if any of its genes has been altered by the mutation operation. It also has a variable which holds the cost of the configuration. Algorithm 1 illustrates the steps of the GAFS algorithm.

The population pool has a size of NP DVFS configurations. To initialize the pool, we use the output of the CS-DVS algorithm to initialize the first chromosome. We randomly initialize the remainder of the $NP-1$ chromosomes by strictly feasible DVFS configurations. We set the parent flags for the initial population and reset the mutation flags. We run each initial DVFS configuration for one hyper-period in our scheduler. After the end of the simulation, the simulator returns the total energy consumed by a configuration. Finally, we store the energy consumption value into the DVFS configuration cost variable **(Lines 1-9)**. We select the lowest energy $^{NP}/_2$ chromosomes as parents and sort them in ascending order according to their cost variable. We employ a top-bottom pairing approach to pair parents from the parent pool.

We perform one point crossover operation with a minimum of 25% and a maximum of 40% of gene exchange. The location of the cross over point is fixed **(Lines 11-15)**. Once we generate the offspring, we apply a mutation operator on the whole population except for the elite chromosome. The elite chromosome is the chromosome in the DVFS configuration that holds the lowest overall system energy within the population. We use this approach to conserve the DVFS configuration with the best result. As the algorithm proceeds, the elite chromosome designation goes to any DVFS configuration which yields better energy savings. We compute the number of mutations according to Equation 3.1.

$$Mutations = \mu \times (NP - 1) \times N \tag{3.1}$$

where NP and μ are the population size and the mutation factor, respectively. The mutation flag for any parent chromosomes that has been affected by the mutation is set **(Lines 16-21)**. We test each one of the offspring and mutated parent DVFS configurations for feasibility. If it fails, then we set its cost variable to ∞ and invalidate its feasibility flag. This saves time from running unfeasible schedules inside our simulator **(Lines 22-26)**. Consequently, in the new population, we only run feasible child and modified parent configurations in subsequent hyper-periods **(Lines 27-30, 34-37)**. Since we are looking for good enough configurations, we terminate the algorithm at any of these points: we find a G_{enph} (early termination), or the algorithm generates unfeasible offspring, or it reaches the maximum specified runs **(Lines 31-33, 38-40, 10)**. In this case, the number of generations is $\lceil \frac{HP}{NP} \rceil$, where HP is the number of test hyper-periods $\mathbb{H}$.

3.5.2 Differential Evolution Frequency Scaling (DEFS)

In our adaptation of the differential evolution algorithm, a DVFS configuration comprises an N-dimensional vector v, a feasibility flag, and a cost variable. In a population of size NP configurations, we follow the same initialization approach as we did for the GAFS algorithm. We initialize one configuration with the output of the CS-DVS. We initialize the remainder $NP - 1$ configurations with random and feasible frequency scales (i.e., the index i of the frequency scale level f_i. We run each DVFS configuration of the initial population in our simulator for one hyperperiod. We store the total energy consumption of the initial configurations in the cost variable **(Lines 1-9)**.

To produce the next set of candidate configurations, for each one of the i^{th} base vector in the population, we randomly choose three different vectors v_{i_1}, v_{i_2} and v_{i_3}. We compute a donor vector t_{v_i} from these three vectors on an element-by-element basis using a scaling

Algorithm 1 GAFS

```
1: Initialization:
2: $\mathbb{C}_1 \leftarrow$ CS-DVS configuration. $\mathbb{C}_1$ is parent, feasible, and unmodified
3: iteration $\leftarrow 0$
4: for $i \leftarrow 2, NP$ do                                    ▷ NP: Population Size
5:     $\mathbb{C}_i \leftarrow$ Random and feasible frequency scales
6:     Set $\mathbb{C}_i$ as parent, feasible, and unmodified
7:     Run $\mathbb{C}_i$ in scheduler - measure and store system energy
8:     iteration $\leftarrow$ iteration + 1
9: end for
10: while iteration $< max\_hp$ do
11:     Produce Next Generation:
12:     Sort NP configurations in ascending order of total system energy consumption
13:     Pair best $\frac{NP}{2}$ configurations in top-down approach
14:     Perform one point cross over and generate offspring
15:     Set state for new configurations as child
16:     for $i \leftarrow 1, \#mutations$ do
17:         Randomly choose a gene and mutate from the new population (exclude elite chromosome)
18:         if chosen configuration is parent then
19:             Change parent state to modified
20:         end if
21:     end for
22:     for $i \leftarrow 1, NP$ do
23:         if new $\mathbb{C}_i$ is unfeasible then
24:             Set power of unfeasible configuration to $\infty$, reset feasible flag
25:         end if
26:     end for
27:     for $i \leftarrow 1, NP$ do
28:         if new $\mathbb{C}_i$ is feasible configuration then
29:             if new $\mathbb{C}_i$ is child or modified parent then
30:                 Run new configuration $\mathbb{C}_i$ in scheduler for one HP, measure total system energy
31:                 if $\mathbb{C}_i$ results in $G_{engh}$ then
32:                     Select $\mathbb{C}_i$ as solution and exit GAFS
33:                 end if
34:                 iteration $\leftarrow$ iteration + 1
35:             end if
36:         end if
37:     end for
38:     if No configuration is feasible in current generation then
39:         Terminate. Choose elite chromosome as solution
40:     end if
41: end while
```

formula that we show in Equation 3.2:

$$t_{v_i} = round(v_{i_1} + \phi \cdot (v_{i_2} - v_{i_3})) \tag{3.2}$$

where $i \neq i_1 \neq i_2 \neq i_3$ and ϕ is the vector difference scaling factor **(Line 13)**. The term difference scaling factor should not be confused with frequency scaling factors. Rounding to integer is one form of discretizing the continuous version of DE algorithm. A boundary check follows to constrain the frequency scales to fall within the supported processor frequency levels according to Equation 3.3. We apply this formula on each element of vector t_{v_i}:

$$t_{v_i}(j) = min(f_{max_i}, max(f_{min_i}, t_{v_i}(j))) \tag{3.3}$$

where j is the element index of vector t_{v_i} and $j \in [1, N]$, f_{min_i} and f_{max_i} are the lowest and highest frequency scales indices supported by the processor, respectively **(Line 14)**. Finally, the trial vector t_{v_i} is crossed over on an element by element basis with its parent v_i, the i^{th} vector of the population using Eq. 3.4:

$$u_i[j] = \begin{cases} t_{v_i}[j] & if\ r_j > CR \\ v_i[j] & otherwise \end{cases} \tag{3.4}$$

where j is the j^{th} element of vectors v_i, t_{v_i} and $j \in [1, N]$. r_j is a randomly generated number for each element j where $r_j \in [0, 1]$. CR is the crossover probability used as a control element for the differential evolution algorithm, $CR \in [0, 1]$ **(Line 15)**.

Each of the candidate vectors undergoes a schedulability check and we set its feasibility flag accordingly. If the vector is unfeasible, then we set its cost to ∞ ensuring it will never replace its parent. We only allow feasible configurations to execute in the next hyper-period **(Lines 16-21)**. We compute the total system energy with the DVFS configuration of the candidate vector. We conduct a replacement check according to: Equation 3.5:

$$v_i = \begin{cases} u_i & if\ Energy(u_i) < Energy(v_i) \\ v_i & otherwise \end{cases} \tag{3.5}$$

where $Energy(u_i)$ is total system energy of the DVFS configuration of the candidate vector during one hyper-period, and $Energy(v_i)$ is the energy of the DVFS configuration of the parent configuration. The algorithm generates candidate configurations until we reach a "good enough" configuration G_{enph} or the algorithm reaches the maximum number of iterations **(Lines 22-28)**.

Algorithm 2 DEFS

```
1: Initialization:
2: $v_1 \leftarrow$ CS-DVS configuration, set $v_1$ as feasible
3: iteration $\leftarrow 0$
4: for $i \leftarrow 2, NP$ do                                   ▷ NP: Population Size
5:     $v_i \leftarrow$ Random and feasible frequency scaling configuration
6:     Set $v_i$ as feasible
7:     Run $v_i$ in next HP - measure and store system-wide energy
8:     iteration $\leftarrow$ iteration + 1
9: end for
10: while iteration $< max\_hp$ do
11:     Produce Next Generation:
12:     for i $\leftarrow$ 1, NP do
13:         Get mutant vector $t_{v_i}$ for parent $v_i$
14:         Perform boundary checking and correction on $t_{v_i}$
15:         $u_i \leftarrow$ crossover between mutant vector $t_{v_i}$ and $v_i$
16:         if $u_i$ is feasible then
17:             Run candidate $u_i$ in next HP and measure system-wide energy
18:             iteration $\leftarrow$ iteration + 1
19:         else
20:             Set power of $u_i$ to $\infty$
21:         end if
22:         if Energy($u_i$) $<$ Energy($v_i$) then
23:             if Energy($u_i$) is within 1% of optimal then
24:                 Select $u_i$ as solution and exit DEFS
25:             end if
26:             Replace $v_i$ with new candidate configuration $u_i$
27:         end if
28:     end for
29: end while
```

3.5.3 Critical Speed – Dynamic Voltage Scaling (CS-DVS)

In this section, we go briefly over the CS-DVS algorithm [176] which is a popular heuristic for this problem. We present CS-DVS in Algorithm 3. The algorithm runs in two straightforward stages. Initially, it computes the critical speed of each system task **(Lines 1-3)**. Then, to maintain feasibility, it determines which tasks and to which supported frequency it needs to increase their frequency to have a feasible schedule while maintaining lowest system-energy. In CS-DVS, any task that is not already assigned the maximum supported frequency f_n remains within the set of tasks under consideration in each iteration. At the core of CS-DVS, it basically computes the power difference between having the task run at its current frequency and the next frequency. It only adjusts the frequency for the task with the minimum power consumption penalty. The algorithm runs as long as the schedule is unfeasible or all tasks are set to maximum speed and no feasible schedule exists.

Algorithm 3 CS-DVS [176]

```
1: Initialisation:
2: Compute critical speed of each task $\tau_i$
3: Set frequency scale $f_i$ of task $\tau_i$ to that of the critical speed of the task
4: while (DVFS configuration unfeasible) do
5:     for All tasks not running at max processor speed $f_n$ do
6:         Compute task associated power at next higher frequency scale
7:         Compute task power consumption difference between current and next
           higher frequency scale
8:     end for
9:     Choose task with lowest increase in power
10:    Set chosen task frequency scale $f_i$ to $f_{i+1}$
11: end while
```

3.5.4 Simulated Annealing Frequency Scaling (SAFS)

In this chapter, we follow the adaptation of simulated annealing meta-heuristic for system-wide energy reduction as presented in [175] with minor modifications. Similar to our proposed algorithms, SA starts with a feasible configuration J which is a vector of size N tasks. The initial vector has a DVFS configuration that matches CS-DVS output. It executes the initial configuration in one hyper-period and computes the system energy consumption. It randomly changes one frequency scale in the current configuration J to another supported scale to generate a neighbouring configuration J^*. It checks J^* for feasibility. If feasible, it runs and measures the energy cost of this neighbour configuration; otherwise, it generates another neighbour J^* from J and repeats the procedure until it finds a feasible neighbour. If the newly found neighbour configuration reduces the system energy compared to the current configuration, the better configuration replaces the current configuration J. However, if the neighbouring configuration results in more system energy consumption, it can still replace the current configuration J. However, it does so on a probabilistic basis. It generates a random probability $\rho \in [0, 1]$ and computes an acceptance probability α according to Equation 3.6

$$\alpha = e^{\frac{Energy(J)-Energy(J^*)}{K.Energy(J)}} \tag{3.6}$$

where K is the annealing factor to be decided through experimentation. If $\alpha > \rho$, then the worst solution replaces the current solution. The algorithm stops when it reaches the number of test hyperperiods or the current configuration is a G_{engh} configuration. We list the SA algorithm in Algorithm 4.

Algorithm 4 Simulated Annealing

```
 1: Initialization:
 2: Set initial configuration $J$ to the output of CS-DVS
 3: Run $J$ in next HP - measure and store system energy
 4: iteration ← iteration + 1
 5: Produce Next Neighbour:
 6: while iteration < $max\_hp$ do
 7:     Generate neighbouring configuration $J*$
 8:     if $J*$ is feasible then
 9:         Run $J$ in next HP - measure and store system energy
10:         if Energy $(J*)$ < Energy $(J)$ then
11:             $J = J*$
12:         else
13:             Generate random probability $\rho$
14:             Compute acceptance probability $\alpha$
15:             if $\alpha > \rho$ then
16:                 $J = J*$
17:             end if
18:         end if
19:         iteration ← iteration + 1
20:     end if
21: end while
```

3.6 Experimental Platform and Simulation

To analyze the performance of the proposed algorithms for frequency scaling and system-wide energy reduction, we first developed an event-driven simulator using SystemC 2.3.0 and Transaction Level Modelling (TLM). Our simulator encompasses many modules necessary for real-time scheduling. At the core of our simulator, we developed a real-time taskset generator and an EDF scheduler modules. Supporting functionality includes both run-time feasibility checks and offline schedulability analysis . We enable frequency affinities for the tasks to support per-Task DVFS. We have modules that specify the processor and device power models. We equip our scheduler with the ability to track task arrivals such that it has the necessary means to know if it should trigger a sleep signal or it is energy-inefficient to transition to low-power mode. We also equip or scheduler module with a built-in energy computation sub-module which keeps track of processor and devices energy-consumption.

Table 3.1 Intel XScale processor power model

Frequency Steps f_i (MHz)	1000	800	600	400	150
$P_{f_i}^{CPU}$ (Watt)	1.6	0.9	0.4	0.17	0.08
$P_{idle_{f_i}}^{CPU}$ (Watt)	0.260	0.222	0.186	0.159	0.064
Voltage V_i (Volts)	1.55	1.45	1.35	1.25	1.15
$E_{sw}^{CPU} = 0.5$ mJ			$t_{sw}^{CPU} = 85$ms		

Table 3.2 Devices power model

Device	$P_{active}^{dev_k}$ (W)	$P_{sleep}^{dev_k}$ (W)	$P_{su}^{dev_k}$ (W)	$P_{sd}^{dev_k}$ (W)	$t_{su}^{dev_k}$ (ms)	$t_{sd}^{dev_k}$ (ms)
Realtek Ethernet Chip	0.187	0.085	0.125	0.125	0.01	0.01
IBM Microdrive	1.3	0.1	0.5	0.5	0.12	0.12
SST Flash SST39LF020	0.125	0.001	0.05	0.05	0.001	0.001
SimpleTech Flash Card	0.225	0.02	0.1	0.1	0.002	0.002
MaxStream Wireless Module	0.75	0.005	0.1	0.1	0.04	0.04

When any taskset initialized with DVFS scales runs through the scheduler, the computed energy-costs will represent an integrated DVS-DPM cost. Finally, for each of the algorithms we present in each chapter, we build a dedicated module. The modules interact online with the scheduler and feed it feasible DVFS configurations. In return, as the scheduler runs the taskset for the entire hyper-period, it evaluates the system-wide energy consumption which represents the result of the objective function.

We use processor and device power models that are consistent with previous work [68, 175]. The processor model is based on the Intel XScale processor power profile which has five frequency levels. Intel XScale is based on armv5 architecture. We show the processor and device set power profiles in Table 3.1 and Table 3.2, respectively.

We consider small taskset Γ of size N, where $N \in [5,7,9]$. For each taskset size, we generate 500 random and unique task sets. We associate each task τ_i with a randomly assigned set of unique devices where the number of different devices per task $\in [0\ ,\ 2]$. To bound the hyper-period and simulation time, we apply the algorithm from [180] to generate periods within the range of [0.5 - 100] ms. We generate WCETs that are randomly selected to be between 2% and 40% of the task period. We evaluate each unique taskset 10 times to average the results due to the randomization of the algorithm. As such, each experiment for a taskset size N has 5000 simulations. We limit our investigation to nine tasks due to the high timing cost of running a taskset with a DVFS configuration through the scheduler for the duration of the hyper-period. This is especially true for the exhaustive search to find an optimal solution that we use as a reference in evaluating the results of our algorithms.

We select and initialize the metaheuristics parameters; namely, population size, mutation factors and crossover probabilities. Arguably, there is no clear consensus on how to set some of these parameters as they can be problem specific. Whereas literature concerning large search space favors large initial populations, we favor the approaches used in micro genetic algorithms (μGA) where they start with much smaller population size [181, 182, 183]. Our approach of starting with one population member initialized with the DVFS configuration of the CS-DVS algorithm, and using elitism to secure the best reached DVFS configuration aids in faster convergence similar to the approach taken in [184]. We consider population sizes of 8, 16, 24, and 32. General recommendations for the mutation rate is to keep it low ($\leq$ 0.05). However, this recommendation is generally given for binary genetic algorithms with large populations and large problem dimensions. Instead, we follow the recommendation in [179] and test the algorithms at the mutation rates 0.1 - 0.2. Larger mutation rates could in theory make it harder for the algorithm to converge as the algorithm will keep jumping between search points. Lower values could possibly lead to premature convergence and produce non-optimal results.

For DEFS, as in GAFS, we use the same range of small population size similar to the work of [185, 186]. We test our algorithms with crossover probabilities CR of 0.3, 0.5 and 0.7. We also choose the scaling factors from uniform probability samples in the range [0 , 1]. For the smaller taskset size of $N = 5$, we explore the effect of varying the maximum number of simulations (i.e. hyper-periods) from a set of [50, 100, 200 and 400]. We use two additional hyper-periods 1000 and 2500 for task sizes of 7 and 9. We assume the scheduling overhead to be low and therefore neglected. For SAFS, we compare our work to the version with annealing factor of 0.005 which our preliminary tests showed it was the best factor.

3.7 Results and Discussion

In this section, we report the effects of different configuration parameters on the performance of the evolutionary algorithms we proposed in the last section. Initially, we show the reference energy savings from DVFS assignments according to the CS-DVS heuristic and optimal search in Figure 3.2. Given that we run our algorithms over 500 unique tasksets for each taskset size and repeat each 10 times, we compare the performance of our algorithms to the reference algorithms as follows: first by reporting the percentage of the 5000 simulations that provided results within 1% of the optimal, and secondly by the percentage of simulations

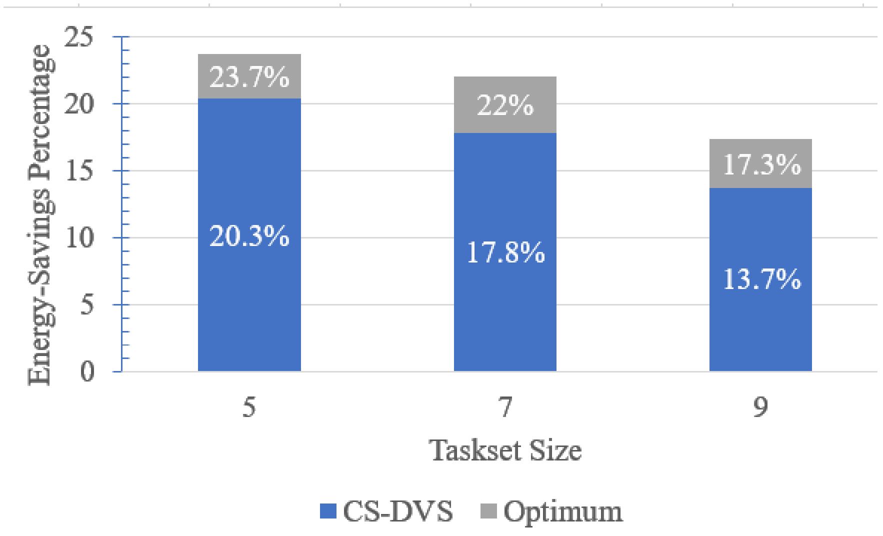


Figure 3.2 Average CS-DVS and optimal DVFS configuration energy savings over 500 unique sets

that provided configurations that performed better than CS-DVS.

3.7.1 Sensitivity Analysis

In the search for the best values for our parameters to yield better results, we fix the values of parameters that are not under investigation. Our experiments include varying the number of maximum hyperperiod iterations for which the algorithm is simulated. We also vary mutation rates (in GAFS) and crossover probabilities and scaling factors (in DEFS) and report our findings. The results in Table 3.3 through Table 3.6 show the cases where one variable is studied, while the others are fixed at the values which gave the best overall results.

We show the effects of running the algorithms over more hyper-periods (generations) in Table 3.3 for both GAFS and DEFS algorithms. We expect that longer simulations time will yield higher percentage of DVFS configurations that are G_{engh} configurations. For the smaller taskset size of $N = 5$, GAFS performs better than DEFS. However, for $N = 7$ and $N = 9$, DEFS performs better than GAFS. In all cases, 85% of DVFS-configurations can reach energy-reduction that is 1% near the optimal. This is important given the very small search space that the algorithm has gone through.

In the next step, we conduct analysis on varying the initial population size. That is, we change the number of initial feasible chromosomes for the GAFS algorithm, as well as the initial candidate vector pool for DEFS. In Table 3.4, we observe that larger population sizes in GAFS yield better results with a clear margin for larger task sets. However, the effects

Table 3.3 GAFS and DEFS sensitivity to HP represented by the percentage of G_{engh} configurations over 5000 simulations

Tasks	Algorithm	Hyper-period					
N		50	100	200	400	1000	2500
5	GAFS	22.9%	36.2%	67.7%	84.6%	-	-
	DEFS	19.5%	22.6%	33.4%	67.1%	-	-
7	GAFS	13.2%	17.7%	32.8%	57.5%	70.1%	78.0%
	DEFS	12.2%	13.1%	17.1%	36.0%	70.9%	82.8%
9	GAFS	13.9%	17.0%	27.0%	46.9%	68.6%	85.3%
	DEFS	12.5%	13.1%	16.5%	28.8%	62.4%	86.5%

Notes: GAFS: population size = 32, $\mu = 0.1$
DEFS: population size = 24, CR = 0.3 and $\mu = 0.5$

Table 3.4 GAFS sensitivity to population size represented by the percentage of G_{engh} configurations over 5000 simulations

Task No.	HP	8	16	24	32
Percentage of G_{engh} configurations					
T5	400	78.3%	82.1%	84.6%	84.6%
T7	2500	67.7%	75.3%	78.0%	78.0%
T9		77.1%	82.4%	84.6%	85.3%
Better than CS-DVS					
T5	400	94.8%	96.7%	97.6%	98.0%
T7	2500	95.8%	97.5%	98.2%	98.0%
T9		97.4%	98.3%	98.4%	98.7%

Notes: $\mu = 0.1$

of population size diminishes between the NP = 24 and NP = 32.

Table 3.5 shows energy savings sensitivity to GAFS mutation rates and task set size when the population size is fixed at 32. We observe that a lower μ= 0.1 gives overall better results and only in a few cases that $\mu = 0.2$ results in marginal gains. Given the population size and the chromosome size of our problem, we expected the lower mutation rate μ= 0.1 to give better results. Higher mutation rates would entail exploring further away from our current best results. As μ increases, the closer the genetic algorithm gets to a random search. We observe that GAFS outperforms CS-DVS in most cases, especially with larger task set sizes.

Table 3.6 shows the results of varying the population size for the DEFS algorithm when CR = 0.3 and the mutation rate (scaling factor) $\phi = 0.5$. We see that a population size of 16 provides slightly better results for the small task set of size $N = 5$; whereas a population size of 24 gives better results for larger task sets $N = 7$ and $N = 9$. Similar to GAFS, large population sizes allow for richer selection of candidate vectors, as well as for more variance in the crossover operations.

We summarize the sensitivity analysis findings of DEFS crossover probability (CR) and

Table 3.5 GAFS sensitivity to mutation factors represented by the percentage of G_{engh} configurations over 5000 simulations

		Percentage of G_{engh} configurations		Percentage of configurations better than CS-DVS	
Task No.	HP	$\mu = 0.1$	$\mu = 0.2$	$\mu = 0.1$	$\mu = 0.2$
T5	400	84.6%	78.6%	94.9%	95.7%
T7	2500	78.0%	81.8%	98.0%	99.0%
T9		85.3%	73.2%	98.7%	97.3%
Notes: population size = 32					

Table 3.6 DEFS sensitivity to population size represented by the percentage of G_{engh} configurations over 5000 simulations

		Percentage of G_{engh} configurations saving configurations			Percentage of configurations better than CS-DVS		
Tasks	HP	Population Size					
		8	16	24	8	16	24
T5		71.2%	80.4%	67.1%	90.5%	96.7%	94.6%
T7	2500	34.3%	77.9%	82.8%	86.8%	97.4%	98.8%
T9		40.6%	81.4%	86.5%	88.1%	98%	98.6%
Notes: CR = 0.3 and $\phi = 0.5$							

mutation (scaling) factor ϕ parameters in Table 3.7. For larger task sizes of 7 and 9, we find that crossover probability and mutation factor carry no statistical differences in yielding better results across different combinations of CR and ϕ. However, for smaller task sizes, a CR of 0.3 and $\phi = 0.5$ provide better results by a wide margin (i.e., up to 14% better results than those at CR = 0.7 and $\phi = 0.7$ for a system with five tasks). For task sizes of $N = 7$ and $N = 9$, scaling factors of 0.5 and 0.7 provide slightly more G_{engh} configurations in general across all CR factors. However, the effects of CR factors are less pronounced and within about 1.5% of each other.

We summarize the reference values for the simulated annealing (SA) algorithm implementation in Table 3.8 and Table 3.9. One major observation is that the results of the SA algorithm do not provide substantial gains as the number of hyper-periods is increased when it comes to G_{engh} results. This is more obvious at the larger task set size of $N = 9$. In fact, lower number of hyper-periods could provide better results. This is due to the algorithm design as implemented by [175], where even though the algorithm can escape a local minimum, there is no guarantee that it will converge to a better solution.

Table 3.7 DEFS sensitivity to crossover and scaling factor ϕ represented by the percentage of G_{engh} configurations over 5000 simulations

Population Size		16	24	
Hyper-period		400	2500	
CR	ϕ	5	7	9
0.3	0.3	76.1%	77.1%	81.8%
	0.5	80.4%	82.8%	86.5%
	0.7	78.2%	83.3%	85.6%
0.5	0.3	75.2%	80.7%	84.9%
	0.5	78.2%	84%	87%
	0.7	76.3%	83.7%	87.0%
0.7	0.3	68.0%	80.1%	83.8%
	0.5	69.2%	82.5%	84.8%
	0.7	66.4%	82.6%	83.9%

Table 3.8 SAFS sensitivity to the number of hyperperiods represented by the percentage of G_{engh} configurations over 5000 simulations

N	Hyperperiod					
	50	100	200	400	1000	2500
5	21.5%	23.3%	26.5%	32.3%	-	-
7	13.9%	14.7%	15.6%	17.3%	19.6%	31.5%
9	20.6%	23.0%	25.3%	27.9%	17.3%	25.6%

3.7.2 Algorithm Comparison and Discussion

Figure 3.3 provides a performance summary of our algorithms and the SAFS reference algorithm. We observe that both GAFS and DEFS outperform the SAFS algorithm in terms of their ability to consistently provide G_{engh} configurations. As the task set size increases, the performance of SAFS decreases, while GAFS and DEFS consistently maintain their performance. Figure 3.3 shows that the SAFS algorithm fails on average 25% of the time to yield DVFS configurations better than CS-DVS across all taskset sizes. In effect, this could lead to high system-wide energy consumption. Both GAFS and DEFS are superior to SAFS in this regard, as they almost always deliver better DVFS configurations than CS-DVS where 83.5% of the time, these configurations are G_{engh} configurations.

GAFS slightly outperforms DEFS for task sets with $N = 5$ and the converse is true for larger task sets. The weakness point of GAFS is that a new generation of test DVFS configurations can only be generated when the current population has been fully examined. DEFS does not suffer from this issue, as we generate candidate DVFS configurations randomly from a list of the so-far best found DVFS configurations that are readily available. The SAFS algorithm suffers from the possibility of replacing an elite solution by a non-optimal one.

Table 3.9 Percentage of SAFS DVFS configurations that yield more energy reductions than CS-DVS over 5000 simulations

N	Hyperperiod					
	50	100	200	400	1000	2500
5	54.7%	60.8%	65.9%	72.8%	-	-
7	48.1%	54.4%	59.8%	66.5%	70.3%	78.6%
9	40.5%	44.9%	48.1%	52.0%	63.4%	72.6%

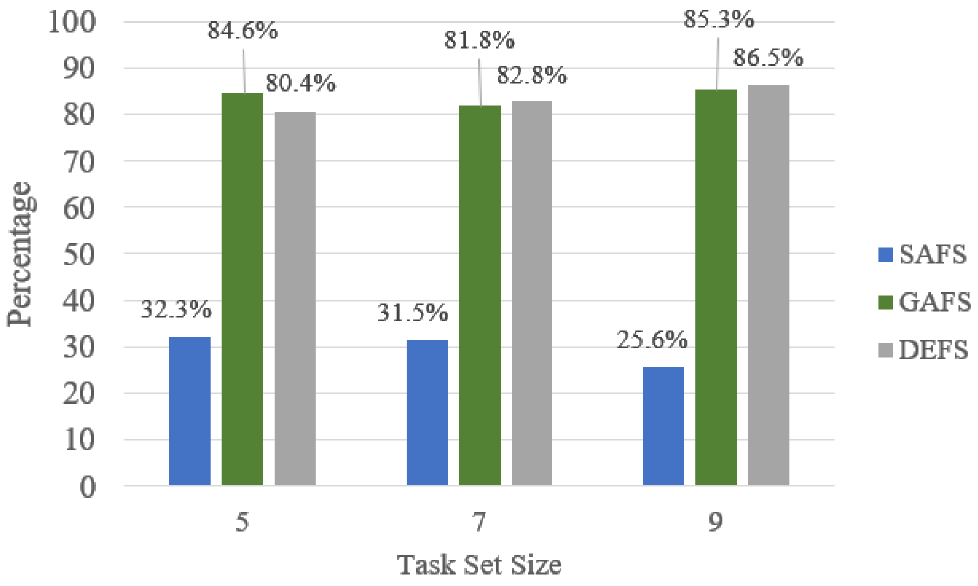


Figure 3.3 Percentage of G_{engh} configurations of the three meta-heuristics over 500 unique sets repeated 10 times

This is due to the inherent design of the algorithm, where it stochastically accepts worse solutions as means to escape a local minimum.

Even though GAFS maintains an elite solution through the generations (which only gets updated if better solution is found), GAFS suffers from the possibility of producing a whole generation of non-feasible solutions aside from the elite. DEFS, on the other hand, does not suffer from these issues as it maintains a population of best found feasible solutions at any time.

3.8 Chapter Summary

System-wide energy minimization is of paramount importance in modern embedded system design. We specifically adapted the use of genetic (GAFS) and evolutionary (DEFS) algorithms with the goal of reducing the overall energy consumption. We have investigated the

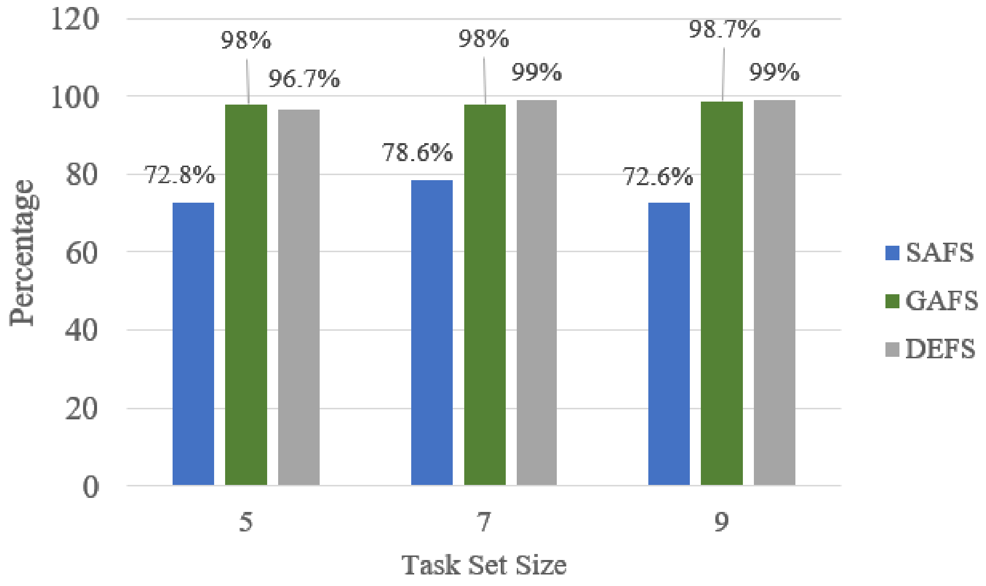


Figure 3.4 Percentage of the three meta-heuristics DVFS configurations that are better than the CS-DVS heuristic over 500 unique sets repeated 10 times

performance of our developed meta-heuristic algorithms that assign frequency scales to tasks in a hard-real-time system. We measure energy consumption at the system level; that is that of the processor and the devices. We have conducted a sensitivity analysis over a range of initial values of the proposed algorithms. We have found that within a very small search space guided by an initial solution of CS-DVS and the concept of preserving elite solutions, the algorithms were able to find a high number of G_{engh} configurations.

Our algorithms outperformed the simulating annealing (SAFS) algorithm by an average factor of 2.82 to 1 for finding a G_{engh} configuration when the system task set is comprised of 5 to 9 tasks. Furthermore, based on 500 unique sets of tasks, our algorithms deliver G_{engh} configurations in over 95% of the cases compared to the CS-DVS algorithm. Simulated annealing is better than CS-DVS by an average of 75% of the time. The proposed techniques allow for energy optimizations in small low-power embedded systems.

Chapter 4

A Methodology for Constructing Tasksets for Evaluation on Embedded Hardware

4.1 Introduction

To evaluate the performance of scheduling algorithms, most of the literature has relied on simulators that have *a priori* knowledge of the processor model. These processor models are often simplified due to the inherent complexity of the hardware or lack of disclosure of design and timing specifications. Moreover, the simulation faithfulness in representing real-world applications is dependent on the taskset properties. A taskset represented by the timing parameters of its individual tasks is sufficient to test the feasibility of scheduling algorithms. However, representing tasksets with only timing parameters offers no insights into the instruction mix of each task, and consequently its cache access behavior, I/O blocking or interaction with other tasks in the system. These insights are consequential to the assessment of reliability and energy-efficiency of real-time algorithms. For example, the work of Saha and Ravindran [168] revealed numerous inconsistencies between reported energy savings in literature simulations and those carried on real hardware platforms. However, to the best of our knowledge, there exists no standard methodology to port simulations from software onto real hardware to gain real insights into the performance and efficiency of the proposed algorithms in literature. Recent frameworks synthesize tasks from time loops, matrix operations or functional code blocks to meet a certain WCET. These approaches have limitations, first all tasks are composed of the same functional blocks or loops and therefore are repetitive in nature. Secondly, some tools are restricted by their ability to generate tasks for single core deterministic platforms where they can build on this determinism to synthesize tasks that meet a desired WCET. Finally, they offer no flexibility of adding any in-house developed tasks to the generated taskset. In this chapter, we try to address these limitations by offering a generic approach that allows the research community to run real-time tasks on real hardware. We also present and compare between algorithms that solve an arising issue related to pairing the real-time properties of the taskset.

4.2 Chapter Contributions

The contributions of this chapter are as follows:

- This chapter presents a methodology that facilitates evaluating real-time systems on real hardware and allows for testing the system at various load points, an important consideration in real-time research. Testing real-time algorithms directly on hardware will provide valuable insights and overcome the limitations of simulator-based platforms and any model simplifications.

- In our methodology, we do not synthesize tasks as most taskset generators do. Instead, we offer the ability of using tasks from publicly or commercially available embedded benchmarks, or any "in-house" programs that the research community develops. The advantages of this approach is the flexibility of choice and that tasks differ functionally from each other. Moreover, the tasks are not restricted to any platform in order to behave in certain desired way.

- In this work, the task WCET must be estimated, and once known, be paired with a task utilization and period to meet the total desired system load. This chapter shows that any **direct** pairing approach between elements of the estimated WCETs and discrete periods sets (using exhaustive approaches) will result in a total utilization that diverges from the desired utilization.

- This chapter proposes a set of feedback-based algorithms that pair the estimated WCETs with discrete bounded or unbounded periods. The algorithms produce tasksets whose total utilization converges to the target utilization with minimal errors. The chapter focuses on bounded periods because they provide reasonable simulation times. The chapter also analyzes the percentage of unique periods assigned in every taskset.

The remainder of the chapter is organized as follows: we present the system model in Section 4.3. We provide a background on the bounds on simulation time, available benchmark suites, and a summary of techniques for estimating WCET in Section 4.4. Our algorithms follow in Section 4.5. We detail the experimental setup in Section 4.6. A detailed discussion of the results of our algorithms is presented in Section 4.7. We conclude with a brief overview of the methodology and results in Section 4.8.

4.3 System Model

We assume a real-time system that runs on either a unicore processor or a multicore processor. The system runs a set of tasks $\tau = \{\tau_1, \tau_2, ... \tau_n\}$. We represent each task τ_i as a tuple $< c_i, T_i >$ where c_i is the WCET for task τ_i and T_i its associated period (for periodic systems), or minimum inter-arrival time (for sporadic systems). Each task might have an offset o_i at which it starts. Each task has a utilization as described in Equation 2.2 and the system has a total utilization as described by Equation 2.3. Each one of these tasks represents one application selected from either publicly or commercially available benchmarks sets or any developed "in-house" codes. Task WCETs are estimated by any convenient means as described in Section 2.1.2. We emphasize that the algorithms we present in this chapter are independent from any task WCET estimation technique employed. That is, WCET estimation is a preliminary step used to generate a vector of task WCETs estimates as inputs to the algorithms presented in this chapter. The periods can be discrete unbounded periods taken from a uniform or log-uniform range, or discrete bounded periods. Given that unbounded periods result in longer simulation intervals, we focus in this chapter on bounded periods. Yet, we also emphasize that our algorithms are independent from the technique used to generate the periods. That is, the periods vector is generated in a preliminary step and used as input to the algorithms presented in this chapter. This chapter uses the *randfixedsum* algorithm [160] to generate task utilizations. In this chapter, we use capital letters to denote a complete set/vector/matrix while the small letter notation denotes individual set/vector/matrix elements. So if A is a vector, a_i denotes a vector element. If A is a matrix, a_{ij} denotes a matrix element. One exception is the notation for the task periods where T_i or T_{ij} represent individual task periods to maintain notation consistency throughout the thesis. This is because the small notation t is universally acknowledged to represent time.

4.4 Background

In this section, we present a brief background on the techniques used in bounding task periods that we use in this chapter with minor modification. We also present some of the well-known embedded benchmarking suits that we rely on in constructing the tasksets.

4.4.1 Bounds on Simulation Time

Comprehensive and faithful modelling and simulation of embedded systems are necessary for evaluating system performance, reliability, power metrics, and assessing the feasibility of scheduling algorithms. It is therefore necessary to run the simulation for an interval that captures the properties and behavior of the system in a representative manner. This gives credence to the results of the system assessment. Numerous works have explored the periodicity of the time schedule for different task and processor models running various scheduling algorithms. The work of Goossens et al. [187] provides a summary of some of these time bounds. The scheduling pattern is cyclic after the established bound on the time schedule. As such, the bound on the time schedule is used as the bound on simulation interval. These bounds depend highly on the hyper-period $\mathbb{H}$ of the taskset under consideration. For example, the work of [188, 189] shows that the upper bound for the simulation interval for fixed-priority schedulers running independent tasks with either arbitrary or implicit deadlines is $O^{max} + 2\mathbb{H}$ where O^{max} is the largest offset of any task in the system. This result holds true for both uni-processor and partitioned-scheduling multiprocessors. More recent works explore the time periodicity for global schedulers on multiprocessor platforms [190, 191, 192]. Consequently, arbitrary task periods lead to significantly large hyper-periods and undesirable large simulation times.

4.4.2 Approaches to Bounding the Simulation Interval

One popular approach to minimize simulation times is the assignment of harmonic periods [193, 194, 195, 196, 197].

Definition 4.4.1. A set of periods is considered harmonic if any two pairs in the set divide each other. In mathematical terms, for a set of periods $T = \{T_1, T_2, ... T_n\}$, for every T_i and T_j and $(i \neq j)$, either ${}^{T_i}/_{T_j} \in \mathbb{N}$, or ${}^{T_j}/_{T_i} \in \mathbb{N}$.

Harmonic periods simplify schedulability analysis and naturally limit the simulation duration. Despite the apparent advantages of harmonic periods, the available range of periods to select from is limited. In a practical implementation, this will force some tasks to run unnecessarily at shorter successive intervals simply to maintain the harmonic relationship among the periods.

Other approaches minimize the hyper-period by building a sorted list of reference periods which are multiples of a selected set of prime numbers [198, 199]. The list is capped at a user defined maximum. Original task periods are adjusted to the closest reference period. Numerous recent works use this approach that is based on integer factorization and base prime numbers [141, 200, 201, 202, 203, 204]. In this work, we adopt this approach by building a period set from the factors of highly composite numbers (HCN) (a.k.a. an anti-prime). We set the hyper-period $\mathbb{H}$ to equal the HCN.

Definition 4.4.2. A Highly Composite Number (HCN) is defined as a positive integer with more divisors than any smaller positive integer: $divisors(n) > divisors(m) \quad \forall\ m < n, \quad m, n \in \mathbb{N}$. For a number to be highly composite, it has to have prime factors as small as possible, but not too many of the same.

As an example, choosing the 31st HCN for a hyper-period of 166,320 yields 160 unique periods. In comparison, using the technique in [199] for a hyper-period of 155,520 built with factors $2^7, 3^5, 5^1$ generates a total of 96 unique periods. We list the set of HCNs used in this chapter in Table 4.1. We provide the rationale behind selecting these HCNs in Section 4.6.3.

Table 4.1 A subset of Highly Composite Numbers (HCN)

HCN order	HCN	Factors	No. of Periods	Used in this chapter
30^{th}	110,880	$2^5.3^2.5.7.11$	144	No (Chapter 5)
31^{st}	166,320	$2^4.3^3.5.7.11$	160	Yes
32^{nd}	221,760	$2^6.3^2.5.7.11$	168	Yes
33^{rd}	277,200	$2^4.3^2.5^2.7.11$	180	Yes
34^{th}	332,640	$2^5.3^3.5.7.11$	192	Yes

4.4.3 Embedded Workloads

When evaluating embedded hardware, the literature uses existing benchmark suites. MiBench [205], EEMBC [206], BEEBS [207], and the IP-free automotive benchmarks [208] are some examples. Of these, MiBench remains one of the mostly widely used. MiBench includes benchmarks that represent realistic workloads from automotive, networking, security, and consumer embedded applications. For example, the *sha* benchmark is a secure hashing algorithm used for fingerprinting and data verification. The *2dfir* is used in digital imaging while *fdct* is used in video decoding in embedded consumer devices. While building their benchmarks suite that is geared towards energy measurements, [207] analyzed a wide range of benchmarks suites. Their evaluation criteria included the benchmark memory footprint, portability to bare-metal embedded platforms, and their applicability in a real embedded system. They assigned a suitability rating for each benchmark ranging from *very low* to *very high*. The suitable benchmarks were collected under the BEEBS suite. In this chapter, we prioritize and select benchmarks with suitability between *medium* and *very high* from MiBench and BEEBS, whenever possible. Users can also add their own production codes that can possibly make use of embedded SoC components to either create more realistic tasks or provide tasks with desirable properties pertaining to their research.

4.5 Proposed Algorithms to Pair Real-Time Tasksets Parameters

In this section, we present a reference algorithm and our CPA family of algorithms that pair actual real-time tasks (i.e. compiled binaries) with real-time task properties. All algorithms take as input vectors of estimated WCETs, periods (either bounded or unbounded), and a set of utilizations that add up to a total system load. Our algorithms do not generate any of the above real-time task properties but merely handles them and pairs them together to generate a taskset that can be run on real hardware. Generating the three input vectors is a preliminary step that can take advantage of future advancements by the research community. Given that estimating WCETs for real-time tasks is an open challenge (in particular for multicores), we note and emphasize that our pairing algorithms are independent from the technique employed in estimating WCETs. In Section 4.6.2, we discuss one recent approach in WCET estimation for multicore platforms and the rationale behind using it in the experimental part in this chapter. We use the *randfixedsum* algorithm [160] to generate random individual task utilizations vector U that adds up to a desired total system utilization. Finally, given that these algorithms aim to produce tasksets to facilitate simulation on real hardware, we do not impose certain restrictions or requirements on the tasks besides having a feasible taskset.

Algorithm 5 (PU): Permuted Utilizations

```
1: Input:
2: H: Vector of discrete periods in ascending order(size $m$)
3: C: Vector of estimated WCETs (size $N$, $N < m$)
4: U: Vector of task utilizations (by randfixedsum algorithm) which sum up to a desired total
   utilization $\mathbb{U}$ (size $N$)
5: Output:
6: $\mathbb{O}$: Vector of $N$ pairs pairs $(c, T)$ (WCET, period)
7: BEGIN:
8: $\mathbb{O} \leftarrow \{\}$
9: $\mathbb{E} \leftarrow \{\}$          ▷ Vector of final utilization errors corresponding to each total utilization
10: Matrix $V$ is all permutations of $U$, size of V is $(l, N)$, where $l \leftarrow N!$
11: for $i \leftarrow 1, l$ do          ▷ Compute periods for each possible permutation
12:     for $j \leftarrow 1, |C|$ do
13:         $T_{ij} \leftarrow c_j/v_{ij}$
14:         $T_{ij} \leftarrow h_k$ for which $|h_k - T_{ij}|$ is minimal $\forall h_k \in H,\ k \in [1, m]$     ▷ find closest period
                                                                  from $H$ to the computed period $T_{ij}$
15:         $\tilde{u}_{ij} \leftarrow c_j/T_{ij}$          ▷ Recompute task utilization
16:     end for
17: end for
18: for $i \leftarrow 1, l$ do
19:     $s_i \leftarrow \sum_{j \leftarrow 1}^{|U|} \tilde{u}_{ij}$          ▷ Compute new total utilization for each permutation
20:     $e_i \leftarrow |(s_i - \mathbb{U})/\mathbb{U}|$     ▷ Error between final and initial utilization, $e_i$ is the $i^{th}$ element in $\mathbb{E}$
21: end for
22: Retrieve $i$ for $min(\mathbb{E})$  ▷ Retrieve index of the permutation that results in the minimum error
23: $\mathbb{O} = \{(c_1, T_{i,1}), (c_2, T_{i,2}),\ \ldots\ (c_n, T_{i,n})\}$
```

4.5.1 Permuted Utilization Algorithm - An Exhaustive Approach

The Permuted Utilization (PU) algorithm utilizes a brute force technique. We outline the pseudo-code of PU in Algorithm 5. Given the individual task utilizations vector U of size N corresponding to the number of tasks in the system, the total possible permutations of utilization assignments is the factorial of the number of system tasks $N!$. We generate all possible orderings of the individual task utilizations. Each ordering denotes a possible mapping to the tasks. We store the resulting permutation in Matrix V where each row corresponds to one mapping (**Step 10**). Given the permuted utilization matrix V and the input WCETs vector C, we compute the initial task periods by applying Equation 2.2 on each of the elements of V. We store the results in the period matrix T (**Step 13**). We replace the periods by their closest periods from the discrete period set H (**Step 14**). Now that the periods have changed, the individual task utilizations have changed as well. We compute the new utilizations and store them into Matrix $\tilde{U}$ **(Step 15)**. The vector S

represent the new total system utilization for each permutation (row) in $\tilde{U}$ (**Step 19**). We compute the relative difference error between the target system utilization and the final computed utilization (**Step 20**). We look for and select the task utilization permutation that when paired with the WCET vector results in the minimal error (**Step 21**). We pair the permutation of task utilizations with the task WCETs and store the result in the output vector O (**Step 23**). If we require multiple tasksets for a simulation at a given target system utilization, we use the *randfixedsum* algorithm to generate a new utilizations vector and apply the PU algorithm again for each required taskset.

4.5.2 Compute Propagate and Adjust Algorithms (CPA)

The CPA algorithm is a heuristic which pairs task WCETs with a set of discrete periods. At its core, it continuously modifies individual task utilizations such that the WCET and discrete period pair results in negligible or zero errors. The modifications has yet to satisfy the target total system load and be as close as possible to the original individual utilization. A feedback step provides the basis for successive adjustments. We define few supporting functions in Algorithm 6. These supporting functions return the minimum or maximum value and the associated index in a vector or an array. We leave the implementation of these supporting functions to the reader as they are quite common. We list the pseudo-code of the CPA algorithm in Algorithm 7 and we explain the algorithm through an example in subsection 4.5.3.

The CPA algorithm takes in a vector of discrete periods H, a WCETs vector C, and an equal-size vector of task utilizations U that is randomly generated by the *randfixedsum* algorithm. The ordering of the utilizations vector corresponds to a variant of the CPA algorithm. We investigate ordering the utilizations in either ascending **(CPA-AU)**, random **(CPA-RU)**, or descending order **(CPA-DU)**.

The CPA algorithm starts the *Compute Phase* by calling the "compute function" **(Step 11)**. This function computes the initial period matrix T per Equation 2.2. That is, we divide each task's WCET c_i by each element in the utilization vector u_j to get each element period T_{ij}. We replace each resulting period T_{ij} by the closest period $h \in H$ to form a new matrix $\tilde{T}$. We derive the error (deviation) matrix $\mathbb{E}$ between the initial periods T and $\tilde{T}$ on an element by element basis. The function returns the calculated errors e_{ij} and the new periods $\tilde{T}_{ij}$ in the matrices $\tilde{T}$ and $\mathbb{E}$, respectively **(Step 34-42)**.

We traverse matrix $\mathbb{E}$ column-by-column to find the WCET/utilization pair with the least error e_{min} in each column. We save the value of the least error e_{min} into the vector

ϵ. We save the row index i_{min} of the least error element into vector I. We save the matrix indices of the least error element into vector V. However, as we move forward to determine $min(e_{ij})$ in the next columns, we exclude elements of rows i stored in vector I from being considered as the column minimum **(Steps 12-17)**.

We locate the matrix indices (i, j) for the entry in the error vector ϵ that has the largest error e_{max} **(Steps 18-19)**. The indices represent the selected WCET and period pair. We add the pair (c_i, $\tilde{T}_{ij}$) to the output vector O **(Step 20)**. Though it might seem counter-intuitive at first glance to choose the pair with the highest error, the rationale behind this selection is straightforward. We eliminate this error completely in the propagate and adjust phase. We proceed by computing the new utilization $\tilde{u}_{ij}$ (based on the assigned discrete period) **(Step 21)** before excluding the pair from further consideration for the remainder of the algorithm **(Step 22)**. We measure the difference δ between the old and the new utilization for the selected pair **(Step 23)**.

When we change utilization u_j to equal the new utilization $\tilde{u}_{ij}$, we effectively set the error to zero. Yet, to keep close to the target system load, we must distribute the utilization difference δ to one of the remaining utilizations in U. We start by making a copy of the remaining utilizations U into a second copy $\ddot{U}$ and adjust every $\ddot{u}_j \in \ddot{U}$ by δ **(Steps 24-25)**. Any negative utilization $\ddot{u}_j$ is out of range and not considered in the next steps **(Steps 26-28)**. We recompute the $T, \tilde{T}_{ij}$ and $\mathbb{E}$ matrices in a similar fashion to the compute phase with the main difference of using the adjusted utilization matrix $\ddot{U}$ instead of U. In the last step, the adjusted utilization $\ddot{u}_j$ which yields the minimum error in the error matrix $\mathbb{E}$ replaces its counterpart u_j in the vector U **(Steps 29-31)**.

In each iteration, as we pair WCETs and periods, we remove the WCET and associated utilization from the vectors C and U. The algorithm runs until we process all utilizations in U and no tasks remain.

4.5.3 A Working example of CPA-AU

To help visualize the algorithm, we present a simple numeric example for the CPA-AU algorithm. Table 4.2 shows a step by step output of the algorithm for the first iteration. The first input to the CPA-AU algorithm is a 4-element vector of WCETs C with values (20, 637, 621, 6). For a total utilization of 75%, a random utilization vector U could be (0.0912, 0.1456, 0.157, 0.3562) which we sort in ascending order. In this example, we assume a bounded period set. We generate all periods from the factors of the HCN 166,320 and sort them into H in ascending order.

Algorithm 6 Supporting functions

```
1: function $\{i, x\} \leftarrow$ MIN1 $(X)$
2:     function that gets vector $X$, and returns the index $i$ of the minimum number and the
3:     minimum number $x$ in $X$, respectively.
4: end function

5: function $\{i, x\} \leftarrow$ MAX1 $(X)$
6:     function that gets vector $X$, and returns the index $i$ of the maximum number and the
7:     maximum number $x$ in $X$, respectively.
8: end function

9: function $\{i, j, x\} \leftarrow$ MIN2 $(X)$
10:     function that gets matrix $X$, and returns the indices $i, j$ of the minimum number and the
11:     minimum number $x$ in $X$, respectively.
12: end function
```

We compute the periods matrix T. *We note that the values in Table 4.2 are rounded from the actual data to fit in the table, so rounding errors are expected when you follow this example.* We select the closest periods in H to the computed periods and store them into matrix $\tilde{T}$. We proceed to calculate the relative errors between T and $\tilde{T}$ and store the results into matrix $\mathbb{E}$ **(Steps 11, 34-42)**.

We find the minimum error e_{min} of the first column and store its corresponding $\mathbb{E}$ matrix indices, in this case the first row and first column. To break a tie, we choose the lowest ordered rows. Now, to find e_{min} for the second column, we exclude the first row as its index has been used before. This results in bypassing the value 1.7175% as a minimum and reporting 1.9226% instead. Similarly, for the third column, we choose the minimum from the remaining second and third rows (0.1050%) leaving us with 3.3409% for the last column **(Steps 12-17)**. Given that the fourth entry has the highest error in the vector of minimum errors e_{min} **(Step 19)**, we pair the period 1848 with the WCET 637 **(Step 20)**. The new utilization of this pair is 0.3447. This differs from the old utilization 0.3562 by $\delta = -0.0115$ **(Step 21)**. We exclude the (637, 1848) pair from further consideration **(Step 22)**.

A decrease of the utilization from 0.3562 to 0.3447 entails that one of the remaining utilizations should be increased by the difference. We elect to add this difference to the utilization for which this change has minimum impact on relative error rates. The remaining utilizations (0.0912, 0.1459 and 0.157) are copied into the vector $\ddot{U}$ and adjusted by adding the difference 0.0115 to become (0.1027, 0.1571, 0.1685) **(Steps 23-28)**. As in the compute phase, we calculate the $T, \tilde{T}$ and $\mathbb{E}$ matrices **(Step 29)**. We notice that the second adjusted utilization has the lowest error rate in matrix $\mathbb{E}$ **(Step 30)**. Therefore, the adjusted uti-

Algorithm 7 (CPA): Compute Propagate Adjust

```
1: Input:
2: H: Vector of discrete periods in ascending order (size $m$)
3: C: Vector of estimated WCETs (size $N$, $N < m$)
4: U: Vector of ordered utilizations (by randfixedsum algorithm) which sum up to a desired total
   utilization $\mathbb{U}$ (size $N$)
   (CPA-AU: Ascending, CPA-DU: Descending, CPA-RU: Random)
5: Output:
6: $\mathbb{O}$: Vector of $N$ pairs $(c, T)$ (WCET, period)
7: BEGIN:
8: $\mathbb{O} \leftarrow \{\}$
9: $\mathbb{E} \leftarrow \{\}$                                ▷ Error matrix, $e_{ij}$ is individual element in $\mathbb{E}$
10: while $|U| > 0$ do
11:     $\{\tilde{T}, \mathbb{E}\} \leftarrow$ COMPUTE(C, U, H)                ▷ Compute Phase
12:     I $\leftarrow \{\}$, $\epsilon \leftarrow \{\}$, V $\leftarrow \{\}$
13:     for $j \leftarrow 1, |U|$ do
14:         $\{i_{min}, e_{min}\} \leftarrow$ MIN1$(e_{ij})$ $\forall i \notin I$
15:         $\epsilon \leftarrow \{\epsilon, e_{min}\}, I \leftarrow \{I, i_{min}\}$
16:         V $\leftarrow$ {V, $(i_{min}, j)$}
17:     end for
18:     $\{j_{max}, e_{max}\} \leftarrow$ MAX1$(\epsilon)$
19:     $\forall\ v \in V$, find $i$ for which $j = j_{max}$
20:     $\mathbb{O} \leftarrow \{\mathbb{O}, (c_i, \tilde{T}_{ij})\}$
21:     $\tilde{u}_{ij} \leftarrow \frac{c_{ij}}{\tilde{T}_{ij}}$
22:     $C \leftarrow C \setminus c_i, U \leftarrow U \setminus u_j$
23:     $\delta \leftarrow \tilde{u}_{ij} - u_{ij}$                            ▷ Propagate Phase
24:     $\ddot{U} \leftarrow U$
25:     $\ddot{U} \leftarrow \ddot{U} - \delta$
26:     for $k \leftarrow 1, \left|\ddot{U}\right|$ do
27:         If $(\ddot{u}_k < 0)$, then $\ddot{U} \setminus \ddot{u}_k$  end if        ▷ Remove negative elements
28:     end for
29:     $\{\tilde{T}, \mathbb{E}\} \leftarrow COMPUTE(C, \ddot{U}, H)$
30:     $\{i, j, e\} \leftarrow$ MIN2$(\mathbb{E})$
31:     $u_j \leftarrow \ddot{u}_j$                                     ▷ Adjust Phase
32:     $T \leftarrow \{\},\ \tilde{T} \leftarrow \{\}, \mathbb{E} \leftarrow \{\}$
33: end while
34: function $\{\tilde{T}, \mathbb{E}\} \leftarrow$ COMPUTE (C, U, H)
35:     for $i \leftarrow 1, |C|$ do
36:         for $j \leftarrow 1, |U|$ do
37:             $T_{ij} \leftarrow \frac{c_i}{u_j}$
38:             $\tilde{T}_{ij} \leftarrow h_k$ for which $|h_k - T_{ij}|$ is minimal $\forall h_k \in H,\ k \in [1, m]$   ▷ find closest period
                                                                   from $H$ to the computed period $T_{ij}$
39:             $e_{ij} \leftarrow \left|\frac{\tilde{T}_{ij} - T_{ij}}{T_{ij}}\right|$
40:         end for
41:     end for
42: end function
```

lization $\ddot{u}_2$ replaces its counterpart in vector U **(Step 31)**. At the beginning of the second iteration, the utilization vector has the values (0.0912, 0.1571, and 0.157). The algorithm proceeds to pair these utilization with the remaining WCETs (20, 621, 6).

When the algorithm runs to completion, the output vector O has the following pairs ((637, 1848), (6, 40), (20, 220), (621, 3780)). The final total utilization is 74.99% with a relative error of 0.013%.

4.5.4 Compute Propagate and Adjust Algorithms - Maximize Unique Periods (CPA-MUP$_\mathrm{x}$)

The CPA algorithm imposes no limit on how many times it selects and pairs any period with tasks. It is quite possible that a set of tasks shares the same period. In some scenarios, a final set with unique periods assigned to each task might be preferable. The Compute Propagate and Adjust - Maximize Unique Periods (CPA-MUP$_x$) algorithm constrains the number of times it shares a period among different tasks by allowing a period to be assigned at most x times. It is worth noting; however, that assigning usage thresholds x applies to all periods in the set except for the last element (which corresponds to the hyper-period itself). We exclude the last element based on experimental analysis. Keeping the last period minimizes the errors when the system has a large number of tasks at a low total utilization.

We base the CPA-MUP$_x$ on the CPA-AU algorithm. We list the pseudo-code of CPA-MUP$_x$ in Algorithm 8. The main difference is that the CPA-MUP$_x$ utilizes a vector L equal in length to the periods vector H **(Input 3)**. The algorithm tracks the number of times it assigns a certain period by incrementing the corresponding entry in L **(Steps 24-25)**. Once the number of times that the algorithms uses a period reaches the limit x, we remove the period and its corresponding tracker from the vectors H, L, respectively **(Steps 26-27)**.

4.6 Experimental Setup

The first stage of the experiment is to select a set of benchmarks to be used as real-time tasks. Initially, we selected 80 benchmarks from MiBench [205] and BEEBS [207] suites. We further reduced the selection to 60 benchmarks due to portability constraints onto the target hardware and Linux OS (i.e. library dependencies, compilation issues). We statically linked and compiled the benchmarks using the *arm-linux-gnueabi-gcc* compiler v.4.8 with *-O0* optimization flag. A precursory step to testing our algorithms is to provide a vector C of WCETs. Should we execute the tasks on a single core platform, the WCET estimation is straightforward and there exists many trustworthy tools and techniques. However, the

Algorithm 8 (CPA-MUP$_\mathbf{x}$): Compute Propagate Adjust - Maximize Unique Periods

```
1: Input:
2: H: Vector of discrete periods in ascending order (size $m$)
3: L: Vector (zero initialized) of the number of times a period $h \in H$ has been assigned (size $m$)
4: C: Vector of estimated WCETs (size $N$, $N < m$)
5: U: Vector of ordered utilizations in ascedning order (by randfixedsum algorithm) which sum up
   to a desired total utilization $\mathbb{U}$ (size $N$)
6: Output:
7: $\mathbb{O}$: Vector of $N$ pairs $(c, T)$ (WCET, period)
8: BEGIN:
9: $\mathbb{O} \leftarrow \{\}$
10: $\mathbb{E} \leftarrow \{\}$                          ▷ Error matrix, $e_{ij}$ is individual element in $\mathbb{E}$
11: while $|U| > 0$ do
12:     $\{\tilde{T}, \mathbb{E}\} \leftarrow$ COMPUTE(C, U, H)                    ▷ Compute Phase
13:     I ← {}, $\epsilon \leftarrow \{\}$, V ← {}
14:     for $j \leftarrow 1, |U|$ do
15:         $\{i_{min}, e_{min}\} \leftarrow$ MIN1$(e_{ij})$ $\forall i \notin I$
16:         $\epsilon \leftarrow \{\epsilon, e_{min}\}, I \leftarrow \{I, i_{min}\}$
17:         V ← {V, $(i_{min}, j)$}
18:     end for
19:     $\{j_{max}, e_{max}\} \leftarrow$ MAX1$(\epsilon)$
20:     $\forall\ v \in V$, find $i$ for which $j = j_{max}$
21:     $\mathbb{O} \leftarrow \{\mathbb{O}, (c_i, \tilde{T}_{ij})\}$
22:     $\tilde{u}_{ij} \leftarrow \frac{c_{ij}}{\tilde{T}_{ij}}$
23:     $C \leftarrow C \setminus c_i, U \leftarrow U \setminus u_j$
24:     Find index $k$ such that $h_k \in H \mid h = \tilde{T}_{ij}$, $k \in [1, m]$   ▷ Retrieve index of assigned period
25:     Increment $l_k$                 ▷ Increment corresponding usage statistics in the $L$ vector
26:     if $(l_k = x\ \&\ k \neq m)$ then, $H \setminus h_k, L \setminus l_k$      ▷ If period assignment reaches threshold,
                                                   exclude period from further use
27:     end if
28:     $\delta \leftarrow \tilde{u}_{ij} - u_{ij}$                                 ▷ Propagate Phase
29:     $\ddot{U} \leftarrow U$
30:     $\ddot{U} \leftarrow \ddot{U} - \delta$
31:     for $k \leftarrow 1, \left|\ddot{U}\right|$ do
32:         If $(\ddot{u}_k < 0)$, then $\ddot{U} \setminus \ddot{u}_k$   end if              ▷ Remove negative elements
33:     end for
34:     $\{\tilde{T}, \mathbb{E}\} \leftarrow COMPUTE(C, \ddot{U}, H)$
35:     $\{i, j, e\} \leftarrow$ MIN2$(\mathbb{E})$
36:     $u_j \leftarrow \ddot{u}_j$                                                ▷ Adjust Phase
37:     $T \leftarrow \{\},\ \tilde{T} \leftarrow \{\}, \mathbb{E} \leftarrow \{\}$
38: end while
39: Function $\tilde{T}, \mathbb{E} \leftarrow$ COMPUTE {C, U, H}               ▷ See Algorithm 7 Lines 33-41
```

Table 4.2 Example of the CPA-AU algorithm, 1st iteration, 4 tasks at 75% utilization

		Compute Phase			
	WCET/Utilization	**0.0912**	**0.1456**	**0.157**	**0.3562**
T	**20**	219.30	137.36	127.40	56.15
	637	6,984.62	4,374.89	4,057.77	1,788.26
	621	6,809.18	4,265.00	3,955.85	1,743.34
	6	65.79	41.21	38.22	16.84
$\tilde{T}$	**20**	220	135	126	56
	637	6930	4158	3960	1848
	621	6930	4158	3960	1680
	6	66	42	40	16
$\mathbb{E}$	**20**	0.3205%	1.7175%	1.1008%	0.2603%
	637	0.7820%	4.9576%	2.4094%	3.3409%
	621	1.7744%	2.5088%	0.1050%	3.6332%
	6	0.3205%	1.9226%	4.6552%	5.0098%
e_{min}		0.3205%	1.9226%	0.1050%	**3.3409%**
		Propagation and Adjust Phases			
δ					-0.0115
$\ddot{U}$		0.1027	0.1571	0.1685	*
	WCET/Utilization	**0.1027**	**0.1571**	**0.1685**	
T	**20**	194.71	127.29	118.70	*
	*****	*	*	*	*
	621	6045.77	3952.40	3685.48	*
	6	58.41	38.19	35.61	*
$\tilde{T}$	**20**	198	126	120	*
	*****	*	*	*	*
	621	5940	3960	3696	*
	6	60	40	36	*
$\mathbb{E}$	**20**	1.69%	1.01%	1.10%	*
	*****	*	*	*	*
	621	1.75%	**0.19%**	0.29%	*
	6	2.72%	4.75%	1.10%	*
	WCET/Utilization	**0.0912**	**0.1571**	**0.157**	

Note: numbers are rounded in this table. Rounding errors might ensue

platform we use in this thesis is a multi-core single-ISA heterogeneous platform based on ARM big.LITTLE architecture. We acknowledge that WCET estimation is an open challenge in multicore platforms due to cache interference between multiple cores and the shared resources. The tightness of the estimates does not affect our algorithms for they act as a black box that pairs the inputs regardless of the methods used to collect the WCET vectors, or generate task periods and utilizations.

To the best of our knowledge, there is very limited or no support for static-based analysis on multicore systems due to the complexity of the hardware and memory subsystems. Measurement-based approaches are equally unreliable. Recent works employ MBPTA techniques based on EVT to estimate probabilistic WCETs on multicores. Slijepcevic et al. [209] used MBPTA on a quad-core platform without cache-partitioning techniques using EEMBC benchmarks that are run 1000 times. Wartel et al. [210] tested MBPTA on a real avionics systems running atop a multicore system and show that it delivers tight estimates. Cros et al. [211] also employ MBPTA for an Aerospace case study, they show that MBPTA provides 19.6% tighter WCET estimate than the industrial practice of adding 20% margin over the maximum observed operation time (MOET). Fedotova applies MBPTA on ARM Cortex-A5 processor in [212]. Silva et al. [213] also show the adequacy and tightness of using EVT-based MBPTA on complex processors running Linux. As such, MBPTA based on EVT looks promising for WCET estimation on multicore platforms.

4.6.1 Measuring Task Execution Times

Given the initial taskset of 60 tasks, we generated a total of 5000 random taskset permutations in which we assigned the tasks random core affinities that span all eight cores of the target hardware. The target hardware platform is an Odroid-XU3 (XU4) board equipped with an ARM big.LITTLE Exynos 5422 chipset. The processor hosts two heterogeneous quad core ARM Coretx-A15 and Cortex-A7 clusters. The board was running LUbuntu 12.04 based on the 3.10.y Linux kernel. We enabled the "performance" scaling governor in the kernel configuration and set it as default. This effectively ensures that Dynamic Voltage and Frequency Scaling (DVFS) is disabled. Therefore, the big (A15) and little (A7) clusters run at their maximum speeds of 2000MHz and 1400MHz, respectively.

Each core in either the A15 cluster or the A7 cluster is equipped with a Performance Monitoring Unit (PMU). Each PMU has a dedicated cycles counter. Though the Linux kernel supports hardware event collection using the *perf* tool, the tool does not concurrently support more than one cluster in heterogeneous platforms. To circumvent this limitation, we use a 3rd party tool, PMCTrack [174]. PMCTrack is an open-source OS-oriented performance monitoring tool for GNU/Linux. We patch the 3.10.y kernel and load the appropriate drivers for the Odroid-XU3 (XU4) board. PMCTrack supports assigning core affinities to the monitored tasks. We use this capability to distribute the tasks over the cores in conjunction

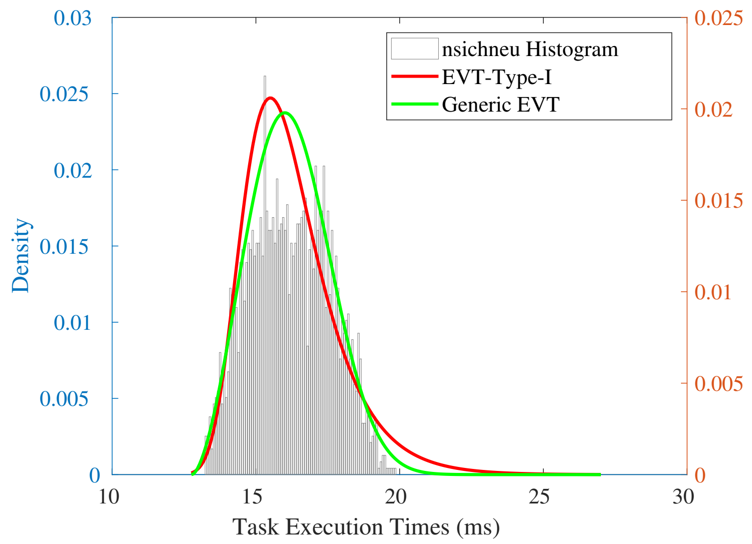


Figure 4.1 Normalized histogram of the measured execution times of Task 46 (nsichneu) with both the EVT-Type-I and Generalized EVT fits

with logging task cycles. We run each taskset for one minute on all cores. Each core assigned tasks keep cycling back to back until they reach the time threshold. Between each taskset and the next, we run a small workload to load the caches with different data. For each of the 5000 tasksets, we collected the total number of cycles of each workload. To measure the execution time of the benchmarks, we divided the number of cycles over the frequency of the core it was running on.

4.6.2 Estimating Worst Case Execution Times

We chose to estimate WCET from collected measurements using a probabilistic approach, namely EVT. In MATLAB's *Distribution Fitter* toolbox, we use both the Gumbel Distribution (EVT-Type I) and the Generalized EVT to fit and analyze the execution times. Of the 60 benchmarks, 55 had a good fit and we discarded the others. We used the probability density function of each benchmark to estimate its WCET. Similar to the approach in [26], and given the guidelines for safety assessment in airborne systems [214] that set the failure rate at 10^{-9} per hour of operation; we considered WCETs at probabilities $[10^{-9}, 10^{-12}, 10^{-16}]$. When the probability is lower, the more pessimistic the estimation; yet the less probability of failure. In this paper, we consider WCETs at 10^{-16} probability. Furthermore, in the final set, we eliminate for practical purposes benchmarks with individual pWCET that exceeds 1500ms when executed on the ARM A15 cores (the fast performance core). This is to keep our choice of periods small and limit the simulation time for any real-time schedule that will use this taskset. We list the estimated pWCETs for the final 50 benchmarks in Table 4.3. We show a sample of EVT fitting for Task 46 "nsichneu" in Figure 4.1.

Table 4.3 Estimated pWCET (ms) for 50 Tasks on ARM Cortex-A15 and Cortex-A7

Task No.	Benchmark	pWCET (A15)	pWCET (A7)	Task No.	Benchmark	pWCET (A15)	pWCET (A7)
1	*basicmath_large*	403.2	822	**26**	*bubblesort*	460.8	1252
2	*bitcnt*	503	1502	**27**	*cnt*	12.5	39.5
3	*qssort*	5.1	14	**28**	*compress*	10.8	49.5
4	*susan -c*	47.6	132	**29**	*cover*	19.8	32
5	*susan -e*	115	322	**30**	*duff*	7.3	21
6	*susan -s*	348.1	992	**31**	*edn*	198	722
7	*jpeg decode*	20	92	**32**	*expint*	11.5	39.5
8	*jpeg encode*	90.1	222	**33**	*fac*	5.8	13
9	*whetstone*	705.1	1362	**34**	*fdct*	9	39.5
10	*lout*	932.9	1522	**35**	*fibcall*	3.6	5.8
11	*aha-mont64*	39.6	122	**36**	*fir*	621	2082
12	*patricia*	1134.7	1102	**37**	*insertsort*	6	14
13	*nbody*	1174.6	2127	**38**	*janne_complex*	3.3	5.3
14	*levenshtein*	208.5	562	**39**	*jfdctint*	11	39.5
15	*sha*	134.6	432	**40**	*lcdnum*	4.2	5.3
16	*sqrt*	267	572	**41**	*ludcmp*	12.1	54.5
17	*inverse fft*	1215.7	1696	**42**	*matmult-int*	260.1	1182
18	*fft*	1299.6	1442	**43**	*minver*	10.3	39.5
19	*raw audio pcm*	510.5	1272	**44**	*ndes*	232.5	472
20	*rawaudio adpcm*	637	1592	**45**	*ns*	27.4	122
21	*st*	51.8	222	**46**	*nsichneu*	29.5	107
22	*statemate*	5.7	17	**47**	*prime*	81.4	232
23	*string search*	16.4	62	**48**	*qurt*	6.6	17
24	*ud*	14.7	54.5	**49**	*recursion*	8.9	32
25	*matmult-float*	62.2	282	**50**	*select*	6.6	20

4.6.3 Evaluation of the Algorithms

To evaluate the performance and compare between the PU, CPA, and MUP algorithms variants, we created a test-bench which iterates over the total utilization range of [0.1 , 0.9] in increments of 0.05 (17 total utilizations). In each iteration, we generated 500 random utilization distributions. Each distribution sums up to the total utilization of the current iteration. We generated our bounded periods set from the 31st HCN. Our rationale in choosing the 31st HCN (166,320) as a starting point is to allow the task of highest pWCET on the Cortex-A15 core to have a utilization as close as possible to 1%. The pWCET of

Task 18: *fft* is 1299.6, resulting in utilization of 0.78%. Similarly, for the Cortex-A7 core, the pWCET for Task 13: *nbody* is 2127 resulting in a utilization of 1.27%.

Subsequently, we pass the generated utilizations, bounded periods and task WCETs to each of the algorithms under consideration. We vary the number of tasks in the system between 4 and 50 (for a total of 47 simulation points per utilization). Yet, we cap the taskset size for the PU algorithm at 10 due to computational limitations. In total, we ran nearly 400,000 simulations for each algorithm (except for PU $\approx$ 60,000 simulations) and collected statistical data.

4.7 Experimental Results

To avoid redundancy, we only show results related to the Big cluster (Cortex-A15 cores). We analyze the results in terms of how the final system utilization of a paired taskset deviates from the desired total system utilization. We start with the results of the PU algorithm that we illustrate in Figure 4.2. For a system with varying number of tasks between four and ten, we notice that the median relative errors of PU algorithm hover around 2.5% and 5.0% (surface plot) whereas they were around 0.3% for two variants of the CPA algorithm. The minimum recorded values of the PU algorithm also designate error rates in the vicinity of 1.0% - 3.0%. The PU exhaustive exploration does not necessarily yield a better solution than what our algorithms provide. We observe that the minimum relative errors of the PU algorithm were on average higher than the average errors of our algorithms. The CPA algorithm has an order of magnitude less errors due to the continuous readjustment of the individual utilizations, a feature that the PU algorithm lacks.

When we compare the average execution time of the main algorithms, the $O(n!)$ complexity of the PU algorithm is clearly visible in Figure 4.3. Increasing the taskset size by one from nine to ten increases the execution time of the algorithm by a factor of ten. In contrast, the CPA-AU algorithm, and the MUP variants are more efficient. Using curve fitting tools, we find that for the CPA-AU, the algorithmic time can be expressed as polynomial of degree three with coefficients $(1.82e-06, 1.507e-05, 1.181e-06, 0.0002625)$. For this model, the R^2 is 1, the SSE equalled $1.8335e-06$ and the RMSE $2.0649e0-4$. Given this model, it will take the algorithm 15 seconds to generate parameters for a taskset of size 200 on our research machine.

In Figure 4.4, we compare the relative percentage errors of the CPA and MUP algorithms when simulated for a hyper-period of 166,320. We notice that the CPA-AU, CPA-RU and CPA-DU (Figs. 4.4a, 4.4b, 4.4c) perform well across much of the exploration space. The

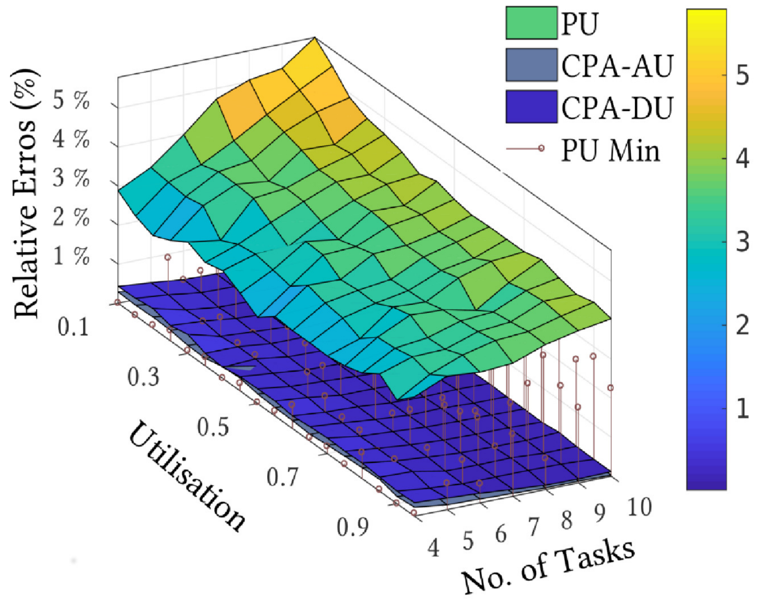


Figure 4.2 Relative error (%) from target system utilization

only exception is when we simulate large tasksets ($>$ 30 tasks) at very low utilizations (specifically a utilization of 10%). In this corner case, the CPA-AU algorithm outperforms both the CPA-RU and CPA-DU with an average error rate of 2% for the case of 50 tasks, compared to 6% and 18% for CPA-RU and CPA-DU, respectively.

When the total utilization is small, the individual task utilizations are even smaller. Therefore, very large periods are required to satisfy the small utilizations. The first source of error is due to the fact that the periods are farther apart when they get large and towards the end of the period set. This means less large periods to select from. The other source of error is capping the largest required period by the maximum available period of the set.

It is also important to report the variance values of our investigation across all simulated tasksets. We show the variance for the best performing algorithm CPU-AU in Figure 4.5. The small variance overall shows the consistent and solid performance of the algorithm. Despite a slight increase in the variance for the corner case, it is still quite low ($< 10^{-5}$).

The effect of imposing limits on the number of times the CPA-MUP$_x$ algorithm selects a period is visible in Figures 4.4d, 4.4e and 4.4f. As expected, CPA-MUP$_1$ shows the most errors. This is due to the fact that requiring more unique periods reduces the size of the periods set by at most the same number of system tasks. Relaxing the limits improves the performance as evidenced in both the CPA-MUP$_2$ and CPA-MUP$_3$ results. If we take the CPA-MUP$_2$ algorithm as an example considering 50 tasks at 30% target utilization,

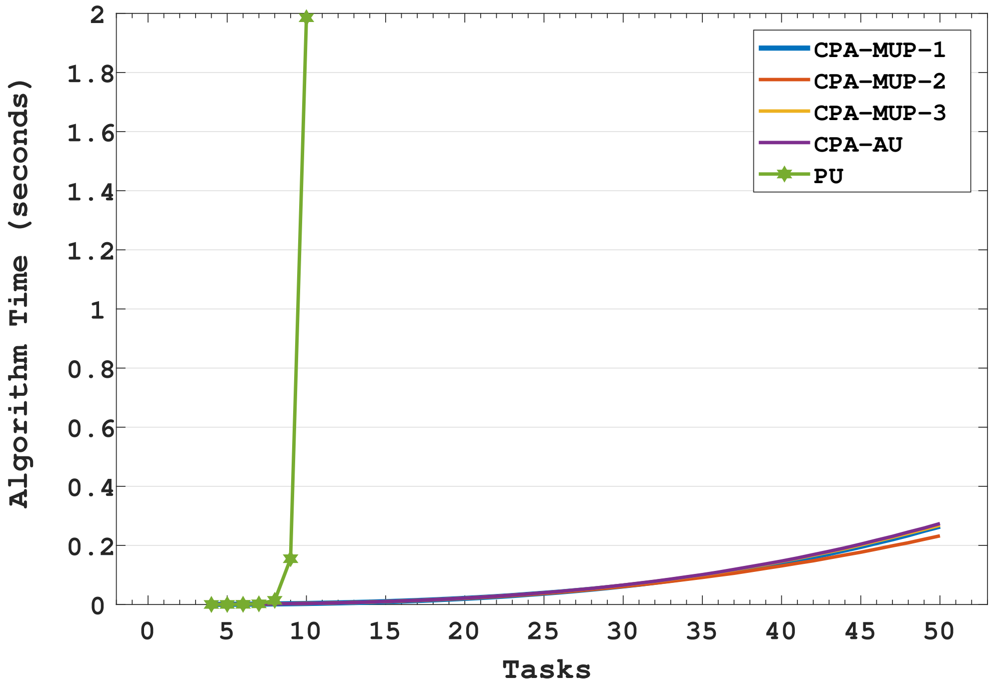
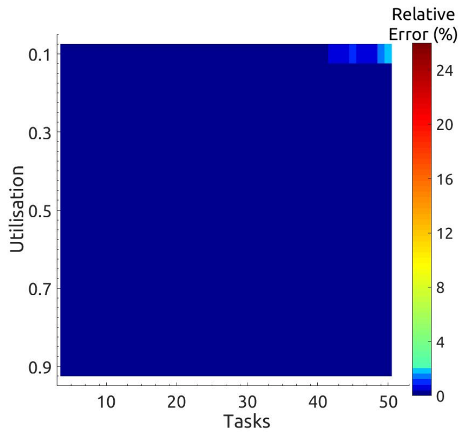


Figure 4.3 Average execution time of the proposed algorithms vs. the task set size

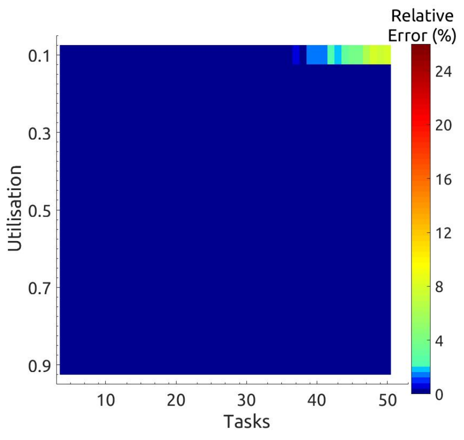
the algorithm is on average capable of constructing tasksets with 28.2% utilization. In comparison, the CPU-AU almost matches the desired utilization at 29.97%. Imposing unique periods in CPA-MUP$_1$ yields a total system load of 22.5%, a 25% drop from the target.

Similar to CPU-AU algorithm, the CPA-MUP$_x$ algorithm suffers from poor performance for the case of large tasksets at small utilizations. However, the errors are more noticeable for CPA-MUP$_x$ because a major downside of discarding periods is limiting the number of available periods to select from. One possible way to mitigate this issue is by starting with a larger initial period set. In Figure 4.6, we show that using larger HCN numbers to build our period set has a diminishing effect on the relative errors for the CPA-MUP$_1$ algorithm.

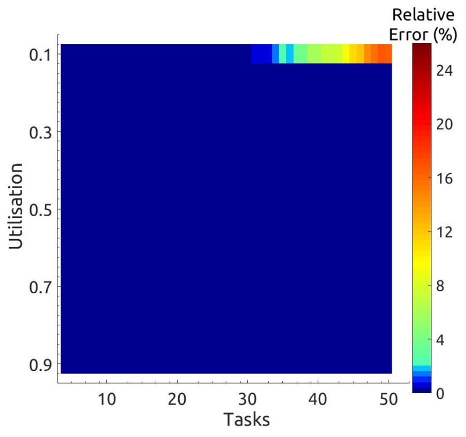
An important criteria we use to compare the algorithms is how many unique periods they can produce on average. Every point of the exploration space (17 utilizations $\times$ 47 possible task sizes) has 500 simulations. For each one of these simulations, we count the number of unique periods in the resulting taskset and divide it by the taskset size. We then average these ratios at every point. We group the averaged uniqueness percentages and use them to draw a histogram. We repeat this procedure for each one of the CPA-AU and CPA-MUP$_x$ algorithms and across four different simulation intervals (166.32, 221.76, 277.20, 332.64) ms. We group the individual histograms into a 3D histogram which we illustrate in Figure 4.7.



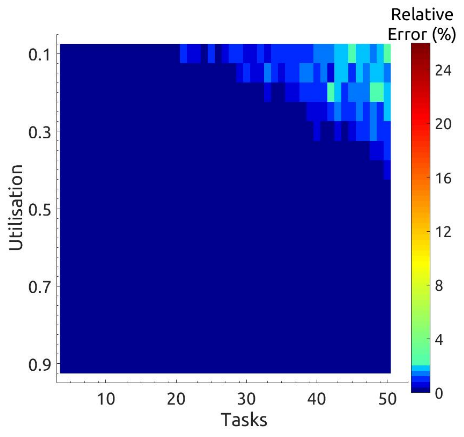
(a) CPA-AU



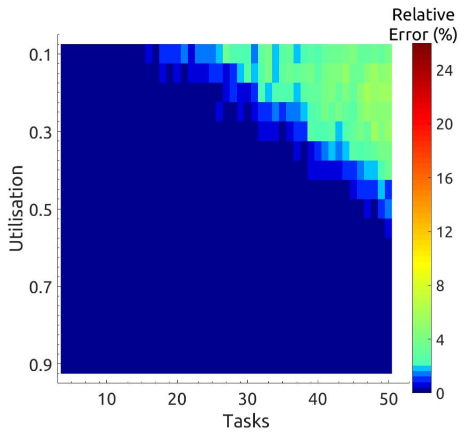
(b) CPA-RU



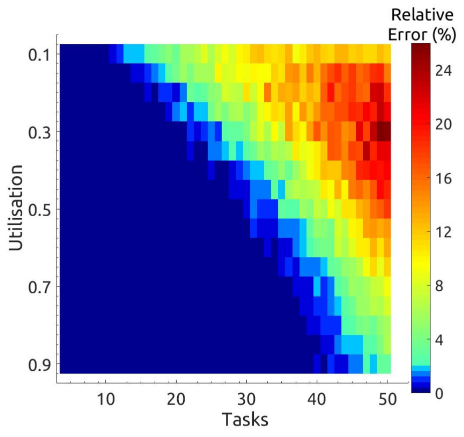
(c) CPA-DU



(d) CPA-MUP$_3$



(e) CPA-MUP$_2$



(f) CPA-MUP$_1$

Figure 4.4 The algorithms CPA-AU, CPA-DU, CPA-RU and CPA-MUP$_x$ assign task periods from a bounded period set to set of N tasks with given pWCETs. The mapping is to satisfy a certain utilization U. Figures (a) to (f) show the relative error rates of the utilization after final mappings. All algorithms use periods with HP = 166.32 seconds.

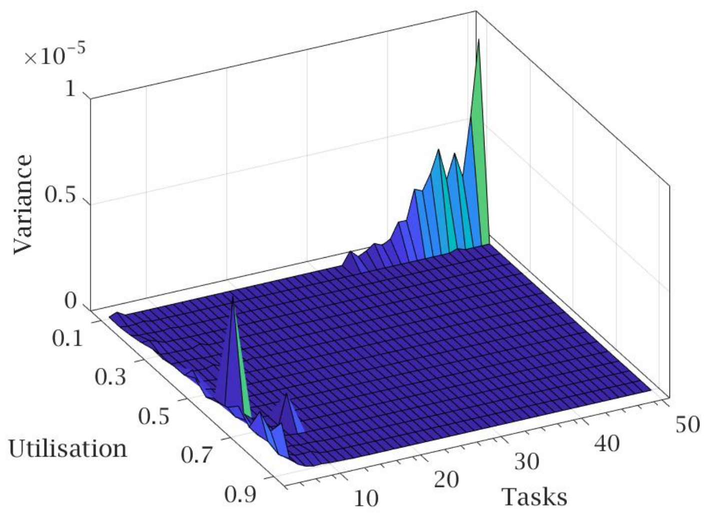


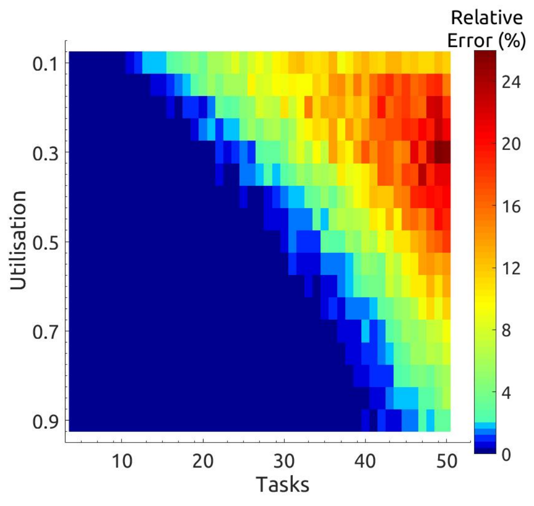
Figure 4.5 Variance of relative errors for CPA-AU algorithm at hyper-period of 166,320ms

Predictably, when we use larger HCNs (Hyper-Periods) to generate the periods set, there is less chance to select the same period multiple times. We notice that at a hyper-period of 166.32 seconds, the majority of the tasksets generated by the CPA-AU, CPA-MUP$_2$ and CPA-MUP$_3$ have between 70% and 100% unique periods. This rises to a majority that has between 80% to 100% unique periods when the hyper-period increases to 332.64 seconds. Even though the CPA-MUP$_1$ algorithm generates the most unique periods, the percentage of unique periods is never 100%. This is because the algorithm does not discard the largest period in the set. For the corner case of large number of tasks at small utilizations, the algorithm repeatedly selects this period degrading the uniqueness index.

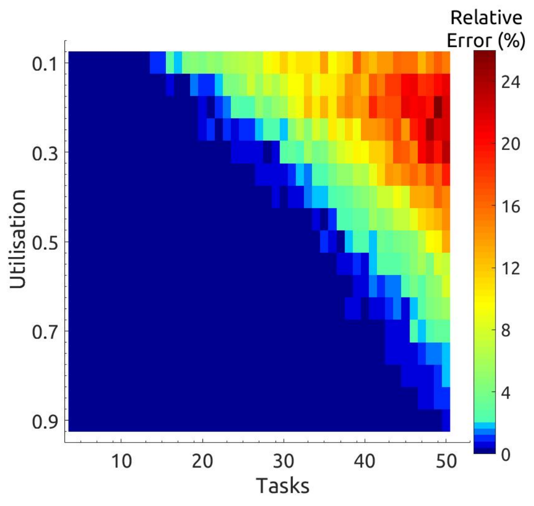
It is worth noting that for any given hyper-period, the differences between CPA-MUP$_2$, CPA-MUP$_3$, and CPA-AU are not that distinct. In fact, the three algorithms produce tasksets with at least 70% unique periods. Weighing on this observation, and contrasting it with the higher error rates the CPA-MUP$_x$ algorithms have, we conclude that the CPA-AU algorithm outperforms the others.

4.8 Chapter Summary

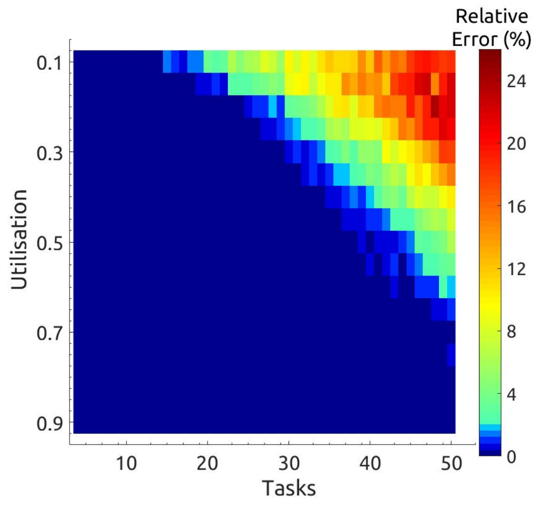
This chapter proposed a set of algorithms to prepare tasksets for the purpose of evaluating real-time algorithms on real hardware. As opposed to conventional approaches, our methodology does not build or synthesize tasks but instead is based on user supplied tasks either



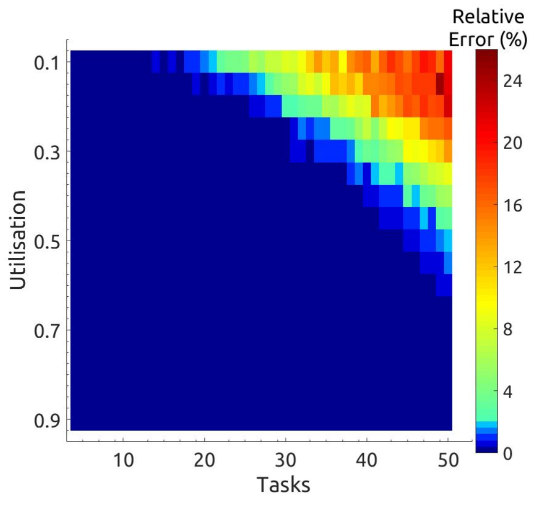
(a) HP=166,320ms



(b) HP=221,760ms



(c) HP=277,200ms



(d) HP=332,640ms

Figure 4.6 The relative errors of CPA-MUP$_x$ algorithm as HP increases

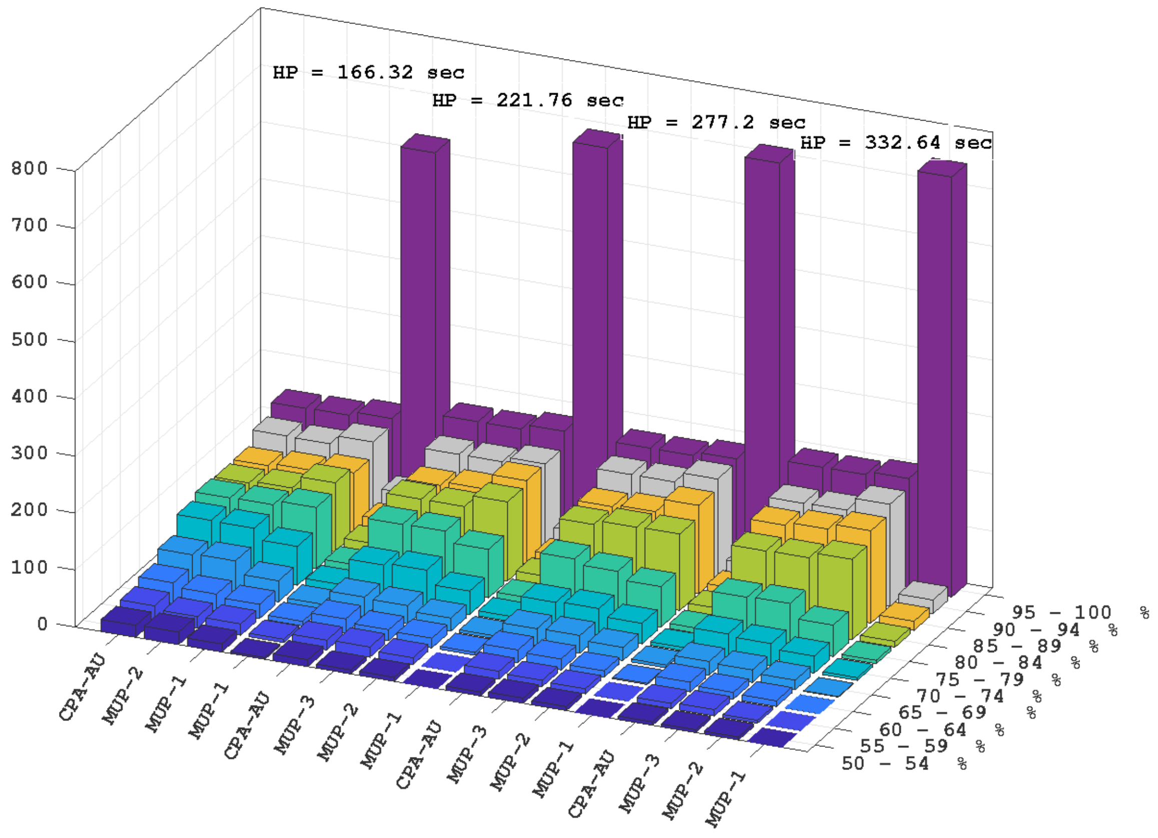


Figure 4.7 A 3D histogram of unique periods for the main algorithms over different simulation hyper-periods

from embedded benchmark suites or “in-house” codes. Coupled with a set of discrete periods and individual task utilizations, the proposed algorithms adjust the random individual task utilizations and pair tasks to periods to satisfy total system utilization constraints.

The first set of algorithms CPA (AU, RU, DU) investigate the effects of initial utilization ordering on the performance of the algorithms by computing and comparing relative error rates. The CPA-AU algorithm outperformed the other CPA variants and delivered results in polynomial time. The CPA-AU relative errors are at least an order of magnitude less than any possible heuristic or exhaustive approach based on direct pairing and no utilization adjustment. This chapter also introduced a set of algorithms based on the CPA-AU algorithm which maximizes the number of unique periods assigned to each task in the taskset. The final assessment showed that there is a trade-off between the number of unique periods assigned in a taskset and overall accuracy. If unique periods are required, the algorithm will not be able to satisfy the target utilization. A more relaxed CPA-MUP$_x$ uniqueness thresholds or longer hyper-periods slightly mitigate the issue. However, the analysis shows that CPA-AU algorithm achieves comparable results to the relaxed versions of the CPA-MUP$_x$ with minimal errors.

In our methodology, generating the period set and individual task utilization vectors as inputs to the algorithms is straightforward. However, estimating WCETs of a set of real-world embedded benchmarks is not. Despite the fact that the CPA algorithms are independent of the techniques used to derive the inputs to the algorithms, it is worth to address the issue of estimating WCETs especially if the goal is to construct tasksets for simulation on multicore platforms. Some preliminary research finds MBPTA approaches based on EVT promising. As embedded processors keep incorporating MBPTA-friendly features (e.g. random replacement cache policy), this will facilitate finding tight and safer WCETs for complex systems.

Chapter 5

Energy and Task-Aware Partitioning on Single-ISA Clustered Heterogeneous Processors (CHPs)

5.1 Introduction

Embedded heterogeneous multi-core processors combine cores of different performance and energy consumption profiles on the same die. The varying performance and energy efficiency between the heterogeneous cores add additional dimensions to an already complex optimization problem. Recent heterogeneous chips include AMD's heterogeneous architecture [215] where the processing unit and the GPU share the same bus, memory, and tasks with the aim of reducing communication latency. In embedded platforms, most offerings are based on cores that share the same instruction set architecture *ISA*. Examples of single-ISA heterogeneous SoCs include Nvidia Tegra3 [216], TI OMAP 5 [217], and ARMs traditional big.LITTLE platforms [218]. Nvidia's approach is to combine quad-core high-performance cores alongside a fifth companion core of the *same* type. The fifth core is built using a special low power silicon process that runs at lower frequencies in what Nvidia calls Variable Symmetric Multiprocessing *vSMP*. TI OMAP 5 combines dual core Cortex-A15 with dual core Cortex-M4 processors. Despite belonging to different ARM Cortex families, the cores share the same armv7 ISA.

ARM's big.LITTLE based SoCs feature dual clusters of different processors of the Cortex-A family that share the same ISA. Processors within the same cluster share the same clock frequency and are controlled by the same DVFS circuity. ARM big.LITTLE processors are ubiquitous in mobile systems and dominated by products from Samsung (Exynos 5 and 7 series), Qualcomm (Snapdragon 600 and 800 series), and Kirin (900 series). Hexa (2+4) and Octa (4+4) core platforms are the two available configurations. These processors are either based on 32-bit architecture (e.g. A15/A7 cores using armv7-a ISA), or 64-bit architecture (e.g. A73/A53 cores using armv8-a ISA). Recently, ARM introduced a refined concept of ARM big.LITTLE under the name big.LITTLE dynamIQ [219]. The first improvement is having the ability to add more clusters with different energy profiles. (i.e. a cluster of a third core type (medium core), or a cluster with the BIG core type but with different supported frequencies). The second improvement allows for combining BIG and LITTLE processors in new configurations (e.g 1 BIG + 7 LITTLE or 2 BIG + 6 LITTLE). A third design improvement adds the potential for per-core DVFS (though recent products do not yet apply it due to its cost). However, as of the time of writing this thesis, only one cluster is allowed to be active at any time instant. This is be expected to change in the future.

The real-time literature adapts to the advancements in the underlying hardware and offers new solutions to challenging problems. However, in most cases, it often simplifies energy, power, and task models. For example, given the real-time system constraints and processor overheads, it might not be possible to make use of the multi-level and deep sleep states. Furthermore, many ignore the impact of the different mixes of the task instructions on energy consumption. Relying on task execution time as the sole metric for the energy consumption evaluation is misleading. Tasks exhibit different execution paths and inherently distinct cache and I/O access behavior. As such, tasks do not necessarily consume the same power if run on the same processor at the same frequency. For example, tasks with heavy cache access would consume more energy. Moreover, tasks with hardware floating point or SIMD instructions have different performance and energy characteristics given the specialized circuity they use.

5.2 Chapter Contributions

Motivated by the ubiquitous proliferation of heterogeneous multicore systems, most notably ARM big.LITTLE processors as an example of a successful single-ISA heterogeneous architecture, the goal of this chapter is to present and analyze task allocation heuristics for reducing the energy consumption on such platforms. Our work does not assume that tasks consume the same power when run on the same core type at the same frequency. We consider realistic cases where the task instruction mix and behavior dictate that tasks will have different power profiles. Instead of relying on simulations, we directly evaluate and compare our algorithms on real-hardware. This reduces any discrepancies between simulation results and real-world deployment. We summarize our contributions as follows:

- We provide a comprehensive analysis and profiling of the energy and performance efficiency of real-world embedded benchmarks running on Samsung Exynos 5422 SoC featuring an Octa-core single-ISA big.LITTLE platform based on ARM Cortex-A ARMv7 Architecture.
- We propose algorithms for task allocation across the heterogeneous cores for hard real-time periodic task sets. Our algorithms are inspired by the analysis of the different energy profiles of the tasks across the different supported cluster frequencies.
- Based on our methodology for facilitating task evaluation on real-hardware, we run our algorithms and report results on Odroid XU3 boards. The boards run a customized Linux kernel patched with Litmus-RT RTOS that supports partitioned EDF scheduling. We test the performance of our algorithms under various system load points. Running our tasksets on real-hardware allows us to report actual energy consumption and takes into account effects that are often ignored in simulations due to their complexity.
- We compare our results to the HIT-LTF algorithm [148], which, up to our best knowledge, is the most recent work utilizing similar hardware, energy, and task models. Our algorithms consider the allocation and frequency simultaneously, in contrast to HIT-LTF.

The remainder of the chapter is organized as follows: Section 5.3 introduces the hardware platform model, its associated power model, and the task and energy models used in this chapter. The chapter proceeds with the proposed algorithms in Section 5.4. The experimental setup follows in Section 5.5. Section 5.6 illustrates and discusses the evaluation results of the presented algorithm. The chapter concludes with a summary.

5.3 System Model

5.3.1 Hardware Platform Model

A heterogeneous platform Π is composed of M clusters κ_i. Each cluster κ_i can have a different number Q_{κ_i} of homogeneous processing elements (cores) π_j. However, the platform Π must consist of at least two different clusters that differ in terms of performance and energy efficiency. We use the notation $\pi_{\kappa_i,j}$ to denote core j of cluster κ_i. The total number of cores in the system is equal to $Q = \sum_{i=1}^{M} Q_{\kappa_i}$.

All clusters κ_i implement the same instruction set architecture (ISA). Given that per-core-DVFS is costly and complicated, we assume that all processing elements π_j within cluster κ_i share the same frequency and voltage domain. This assumption is corroborated by the fact that modern embedded heterogeneous processors to date implement this model [217, 218]. Each cluster κ_i, and by extension, each core within the cluster $\pi_{\kappa_i,j}$ share a finite set f_{κ_i} of frequency scales $f_{\kappa_i,1}, f_{\kappa_i,2}, \ldots, f_{\kappa_i,n_{\kappa_i}}$ in increasing values. The notation $f_{\kappa_i,k}$ denotes the k^{th} frequency supported by cluster κ_i while n_{κ_i} represents the number of available frequencies supported by a cluster. The maximum frequency supported on a cluster is denoted as $f_{\kappa_i}^{max}$ which corresponds to the cluster frequency at index n_{κ_i}.

5.3.1.1 Experimental Platform

In this chapter, we consider a single-ISA heterogeneous platform based on ARM's big.LITTLE architecture. We use the Odroid-XU3 board. This board features Samsung Exynos 5422 SoC that boasts an octa-core configuration in two clusters ($M = 2$). Each cluster has a set of quad-core processors ($Q_1 = Q_2 = 4$). The first cluster κ_1 (BIG cluster, a.k.a κ_B) incorporates four Cortex-A15 cores while the second cluster κ_2 (LITTLE cluster, a.k.a κ_L) has four Cortex-A7 processors. The Cortex-A15 processor is performance oriented while the Cortex-A7 is more energy-efficient. Both processor types are based on ARM's armv7a architecture. The big cluster supports 19 frequency levels from $f_{B,1} = 200$ MHz to $f_{B,19} = 2000$ MHz. On the other hand, the small cluster supports 13 frequency levels from $f_{L,1} = 200$ MHz to $f_{L,13}$ $= 1400$ MHz. Each of the eight cores in both clusters has an 8-set associative 32KB L1 instruction and data caches. The quad processors in the LITTLE cluster share a 512KB L2 cache whereas the quad cores of the BIG cluster share a 2MB L2 cache. The two clusters have a 128-bit coherent bus interface. Fig. 5.1 presents the general layout of the Exynos 5422 SoC. In Fig. 5.2, we show the speed-ups attained for running a selection of benchmarks from the MiBench [205] and BEEBS Benchmarks [207] suites. We set our baseline to be the results of running the benchmarks on the Cortex-A7 processor at the maximum frequency of 1400MHz. We report speedups when the tasks are run on the Cortex-A15 processor at both 1400MHz (the same as the maximum frequency of the Cortex-A7) and the maximum A15 cluster frequency of 2000MHz.

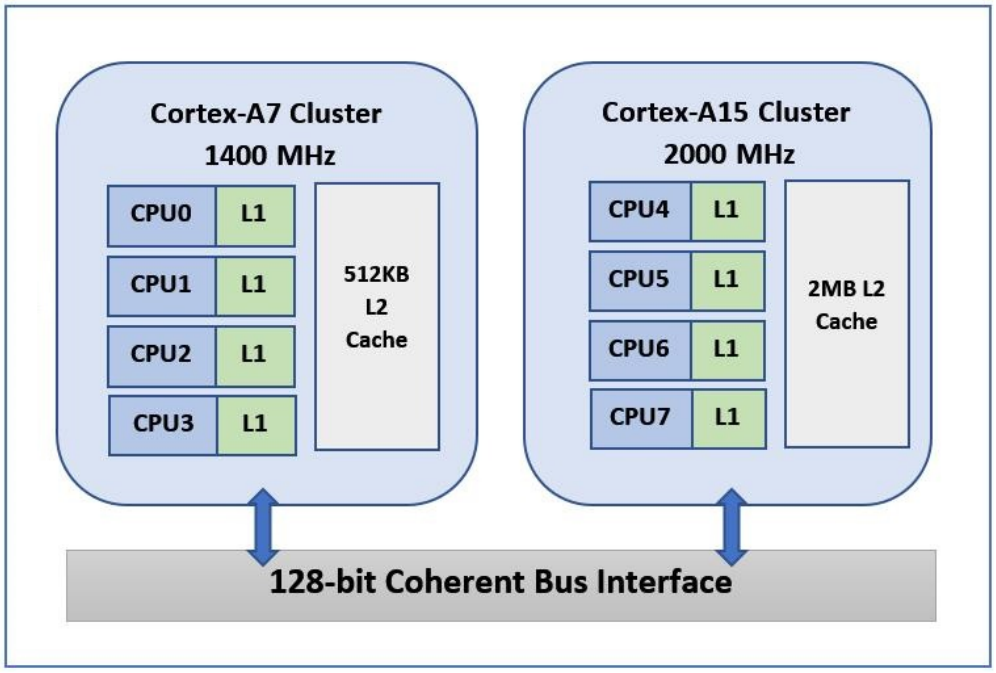


Figure 5.1 Architecture of the Samsung Exynos 5422 SoC featuring an octa-core big.LITTLE heterogeneous clusters

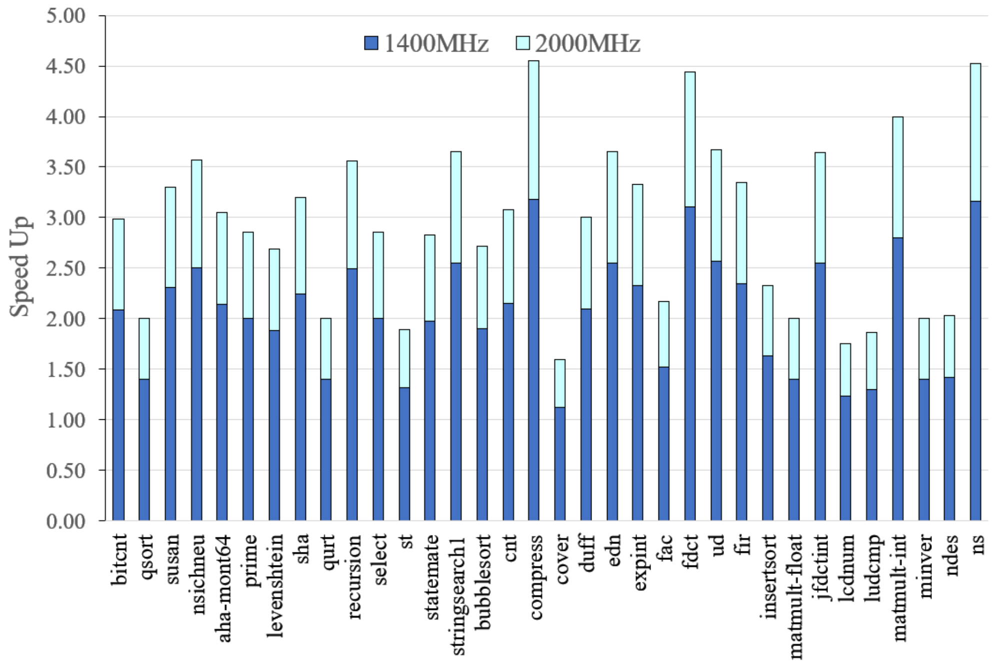


Figure 5.2 Normalized performance speedup of running embedded benchmarks on Cortex-A15 (at 1400 and 2000MHz) against Cortex-A7 (at 1400MHz)

5.3.2 Task Model

We consider a hard-real-time system modeled by a set Γ of N independent periodic tasks τ_n. We assume a synchronous release-time model where all tasks are released at the same time $t = 0$. The system releases infinite number of task instances (jobs) with a period T_n associated with each task τ_n. We further assume an implicit deadline model where each task has a hard deadline d_n equal to its period. A task needs a number of cycles to complete that varies depending on different inputs, initial conditions, or the system state. Each task has a worst-case execution cycles (WCEC) parameter $\mathring{c}_{\kappa_i}^{\tau_n}$ that depends on the task instructions, processor architecture, and system state. We calculate the worst-case execution time (WCET) by dividing the number of the worst-case execution cycles needed to complete the task over the frequency of the cluster on which it is running $c_{\kappa_i}^{\tau_n} = \mathring{c}_{\kappa_i}^{\tau_n} / f_{\kappa_i,k}$. We initially estimate the WCECs and WCETs at the highest frequency of each cluster given the procedure outlined in Section 4.6 based on EVT-based MBPTA. However, to analyze the relationship between WCECs and cluster frequency, we conducted our tests on one active core on each cluster because the interference from other cores will be difficult to isolate. In this scenario, we find that WCEC varies with frequency by at most 1% for each task in our taskset Γ. This might not be true for any other tasksets based on different benchmarks as it depends on the size of the tasks, their nature (CPU bound or memory bound), and the size of the data they handle. Given the small variation in WCECs that we measured, we assume that the WCET scales linearly with the frequency in an approach similar to other works [148]. We denote the utilization of the task running on cluster κ_i as $u_{\kappa_i}^{\tau_n}$.

We consider partitioned multi-core scheduling in which we permanently assign a fixed core for each task to execute on. All task instances execute on the assigned core and task migration is not allowed between cores or clusters. For any taskset Γ, we denote the task partitioning on all system cores by Θ, and on a specific cluster by Θ_{κ_i}. We denote the set of tasks assigned to a core in a processor cluster by $\Theta_{\kappa_i,j}$. For a task partitioning to be considered feasible, all tasks must meet their deadlines. In this work, we use the Earliest Deadline First (EDF) scheduler. Each core has its own task queues and separate EDF scheduler that handles the set of tasks assigned to it. In EDF scheduling, a partitioning of the tasks $\Theta_{\kappa_i,j}$ is schedulable when it satisfies the condition that the total task utilization on each core must not exceed unity; that is $U_j = \sum_{\tau_n \in \Theta_{\kappa_i,j}} u_{\kappa_i}^{\tau_n} \leq 1$.

The upper bound for the simulation interval for EDF schedulers running independent tasks with implicit deadlines on uni-processors or partitioned multiprocessors is equal to the hyper-period [188, 189]. The scheduling pattern is cyclic after the established bound on the

Table 5.1 Latency (in ms) to deactivate/wake up the Exynos 5422 big.LITTLE cores [4]

Core Type	**Cortex-A7**		**Cortex-A15**	
Freq. (MHz)\Transition State	*On*	*Off*	*On*	*Off*
200	4.19	10.03	4.91	13.89
1400	1.82	2.17	4.52	6.30
1800	-	-	4.54	5.40

time schedule.

5.3.3 Power Model

We consider the same power model introduced in Section 2.2 where processors consume dynamic and static power. We denote the total power the task consumes when running at cluster κ_i and cluster frequency $f_{\kappa_i,k}$ by $P^{\tau_n}_{f_{\kappa_i,k}}$.

We consider two cases. The first case is when we aggressively switch the cores to sleep mode whenever no tasks are running such that the cores only consume negligible sleep power $P^{sleep}_{\kappa_i}$. However, in this case the switching overhead to deep sleep and back to the active state might be prohibitive in certain real-time applications. In the second case, the processors never goes to sleep mode and instead consumes idle power $P^{idle}_{\kappa_i}$. Colin et al. [4] have analyzed the switching overhead (in ms) for activating and deactivating the cores of the Exynos 5422. We report their findings in Table 5.1.

5.3.3.1 Platform and Task Power Analysis

The average power dissipation varies between tasks even if they execute on the same frequency on the same core. We profile the average power consumed by each task using the built-in Texas Instruments INA231 on the Odroid-XU3 board. These sensors connect to each of the quad core A15 and A7 clusters. We run each one of the 45 system tasks back-to-back for 60 seconds on each of the heterogeneous clusters where one core was active at a time. We take the readings at intervals of 200 ms. Similarly, we measure the idle power by invoking the Linux *sleep* command similar to the approach taken in [220].

Fig. 5.3 and Fig. 5.4 illustrate how the total task power varies between tasks for a given core at the same specified frequency. For the taskset in our experiment, the variation between the maximum (*fir2dim*) and minimum power observed (*edn*) is 102% at 2000MHz, and 86.4%

at 1400 MHz for the Cortex-A15. Whereas for the Cortex-A7 processor, the maximum variation in power consumption between tasks at maximum frequency is 43.1%. We further observe that the tasks that consumed the highest and lowest power on the Cortex-A15 are not necessarily the same for Cortex-A7. The architectural differences between the two cores entail different active circuits and behavior for the various task instructions which in turn affect the power consumed.

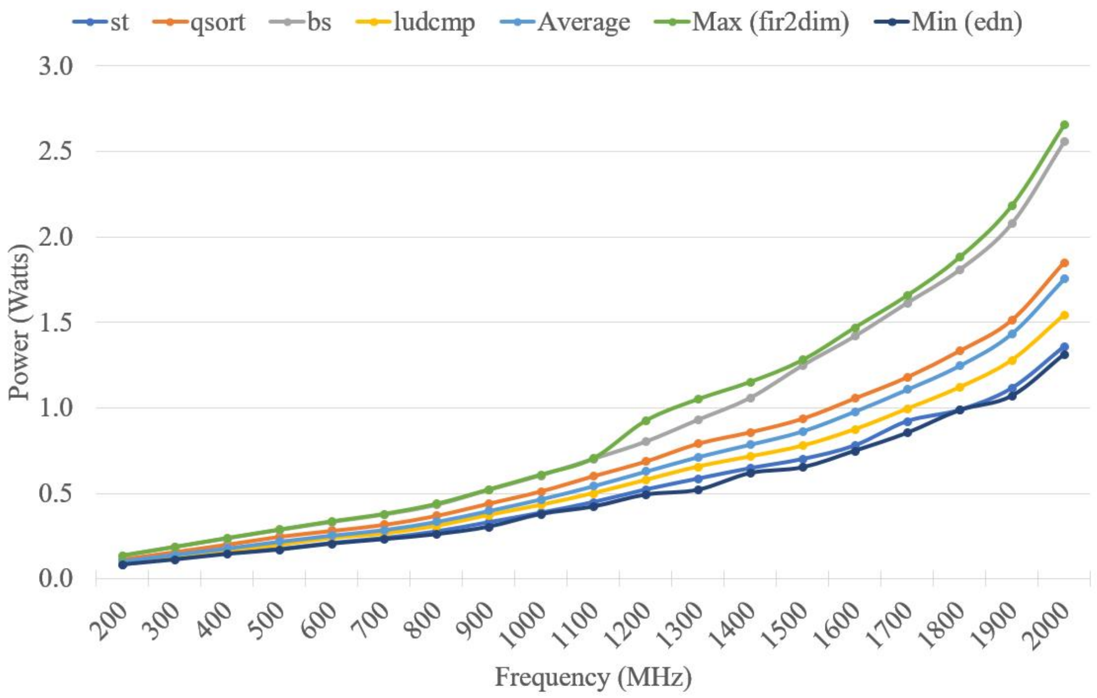


Figure 5.3 Total power consumption for select tasks and the average power for the entire taskset for the Cortex-A15 processor

5.3.4 Energy Model

Task energy is the integration of task power over the task execution time. We express the energy consumption of a single instance of task τ_n running on a core of cluster κ_i at frequency $f_{\kappa_i,k}$ as in Equation 5.1:

$$e^{\tau_n}_{f_{\kappa_i,k}} = P^{\tau_n}_{f_{\kappa_i,k}} \times c^{\tau_n}_{\kappa_i} \tag{5.1}$$

For any task τ_n, $e^{\tau_n}_{\kappa_i}$ denotes the vector of energy consumption of a single instance of task τ_n at the various frequencies f_{κ_i} of cluster κ_i. We further denote $e^{\Gamma}_{\kappa_i}$ as the grouping of vectors $e^{\tau_n}_{\kappa_i}$ for all $\tau_n \in \Gamma$.

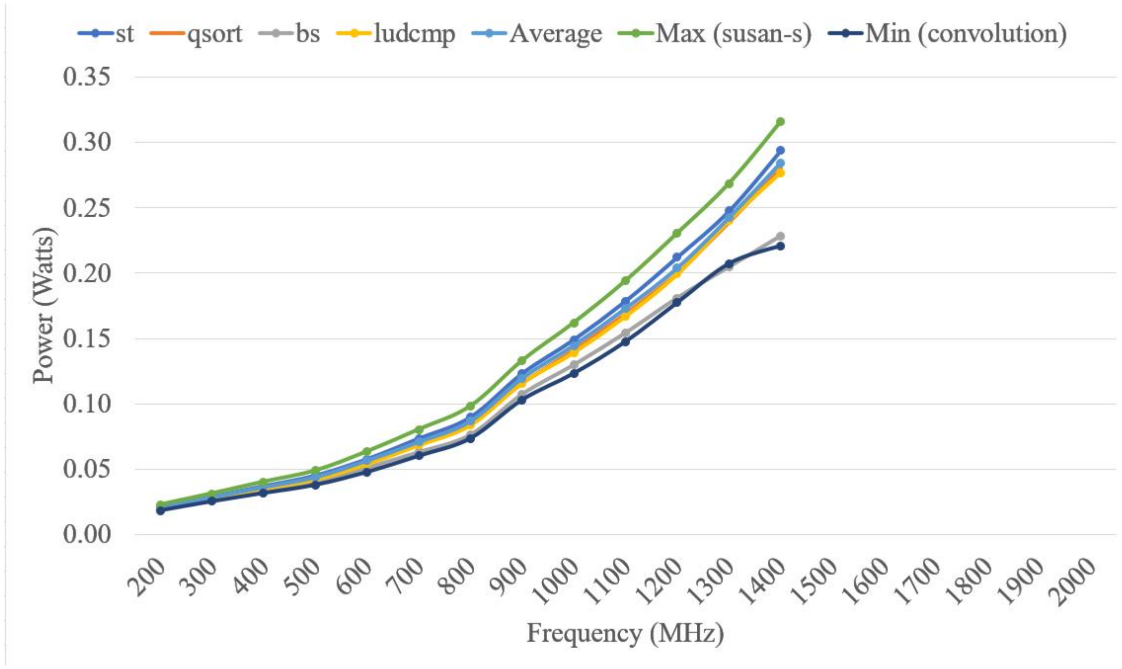


Figure 5.4 Total power consumption for select tasks and the average power for the entire taskset for the Cortex-A7 processor

Equation 5.2 expresses the total energy consumption of a task τ_n running on a core of cluster κ_i at frequency $f_{\kappa_i,k}$ for the whole duration of a hyper-period as:

$$E^{\tau_n}_{f_{\kappa_i,k}} = \frac{H}{T_n} \times e^{\tau_n}_{f_{\kappa_i,k}} \tag{5.2}$$

In case the processor remains idle when no tasks are running, we define the total energy consumed by all tasks assigned to a processor cluster running at frequency $f_{\kappa_i,k}$ by Equation 5.3:

$$E_{f_{\kappa_i,k}} = \sum_{j=1}^{Q_{\kappa_i}} \sum_{\tau_n \in \Theta_{\kappa_i,j}} (E^{\tau_n}_{f_{\kappa_i,k}} + H(1-U_j)P^{idle}_{\kappa_i}) \tag{5.3}$$

Similarly, if the processor is to be switched to sleep modes in between active tasks, we define the total energy consumed within the cluster in Equation 5.4. Though, in most cases, the sleep energy can be negligible.

$$E_{f_{\kappa_i,k}} = \sum_{j=1}^{Q_{\kappa_i}} \sum_{\tau_n \in \Theta_{\kappa_i,j}} (E^{\tau_n}_{f_{\kappa_i,k}} + H(1-U_j)P^{sleep}_{\kappa_i}) \tag{5.4}$$

Given the intricate interplay between dynamic and static energy consumption, lower cluster frequencies do not necessarily translate to lower overall energy consumption. The increased execution time and therefore increased leakage current might outweigh the reductions in dynamic energy consumption at lower frequencies. As such, there exists a subset of low frequencies at which it is no longer energy-efficient to operate. The critical frequency is defined as the cluster cut-off frequency below which it is energy-inefficient to assign tasks to the cluster. We denote the cluster critical frequency as $f^{crit}_{\kappa_i}$. We profile the energy performance of both the Cortex-A15 and Cortex-A7 clusters in Fig. 5.5. We report the results in units of *energy-per-cycle* for the maximum, minimum and average of our 45 taskset. It is evident that the critical frequency for the A15 cluster (f^{crit}_{B}) rests at the 700 MHz point. The critical frequency of the A17 cluster (f^{crit}_{L}) sits at the 500 MHz frequency point.

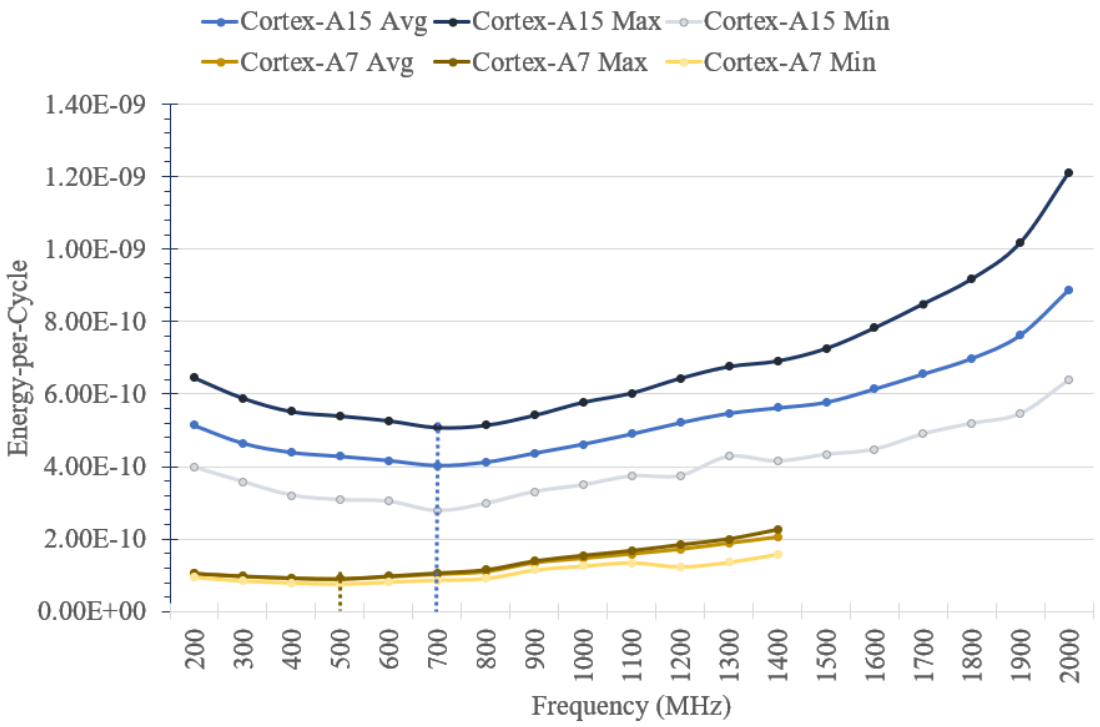


Figure 5.5 Profiling the energy-per-cycle consumption for the Cortex-A15 and Cortex-A7. The critical frequency (dashed line) is shown to be 700MHz for the A15 and 500MHz for the A7

5.4 Our Algorithms

In this section, we present our main algorithm, *Task and Cluster Heterogeneity Aware Partitioning* ***(TCHAP)***. Our algorithm tried to reduce overall system energy by leveraging the different energy profiles of the heterogeneous clusters of ARM big.LITTLE platforms. Furthermore, the algorithm realizes the difference in power consumption between tasks running on the same cluster. We present two variants of **(TCHAP)**; the first algorithm, which we denote as TCHAP$_I$, targets systems where the overhead from switching into sleep modes might affect the timeliness of certain real-time systems. In these systems, DPM is disabled and the processor is kept idling when no tasks are running. We present this algorithm in Section 5.4.3. The second variant, TCHAP$_S$ handles the case where the processor makes use of available sleep states and optimizes the total energy consumption accordingly. We discuss the variant algorithm in Section 5.4.4. The TCHAP algorithm invokes two algorithms, AICF and T-CAFE. The former sets the initial clusters frequencies whereas the latter explores the energy consumption of each task across all available frequencies on the heterogeneous clusters. T-CAFE algorithm generates the set of frequencies for each task at which TCHAP decides whether it is more energy-efficient to allocate the task to the LITTLE or BIG cluster.

5.4.1 Task-Aware Cluster Assignment Frequencies Exploration algorithm (T-CAFE)

Tasks differ in their power profile based on their behavior dictated by their instruction make up. To fully exploit the potential of energy-aware partitioning, it is vital not only to consider the heterogeneous processors power profiles but also demonstrate awareness of the underlying taskset power properties. Before invoking T-CAFE, we perform complete power profiling of all tasks in the system across all clusters at all supported frequencies. We compute the task energies given the scaled worst case execution times of the tasks at the specified frequencies. We show an example of profiling two tasks *bitcnts* and *bubblesort* in Table 5.2. For each task, we show the energy consumed when run on a processor of the LITTLE cluster $e^{\tau_1}_{L,k}$, and BIG cluster $e^{\tau_1}_{B,k}$ at a designated frequencies $f_{L,k}$ and $f_{B,k}$. T-CAFE does not consider frequencies below the critical frequencies. The critical frequencies are shaded in Table 5.2.

One can observe that energy efficiency partitioning on either cluster is dependent on the frequency the cluster is running at. Even though the BIG cluster is generally less energy-efficient, there are certain frequencies which are more energy-efficient especially when the LITTLE cluster is running at high frequencies. For example, running *bitcnts* on the BIG cluster at frequencies between 700-900 MHz costs less energy than running the same task on the LITTLE cluster at frequencies larger than 1200. Yet, if the BIG cluster runs at 1000MHz, it is more efficient to reallocate the task onto the LITTLE cluster.

Given the task energy parameters and cluster frequencies, the Task-Aware Cluster Assignment Frequencies Exploration algorithm (T-CAFE) traverses $e^{\tau_n}_{L,k}$ and $e^{\tau_n}_{B,k}$ in a zig-zag fashion. It looks for a set of "cut-off" frequencies that if a cluster operates at, it will potentially cost less energy to move the task to the next cluster. Our main algorithm TCHAP subsequently uses these frequency sets to determine if reallocation is beneficial.

The T-CAFE algorithm which we present in Algorithm 9 takes as input the arrays e^{Γ}_{L} and e^{Γ}_{B}. The arrays store the energy consumption of each task in the taskset Γ at all supported cluster frequencies $f_{L,k}$ and $f_{B,k}$ **(Lines 1-2)**. The array Ω_{LB} holds the set of the LITTLE cluster frequencies for each task τ_n at which it is more energy-efficient to switch to the BIG cluster. Similarly, Ω_{BL} holds the BIG cluster frequencies at which it is more energy-efficient to switch back the allocation into the LITTLE cluster **(Lines 3-4)**. We initialize the Ω_{LB}, Ω_{BL} arrays to zero and their size is dictated by the number of tasks and supported frequencies of each cluster **(Line 6-7)**. We run the T-CAFE algorithm over all tasks **(Line 8)**.

We point that not all tasks exhibit a similar energy-profile to the tasks in Table 5.2. Some tasks are always energy-efficient when run on the LITTLE cluster. To determine this, we search for any frequency of the LITTLE cluster at which the task's energy consumption $e^{\tau_n}_{L,k}$ is larger than that of the critical frequency of the BIG cluster $e^{\tau_n}_{B,f^{crit}_B}$. If such frequency exists, we store it into *firstSwitchToBig* and its associated index k into j **(Line 9)**. The variable j tracks the indices of cluster frequencies at which we make a switch between clusters. Subsequently, we store this frequency as the first element of the vector in array Ω_{LB} associated with task τ_n **(Lines 16-17)**. We conduct a similar search for the frequency of the BIG cluster $f_{B,k}$ at which it is always more energy-efficient to allocate the task to the LITTLE cluster. This condition holds when the task energy-consumption of the BIG cluster $e^{\tau_n}_{B,k}$ at frequency $f_{B,k}$ is larger than that at the maximum frequency of the LITTLE cluster $e^{\tau_n}_{L,f^{max}_L}$. If such frequency exists, we store it into *lastSwitchToLittle* **(Line 10)**. The variables η_L and η_B are indexing variables which we use to correctly store the frequencies into Ω_{LB} and Ω_{BL} arrays. We reset those variables for each task **(Lines 11)**.

If potential energy-efficient frequencies exist on the big cluster **(Line 12)**, the exploration and population of Ω_{LB} and Ω_{BL} commences. Once the first frequency is stored and the index η_L updated, the algorithms looks for the frequency of the BIG cluster $f_{B,k}$ at which the task energy $e^{\tau_n}_{B,k}$ is larger than that at which we decided to switch to the BIG cluster $e^{\tau_n}_{L,j}$ **(Line 20)**. We store this frequency ω_B as the first element of the vector in array Ω_{BL} associated with task τ_n **(Lines 21-23)** . We update η_B to correctly store subsequent frequencies in the array. The algorithm proceeds similarly in a zig-zag fashion by finding and storing the next frequency of the small cluster ω_L at which we switch to the BIG cluster **(Lines 24, 14-15, 19)**. The algorithm terminates once we reach the last frequency ω_B at which we switch to the LITTLE cluster; this is satisfied when ω_B reaches *lastSwitchToLittle* **(Line 26)**.

To demonstrate, if we execute the T-CAFE algorithm over the tasks in Table 5.2, the contents of Ω_{BL} and Ω_{LB} will be:

$$\Omega_{LB} = \begin{bmatrix} 1200 & 1300 & 1400 \\ 1300 & 1400 & - \end{bmatrix} \Omega_{BL} = \begin{bmatrix} 1000 & 1100 & 1200 \\ 1000 & 1100 & - \end{bmatrix}$$

For the **bitcnts** task, once the LITTLE cluster reaches a frequency of 1200 MHz, we prefer to alloacte it onto the BIG cluster as long as its operating frequency is below 1000 MHz. If it is 1000 MHz or over, we prefer to allocate the task back onto the LITTLE cluster and so on.

Table 5.2 Energy Profiling of *bitcnts* and *bubblesort* on Samsung 5422 ARM Big.LITTLE SoC

bitcnts				**bubblesort**			
$f_{L,k}$ *MHz*	$e^{\tau 1}_{L,k}$ *J*	$e^{\tau 1}_{B,k}$ *J*	$f_{B,k}$ *MHz*	$f_{L,k}$ *MHz*	$e^{\tau 2}_{L,k}$ *J*	$e^{\tau 2}_{B,k}$ *J*	$f_{B,k}$ *MHz*
200	0.235	0.472	200	200	0.182	0.403	200
300	0.213	0.418	300	300	0.166	0.356	300
400	0.204	0.364	400	400	0.160	0.335	400
500	0.213	0.381	500	500	0.155	0.323	500
600	0.210	0.373	600	600	0.166	0.319	600
700	0.226			700	0.179		
800	0.244			800	0.191		
900	0.295			900	0.237		
1000	0.321			1000	0.258		
1100	0.348			1100	0.279		
		0.356	700	1200	0.304		
		0.365	800			0.314	700
		0.375	900			0.310	800
1200	0.381					0.328	900
		0.404	1000	1300	0.335		
1300	0.414					0.344	1000
		0.425	1100	1400	0.353		
1400	0.444					0.379	1100
		0.451	1200			0.395	1200
		0.443	1300			0.375	1300
		0.459	1400			0.377	1400
		0.495	1500			0.425	1500
		0.483	1600			0.455	1600
		0.528	1700			0.507	1700
		0.596	1800			0.533	1800
		0.670	1900			0.576	1900
		0.728	2000			0.664	2000

Algorithm 9 Task-Aware Cluster Assignment Frequencies Exploration (T-CAFE)

```
1: Input:
2: $e_L^\Gamma$, $e_B^\Gamma$, $f_L$ and $f_B$.        ▷ $e_{L,k}^{\tau_n}, e_{B,k}^{\tau_n}$ are elements of the arrays $e_L^\Gamma$, $e_B^\Gamma$ at frequency $k$
3: Output:
4: $\Omega_{BL}$, $\Omega_{LB}$  ▷ Arrays which holds cluster frequencies at which $\tau_n$ is potentially assigned to the other cluster.
5: Initialization:
6: $\Omega_{BL} \leftarrow$ zero initialized $N \times n_B$ array
7: $\Omega_{LB} \leftarrow$ zero initialized $N \times n_L$ array
8: for $i \leftarrow 1, N$ do
9:     [firstSwitchToBig, $j$] $\leftarrow$ find first $f_{L,k}$ and index $k$ s.t $e_{L,k}^{\tau_n} > e_{B,f_B^{crit}}^{\tau_n}$
10:    lastSwitchToLittle $\leftarrow$ find $f_{B,k}$ s.t $e_{B,k}^{\tau_n} > e_{L,f_L^{max}}^{\tau_n}$
11:    $\eta_L \leftarrow 1, \eta_B \leftarrow 1$
12:    if firstSwitchToBig $\neq \phi$ then
13:        while true do
14:            if $\eta_L > 1$ then
15:                $\Omega_{LB}(i, \eta_L) = \omega_L$
16:            else if $\eta_L = 1$ then
17:                $\Omega_{LB}(i, \eta_L) \leftarrow$ firstSwitchToBig
18:            end if
19:            $\eta_L \leftarrow \eta_L + 1$
20:            $[\omega_B, j] \leftarrow$ find $f_{B,k}$ and index $k$ s.t $e_{B,k}^{\tau_n} > e_{L,j}^{\tau_n}$
21:            if $\omega_B \neq \phi$ then
22:                $\Omega_{BL}(i, \eta_B) \leftarrow \omega_B$
23:                $\eta_B \leftarrow \eta_B + 1$
24:                $[\omega_L, j] \leftarrow$ first $f_{L,k}$ and index $k$ s.t $e_{L,k}^{\tau_n} > e_{B,j}^{\tau_n}$
25:            end if
26:            if $\omega_B =$ lastSwitchToLittle then
27:                break
28:            end if
29:        end while
30:    end if
31: end for
```

5.4.2 Assign Initial Clusters Frequencies (AICF)

The aim of the ACIF algorithm is to find a frequency for the LITTLE cluster at which we are able to fit the most tasks. At low utilization, it is likely that all tasks will fit within the LITTLE cluster. We present this algorithm in Algorithm 10.

The AICF algorithm takes as input the set of tasks Γ alongside the supported cluster frequencies f_L and f_B ordered in increasing value **(Lines 1-2)**. Since we only consider energy-efficient frequencies, all frequencies below the critical frequency are discarded **(Line**

8). The algorithm resumes with all remaining frequencies $\tilde{f}_L$ **(Line 9)**. For each frequency in $\tilde{f}_L$, we compute the scaling factor S_L by dividing the maximum cluster frequencies over the other frequencies **(Line 11)**. We proceed by accumulating the scaled utilization of the taskset as long as it fits within the total cluster capacity. The cluster capacity equals the number of cores within the cluster Q_L. We update the number of tasks we can fit into the LITTLE cluster for all frequencies in $\tilde{f}_L$ and store them in Max_L **(Lines 10-20)**. The entries of Max_L will be in non-decreasing order. We pick the *first* maximum and store it in max_τ. The reason we emphasize *first* is due to the possibility that multiple frequencies in $\tilde{f}_L$ might fit the same number of tasks. We opt for the first maximum because it corresponds to the lower frequency; henceforth, lower energy footprint. We designate this frequency as our initial frequency and assign it to ω_L. The BIG cluster initial frequency ω_B starts at the critical frequency of the BIG cluster **(Line 21-23)**. Finally, we further retain the next frequency in $\tilde{f}_L$ that is larger than ω_L. This proves beneficial in a corner case of the TCHAP$_I$ algorithm.

Algorithm 10 Assign Initial Clusters Frequencies (AICF)

```
1: Input:
2: A set of tasks $\Gamma$ of size $N$, the frequency sets for both LITTLE and BIG clusters $f_L$ and $f_B$ in
   increasing order.
3: Output:
4: $\omega_L, \ddot{w}_L$, and $\omega_B$          ▷ Initial frequencies for the LITTLE and BIG clusters
5: $max_\tau$                                      ▷ Maximum tasks that fit LITTLE cluster at $\omega_L$
6: BEGIN:
7: $Max_L = \{\}$
8: $\tilde{f}_L \leftarrow (f_L^{crit} \;..\; f_L^{max})$
9: for $i \leftarrow 1, length(\tilde{f}_L)$ do
10:     sum $\leftarrow 0$
11:     $S_L \leftarrow \{f_L^{max} / \tilde{f}_{L,i}\}$
12:     for $j \leftarrow 1, N$ do
13:         $sum \leftarrow sum + u_{L,j} \times S_L$
14:         if $sum > Q_L$ then
15:             break
16:         else
17:             $Max_L(i) \leftarrow j$
18:         end if
19:     end for
20: end for
21: $max_\tau \leftarrow$ first maximum value in $Max_L$ vector when sequentially traversed from lowest index
22: $\omega_L \leftarrow$ the frequency $f_{L,k} \in \tilde{f}_L$ that corresponds to $max_\tau$
23: $\omega_B \leftarrow f_B^{crit}$
24: $\ddot{w}_L \leftarrow$ Next frequency in $f_L$ larger than $\omega_L$
```

5.4.3 Task and Cluster Heterogeneity Aware Partitioning (optimize for idle version) (TCHAP$_I$)

Our main algorithm which we present in Algorithm 11 takes the same inputs as the AICF and T-CAFE algorithms and produces - if feasible - task partitioning Θ_L and Θ_B which hold the individual task assignments for each core within the cluster. The taskset Γ is ordered in non-increasing order of utilization (for the LITTLE cluster). We start by invoking the *T-CAFE* algorithm to obtain the sets for inter-cluster switching frequencies for each task **(Line 8)**. We initialize the cluster frequencies by calling the *AICF* algorithm once **(Lines 11-13)**. Initially, the task partitioning arrays are empty. No tasks are given priority to be scheduled on the BIG cluster (ρ_B is empty). We further initialize flags that are frequently updated to control the execution path of the algorithm **(Lines 14-15)**.

In this version of the TCHAP algorithm where we consider idling states instead of sleep states, we skip prioritizing tasks to run on the BIG cluster if we are able to fit all tasks on the LITTLE cluster. The rationale behind this stems from our experimental work. We observed that the cost of activating a BIG core with idle intervals overcomes the energy-efficiency differential when assigning few tasks to the BIG core. We encounter this corner case when the system utilization is quite low. In most cases where tasks would only fit by utilizing the capacity of both clusters, task prioritization takes place. For every task in the system, and for the currently assigned cluster frequencies ω_L and ω_B, we consult the inter-cluster switching frequencies Ω_{LB} and Ω_{BL}. If the task is more energy-efficient to run on the BIG cluster, it is added to the priority queue ρ_B **(Lines 16-26)**. This block will be invoked any time the cluster frequencies are updated throughout the execution of the algorithm. To clarify, suppose the current frequencies are ($\omega_L = 1200, \omega_B = 800$), in this case we prioritize the task *bitcnts* to run on the BIG cluster, whereas we schedule *bubblesort* on the LITTLE cluster as long as it is feasible to do so.

We use bin packing algorithms as the underlying scheduling algorithms. We use *Worst Fit Decreasing* ***WFD*** for the LITTLE cluster. However, we opt for *Best Fit Decreasing* ***BFD*** for the BIG cluster. We select BFD to minimize the number of active cores in the BIG cluster and therefore reduce the energy dissipated when idling. With less active cores needed, we can completely shutdown unused cores for further energy-minimization. For every task that is not prioritized to run on the BIG cluster, we attempt to schedule it on the LITTLE cluster. If successful, we update the assigned core total utilization. If we are not able to schedule it on the LITTLE cluster (i.e. lack of available capacity), we append the task to the priority queue ρ_B **(Lines 30-35)**. If the LITTLE cluster is unable to accommodate any of the remaining eligible tasks, we designate it as full **(Lines 33-34)**. We proceed to schedule any task in the priority queue onto the BIG cluster as long as it has available capacity, otherwise, we designate it as full **(Lines 37-40)**.

If both clusters are full and the clusters are at their maximum supported capacities; yet we still have tasks to schedule, we flag the taskset as unfeasible and terminate the algorithm **(Lines 41-45)**. This is one of two exit conditions to terminate the loop **(Line 10)**. However, if both clusters are full, yet we have not explored scheduling at all possible frequencies, we update the current cluster frequencies ω_L and ω_B and restart the algorithm **(Lines 46-54)**. If the algorithm is able to reach a feasible schedule, it terminates **(Lines 63-64)**. Finally, for the corner case where we know that the taskset can be wholly scheduled on the LITTLE cluster, yet some tasks have been assigned to the BIG cluster, we move to the subsequent frequency yielded by the ACIF algorithm **(Lines 55-57)**. This issue arises because the ACIF algorithm considers the total capacity of the cluster whereas the WFD algorithm deals with individual core capacities. As such, some tasks might not fit within the residual core capacities.

5.4.4 Task and Cluster Heterogeneity Aware Partitioning (optimize for sleep version) (TCHAP$_S$)

This variant of the TCHAP algorithm introduces minor changes to the original algorithm. Since we make use of sleep states instead of idling, we are no longer bounded to fit all tasks on the LITTLE cluster whenever it is possible. We can make full use of prioritizing tasks to run on the BIG cluster without the diminishing returns of idle power. Furthermore, we revert to using WFD for the BIG cluster for it yields better schedulability. We highlight the affected lines and changes to the main TCHAP algorithm in Algorithm 12.

5.5 Experimental Setup

In this chapter, we use the same tasks which we used in Chapter 4. We select 45 tasks from the 50 tasks in Table 4.3 as we exclude tasks with pWCET of over 1000ms. We do so such that we can build a discrete period set from all the factors of a highly composite number (110,880) which translates to a hyper-period of 110.88 seconds ($\approx$ 1.6 minutes) and reduce the evaluation time on the hardware.

To assess the performance and feasibility of both our and the reference HIT-LTF algorithm, we run the algorithms over utilization points that range from low to high system load. Though it is easy to prepare tasksets which yield exact total utilizations in uniprocessor and

Algorithm 11 **Task and Cluster Heterogeneity Aware Partitioning (optimize for idle version) (TCHAP$_I$)**

```
 1: Input:
 2: A set of tasks $\Gamma$ of size $N$.
 3: $f_L$, $f_B$: frequency sets for LITTLE & BIG clusters in increasing order.
 4: $e_L^\Gamma, e_B^\Gamma$: energy consumption arrays of taskset $\Gamma$ at all frequencies.
 5: Output:
 6: $\Theta_L, \Theta_B$ A feasible task partitioning, if possible.
 7: Begin:
 8: $\{\Omega_{BL}, \Omega_{LB}\} \leftarrow$ Execute T-CAFE
 9: retry $\leftarrow 0$, feasible $\leftarrow 1$
10: while feasible do
11:     if retry $= 0$ then
12:         $\{\omega_L, \ddot{w}_L, \omega_B, max_\tau\} \leftarrow$ Execute AICF algorithm
13:     end if
14:     valid $\leftarrow 1$, $Full_L \leftarrow 0$, $Full_B \leftarrow 0$
15:     $\rho_B \leftarrow \{\}, \Theta_L \leftarrow \{\}, \Theta_B \leftarrow \{\}$
16:     if $max_\tau \neq N$ then
17:         for $i \leftarrow 1, N$ do
18:             $k \leftarrow k^{th}$ index of first $\Omega_{LB}(i,k) \geq \omega_L$
19:             $l \leftarrow l^{th}$ index of first $\Omega_{BL}(i,l) \geq \omega_B$
20:             if $l \neq \phi$ & $k \neq \phi$ then
21:                 if $\omega_L \geq \Omega_{LB}(i,k)$ & $\omega_B < \Omega_{BL}(i,l)$ then
22:                     $\rho_B \leftarrow \{\rho_B, \tau_i\}$
23:                 end if
24:             end if
25:         end for
26:     end if
    .. conitinued on next page
```

homogeneous multicore systems; this is further complicated in heterogeneous systems for we don't know *a priori* where each task will be allocated. To solve this, we introduce the stress factor $\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$. We define the target system utilization as:

$$U_T = (Q_L + Q_B)(1 + \alpha)(U_p) \tag{5.5}$$

where Q_L, Q_B are the number of cores in the LITTLE and BIG clusters, U_p denotes a percentage which falls within [0.1 - 0.9] in steps of 0.05 resulting in 17 initial simulation points (at $\alpha = 0$). Assigning the other α values extends the number of simulation points. To reduce the number of simulations required, we remove redundant or close total utilizations. The lowest and highest resulting system utilization points are 0.8 and 15.6, respectively. Now that we know the total utilizations points, we generate *initial* individual task utilizations

```
27:     for $i \leftarrow 1, N$ do
28:         if $valid = 1$ then
29:             retry $\leftarrow 0$
30:             if $\tau_i \notin \rho_B$ then
31:                 Try to schedule $\tau_i$ on $\kappa_L$ by WFD, update $U_{L,j}$
32:                 if $\tau_i \notin \Theta_L$ then $\rho_B \leftarrow \{\rho_B, \tau_i\}$ end if
33:                 if $(1 - U_{L,j}) > u_{L,n}, \forall j \in [1..Q_L], \forall n > i$ then
34:                     $Full_L \leftarrow 1$
35:                 end if
36:             end if
37:             if $\tau_i \in \rho_B$ then
38:                 Try to schedule $\tau_i$ on $\kappa_L$ by BFD, update $U_{B,j}$
39:                 if $\tau_i \notin \Theta_B$ then $Full_B \leftarrow 1$ end if
40:             end if
41:             if $Full_B$ & $Full_L$ & $(\omega_L = f_L^{max})$ &
                $(\omega_B = f_B^{max})$ then
42:                 feasible $\leftarrow 0$, valid $\leftarrow 0$
43:                 $\Theta_L \leftarrow \{\}$, $\Theta_B \leftarrow \{\}$
44:                 break
45:             end if
46:             if $Full_B$ & $Full_L$ then
47:                 if $\omega_L < f_L^{max}$ then
48:                     Given $\omega_L = f_{L,k}$, the new $\omega_L \leftarrow f_{L,k+1}$
49:                 else if $\omega_B < f_B^{max}$ then
50:                     Given $\omega_B = f_{B,k}$, the new $\omega_B \leftarrow f_{B,k+1}$
51:                     if $\omega_B \leq f_B^{max}$ then $\omega_L \leftarrow f_L^{crit}$ end if
52:                 end if
53:                 valid $\leftarrow 0$, retry $\leftarrow 1$
54:             end if
55:             if $max_\tau = N$ & $\exists \tau_n \in \Theta_B$ then
56:                 $\omega_L \leftarrow \ddot{\omega}_L$, valid $\leftarrow 0$, retry $\leftarrow 1$
57:             end if
58:         else
59:             break
60:         end if
61:     end for
62:     if $\forall \tau_n \in \Gamma, \tau_n \in \Theta_L \mid\mid \tau_n \in \Theta_B$ then
63:         break
64:     end if
65: end while
```

that sum up to the total using the *randfixedsum* algorithm [160]. To pair task WCETs with discrete periods yet match the task and system utilizations, we employ the CPA-AU algorithm that we introduced in Section 4.5.2. In applying the CPA-AU algorithm, we use

Algorithm 12 **Task and Cluster Heterogeneity Aware Partitioning (optimize for sleep version) (TCHAP$_S$)**

```
1: Remove lines 16 and 26 in TCHAPS$_I$, keep the body in between
2: Remove lines 55 through 57 in TCHAPS$_I$
3: Change the scheduling algorithm for BIG Cluster from BFD to WFD
```

task WCETs based on the LITTLE cluster. For each utilization point, we repeat this 750 times to generate 750 tasksets.

To the best of our knowledge, we do not know of any tool, methodology, or hardware support that enables us to measure task power at the job level such that we can observe the effects of multicore interference on actual job run-time and energy consumption. To this end, for the purpose of running our partitioning algorithms, we start with measuring task power without considering the effects of interference. We run each task back to back on one core of each cluster for a duration of one minute and measured the power at intervals of 200ms. We average the power values for each task. We repeat this procedure for all supported cluster frequencies. This approach was similarly taken in [221] for building their power model of the same processor. Given the tasks power matrix, and the task estimated pWCETs, we compute the energy-consumption arrays e_L^Γ and e_B^Γ. However, when we measure the taskset energy consumption on real-hardware, the effects of resource sharing, contention, and interference will be taken into account in the final measurement.

To run our tasksets in real-time on the XU3 platform, we first patched the 3.10.y Linux kernel with the Litmus-RT patch (refer to Section 2.1.4). For this experiment, we used the partitioned-EDF scheduler (P-EDF) supported by Litmus-RT. We wrap each task with a Litmus-RT API wrapper to define and launch it as a periodic real-time task. We use the task parameters obtained through the CPA-AU algorithm for the task definition. From the final partitioning Θ_L and Θ_B that we obtained for the TCHAP$_I$, TCHAP$_S$, and HIT-LTF algorithms, we assign core affinities to the tasks. We release the tasks synchronously and we run the schedule for the specified simulation time. We measure the power consumption using the on-board sensors connected to each cluster at intervals of 200ms. Then we compute the total energy-consumption of each taskset and aggregate the results from both clusters.

Even though our algorithms rely on the EDF schedulability test to check for the feasibility of each core's schedule, this test highly depends on the estimated pWCETs. Our initial assumption is that the EVT-based MBPTA estimated pWCETs do include the effects of interference in the underlying multicore system. Yet, given that we are running our algorithms on real-hardware, before evaluating the efficacy of our algorithms in system energy reduction, we conduct feasibility tests on the tasksets to check that they are feasible on hardware despite the scheduler overhead and multicore interference. We automate the run of each taskset through Litmus-RT "st-trace-schedule" tool and observe that none of the jobs miss their deadline.

It is important to note that on the XU3 board, when the processor is idle, the scheduler invokes the Exynos *cpuidle* driver which in turns invokes ARMs *WFI* instruction that puts the processor into sleep mode. So by default, the idling state is a sleep state. We use this mode when evaluating the TCHAP$_S$ algorithm. In order to evaluate the TCHAP$_I$ algorithm, we build a Linux kernel with the configuration option "CONFIG_CPU_IDLE=n" such that the scheduler will not call the default implementation of the *cpuidle* driver that puts the processor to sleep. Instead, this will keep the processor idling as intended.

5.6 Experimental Results

In this section, we present our results against the latest research that we know of that utilizes the same platform and system model. We compare our results to the HIT-LTF algorithm [148]. We normalize our algorithms results against HIT-LTF to highlight the energy savings we obtained.

We show the results for the first variant of our algorithm TCHAP$_I$ in Fig. 5.6. We note that for low system utilization (U $\leq$ 4), both algorithms exhibit the same energy performance. This is because it is possible to fit all tasks on the LITTLE cluster and we deliberately avoid assigning tasks to the BIG cluster due to the diminishing returns of the idle power of the BIG cluster. For this case, our algorithm and the HIT-LTF yield the exact partitioning schedule. This is because the LTF heuristic is basically a WFD heuristic. However, once the load starts to increase (4 $<$ U $\leq$ 9), the TCHAP$_I$ algorithm yields energy savings between 13% and 23% on average compared to HIT-LTF with narrow variance range. Once the processor gets heavily loaded, our algorithm yields between 5% and 10% more energy savings. This drop in energy savings is expected. At higher utilization, it is fundamental to maintain a schedulable system. The TCHAP$_I$ algorithm responds to the high load by increasing the cluster frequencies to accommodate all tasks, if possible. The energy savings from running clusters at lower frequencies are no longer viable. We attribute the energy-savings at high utilization to the merits of our task-aware allocation and prioritization scheme.

The results for our second variant of the TCHAP algorithm show similar energy-savings pattern across the system utilization points. We illustrate these results in Fig. 5.7. However, we emphasize a notable difference at lower utilization levels; in particular, for the case (2 $<$ U

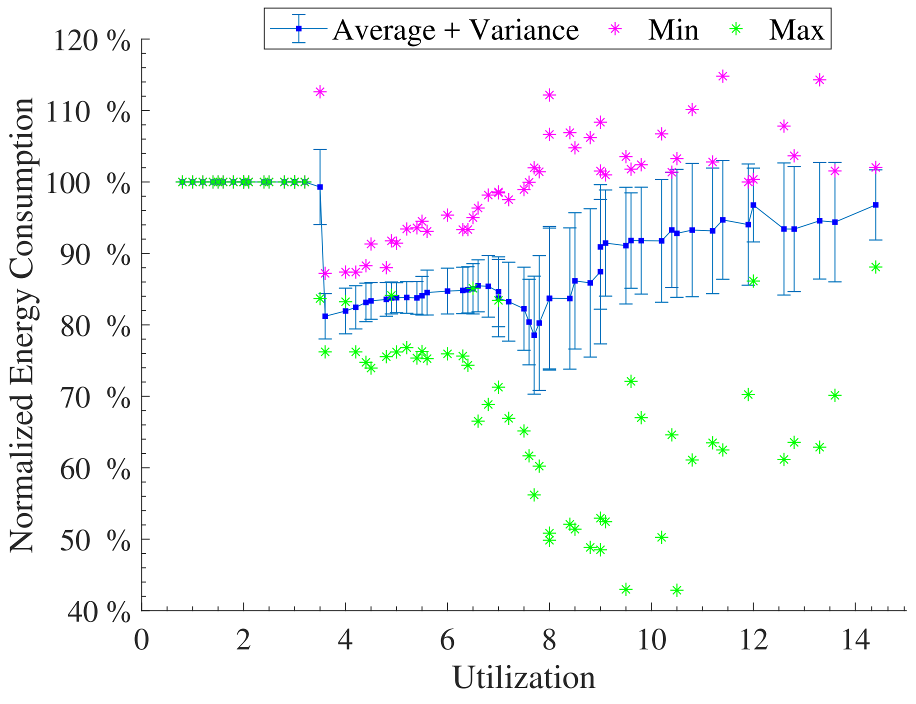


Figure 5.6 Normalized energy consumption for the TCHAP$_I$ algorithm compared to HIT-LTF (The processor remains idle in both cases)

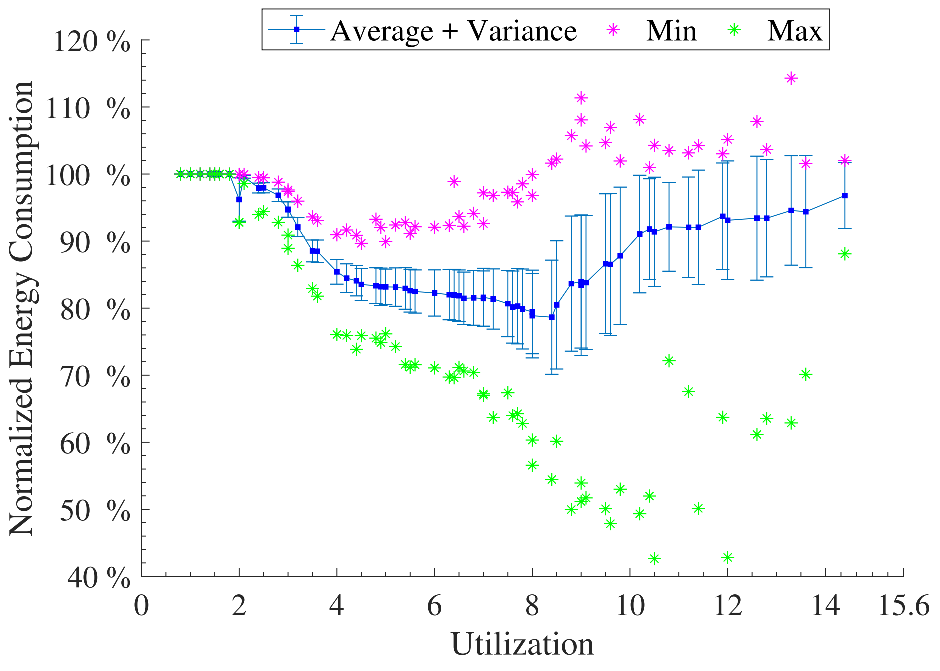


Figure 5.7 Normalized energy consumption for the TCHAP$_S$ algorithm compared to HIT-LTF (The processor utilizes sleep states in both cases)

Table 5.3 Percentage of tasksets delivering better energy savings than HIT-LTF

	Energy Savings Compared to HIT-LTF							
Algorithms	<0	0%	10%	15%	20%	25%	30%	35%
TCHAP$_I$	16.60%	83.40%	52.31%	35.68%	12.14%	3.43%	1.27%	0.53%
TCHAP$_S$	4.85%	95.15%	62.27%	42.76%	16.30%	4.78%	1.52%	0.56%

≤ 4). Whereas the TCHAP$_I$ algorithm matched the HIT-LTF results for the idle processor experiments at low utilization, when we allow the processor to sleep for both algorithms, our variant progressively achieves more energy savings. This is due to the fact that the TCHAP$_S$ avoids aggressively fitting all tasks onto the smaller cluster. Instead, task-aware allocation prioritization is allowed to run without the risk of incurring high energy penalty once the BIG cluster is activated.

Given that the variance can be affected by outliers, we show in Table 5.3 the percentage of all tasksets that have more energy savings over a certain percentage threshold compared to HIT-LTF. We observe that TCHAP$_I$ algorithm delivers better energy savings over HIT-LTF 83.4% of the time. With 52.31% of the tasksets having at least 10% more energy savings and more than third of the tasksets delivering over 15% energy savings across all system utilization points. Similarly, when we invoke our TCHAP$_S$ algorithm, no less than 95.15% of tasksets yield better energy savings. Around third of these savings are less than 10%, 19.41% of the tasksets yield energy savings between 10% and 15% and about a quarter between 15% and 20%.

Finally, we compare the feasibility metric between the TCHAP and HIT-LTF algorithms in Fig. 5.8. At high utilization, once the ability to schedule tasksets starts to decline, our algorithm maintains around 5% to 10% marginal schedulability advantage at $13 < \text{U} \leq 14$ over HIT-LTF. As expected, pushing more capacity ($\text{U} > 14$) further curtails feasibility.

5.7 Chapter Summary

ARM big.LITTLE processors are ubiquitous in embedded heterogeneous computing. SoCs employing ARM's designs dominate the mobile computing industry due to their energy-efficiency. Many ARM based cores are increasingly used in real-time, cyber-physical and multi-criticality systems. To this end, and leveraging upon the energy-reduction technologies supported by big.LITTLE platforms, we tackled the issue at the partitioning and scheduling level to provide further energy-reductions. We proposed an energy-efficient partitioning

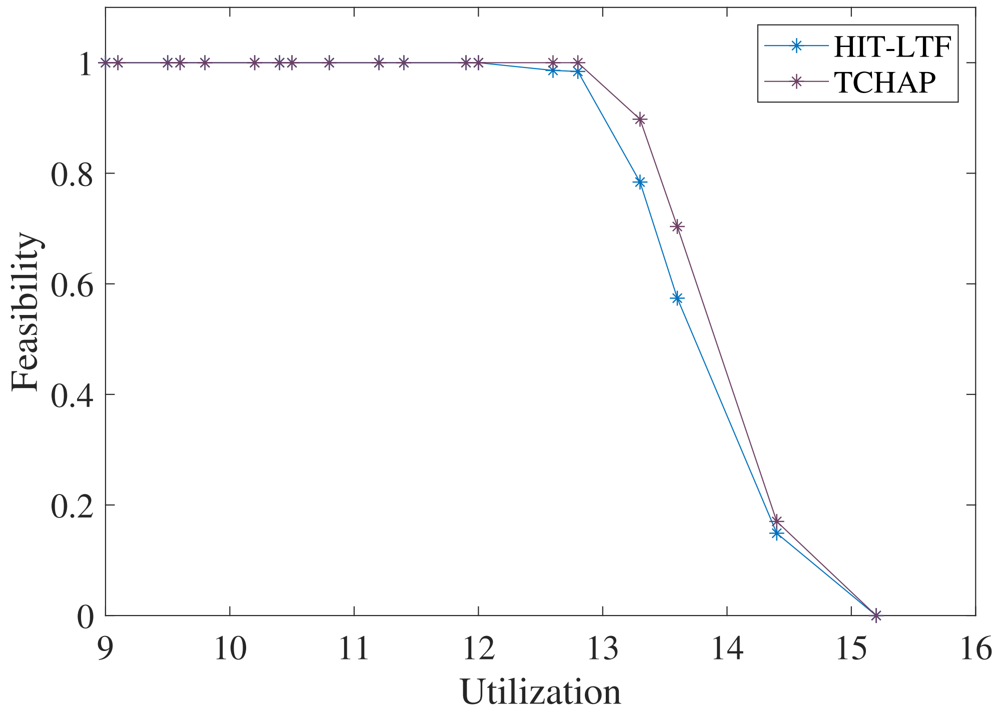


Figure 5.8 Feasibility of the TCHAP and HIT-LTF algorithms at high utilization

algorithm that is aware of the underlying cluster heterogeneity and performance / energy variance. We further considered the different power requirements for the system tasks thus extending our heterogeneity-aware algorithm to be equally task-aware. We presented two variants of our algorithm; TCHAP$_I$ and TCHAP$_S$. The former optimizes energy consumption when it is not feasible to use sleep modes in real-time scenarios where overheads affects the timeliness of the schedule. The latter optimizes for the case where we are able to make use of DPM techniques.

We conducted our evaluation of the algorithms directly on the hardware using a Linux OS patched with real-time extensions. We used the methodology of the previous chapter that facilitates the evaluation of real-time systems at multiple task load points. We estimated the task WCETs on a multicore platform using probabilistic approaches as has been suggested by recent literature. We constructed bounded tasksets for the evaluation of our algorithms in reasonable time. We showed that for both variants of the TCHAP algorithm, we were able to achieve on average between 13% and 23% energy reductions compared to HIT-LTF when the system is under medium load. We also showed that 52.31% and 62.27% of the tasksets have at least 10% more energy savings than HIT-LTF across all utilization points for

TCHAP$_I$ and TCHAP$_S$ algorithms, respectively, Our algorithms have also yielded slightly better feasibility under high system load.

Our approach of direct evaluation on hardware allows us to avoid the shortcomings of simulators (e.g. model simplifications, no task-awareness … etc). However, the accuracy of direct evaluation on hardware is subject to the tight estimation of WCETs. Our algorithms will benefit from future advancements in the field of WCET estimation on multicore platforms and can also be further fine-tuned and refined once literature and industry finds a reasonable way to measure task energy consumption at the job level, a metric highly affected by the hardware design. As ARM and other embedded processor manufacturers keep introducing novel architectural innovations that maximize processor isolation, determinism, or facilitate tight and safe WCET estimation, this will aid in better energy-efficient scheduling algorithms and the possibility of using COTS processors in future embedded real-time designs.

Chapter 6

Conclusions and Future Work

This chapter summarizes the work that we presented in this thesis and discusses some implications and limitations. It further presents few suggestions for future work.

6.1 Conclusions

The power consumption of computing systems in general, and embedded systems in particular has been a prime focus of industry and research. Advancements in processor fabrication and circuit design technologies allow for further performance improvements and lower energy consumption. The use of DVFS hardware exploits the relationship between power-consumption and the cores frequency and voltage to minimize energy costs. Support for multiple sleep modes further cuts the total energy-consumption. Most operating systems are equipped with the ability to directly control DVFS an DPM hardware. Consequently, energy-aware scheduling has become a major approach for fine-tuning and attempting to minimize overall system energy consumption. However, the use of DVFS does not necessarily translate to energy-reductions. The extended execution time due to slowed down core frequency increases leakage power, and certain frequencies are energy-inefficient. In addition, deeper sleep modes incur power and performance overheads.

These issues are further complicated when energy-aware scheduling is performed within the context of real-time systems. The system timeliness and task deadlines impose rigid constraints to the extent on which underlying hardware technologies can be employed to efficiently reduce the system energy. Due to the NP-hard complexity of the scheduling problem, this necessitates proposing solutions based on heuristic or mathematical approaches. This thesis proposed a set of algorithms to reduce energy consumption on DVFS-capable uni-core and heterogeneous multicore processors as well as a methodology and a set of algorithms that aid in constructing tasksets for evaluation on real hardware.

In the third chapter, the focus was real-time embedded systems that use a single core processor with DVFS and DPM capabilities. The case considered involved system tasks with an associated set of external devices. As such, the chapter addressed the issue of system energy-reduction concerning both processor and devices using metaheuristics and we showed that a micro versions of the genetic algorithm and differential evolution reach energy-savings to within 1% of the optimal. However, the analysis was conducted offline based on worst case estimates of the execution times. This is due to high variability of the execution times, and due to the high timing cost of evaluating the objective function if the system was run online. Online algorithms would be a better choice to use additional slack time to harness more energy savings. The device model used in this early research is quite simple and assumes that devices remain in active state during the execution of the associated task. This could be impractical as device access and behaviour might be different in real scenarios. Future work could benefit from recent device modelling approaches.

Even though literature relies heavily on simulation based analysis for proposed solutions, this approach is not without limitations as it is subject to the validity of the model against the hardware. Researchers often try to corroborate their results through running case studies on real hardware, a straightforward approach for generic and non real-time computing. A difficult tasks for real-time systems evaluation. The methodology in literature used in such cases is at best ambiguous. To address this, the fourth chapter proposed a methodology that facilitates evaluating real-time tasks on embedded hardware. The proposed methodology allows for evaluating the tasks at multiple system load points. It further helps in overcoming the shortcomings of simulation based analysis as it allows researchers to assess and handle real-life issues often encountered in industrial deployments. We based our methodology on using publicly available embedded benchmarks with realistic embedded applicability close to industrial and commercial applications. This work built on previous research that selects real-time parameters such as task periods and utilizations. The proposed algorithms specify the real-time taskset parameters that are assigned to real tasks which can be used to evaluate the taskset in a real-time operating system on real-hardware.

The approach undertaken is a bottom-up approach where we start from estimating the tasks' WCETs and build the taskset properties around them. This is in contrast to conventional techniques which start with generating taskset properties according to certain statistical distributions and then synthesize tasks to match the WCET property. Though the latter approach is easy, it cannot be extended to multicore platforms whereas our approach is applicable to any platform. A heavy downside; however, is the overhead cost incurred during the WCET estimation phase and the trust that we base on the estimate tightness and safety. Automating the task run and analysis through configurable scripts will simplify the procedure for approaches relying on MBTA or MBPTA.

Finally, the fifth chapter addressed the issue of energy-aware partitioning and scheduling on embedded processors with clustered heterogeneous cores. Performance and energy variations between processor cores add further dimensions to the optimization problem. The thesis addressed two operational cases; the first when the system is capable of switching to sleep modes between task runs without incurring much overhead that affects the timeliness of the system; the other is when the processor is kept idling between tasks.

To improve on literature, this work considered the issue of task-awareness. Whereas most literature assume that tasks consume the same power when running on the same type of core at the same specific frequency, this work factored in the task power variance into the energy-aware partitioning problem. This thesis further proposed algorithms that initialize the cluster frequencies, produce a set of in-cluster transition frequencies, and partition the tasks into heterogeneous clusters. This thesis considered the ARM big.LITTLE platforms with two clusters as they are the most pervasive ICs for embedded heterogeneous multicores. The recently announced ARM big.LITTLE DynamiQ allows for the use of a third "MEDIUM power" cluster. The announced products based on this newer revision have the MEDIUM cluster identical to the BIG cluster but with its cores capped at lower frequencies (e.g. Kirin 980). The technical details of the new processors are not available at the time of writing this thesis. However, should the MEDIUM cluster share a subset of the frequencies of the BIG cluster, our algorithms are easily adapted to cover three clusters. If future generations of DynamiQ chipsets have one or more MEDIUM clusters with different performance and power profiles than either of the BIG or LITTLE clusters, then our algorithms execution time will exponentially increase. As such, a frequency pruning step might be needed to limit the frequency search space of our TCHAP algorithm.

6.2 Future Work

There are many opportunities for research in energy-aware partitioning and scheduling. We present possible venues for future work to extend this thesis.

6.2.1 Enhancing Real-Time Simulators with Task and Thermal Awareness

Similar to most literature, the event-driven simulator that we built in Chapter 3 is primarily task-agnostic. The modules used to generate and launch the tasks with real-time parameters lacked any task-specific power consumption specification. Another project involves extending the simulator task modules to account for the differences in task power consumption. A naive approach could simply add a power factor to differentiate between tasks. A realistic and more elaborate approach involves modelling task power consumption on real-hardware by measuring the power consumption on various hardware platforms at multiple frequency levels. By taking into account the tasks instruction mix and the underlying hardware architecture, the task power consumption can be modelled. The task-aware simulator enhanced modules will take into consideration the target platform and the user-specified instruction mix to assign relative power factors.

Furthermore, in many-core homogeneous or heterogeneous processors, thermal-aware partitioning and scheduling can also be considered to avoid thermal hot spots. That is, assigning heavy loads to certain cores while others remain lightly loaded. Cores under heavy loads will often reach temperatures high enough (or potentially exceed) their thermal design power (TDP) especially when the core is operating at high frequencies. In such cases, dedicated hardware controllers override any core frequencies assigned by the operating system and throttle the affected core speed to reduce core temperature. Another project introduces additional modules to the simulator to model the heat dissipation of many-core processors. Energy-Aware simulation can benefit from more accurate models that closely mirror real-hardware and task behavior.

6.2.2 Energy-Aware Scheduling under the Limited Preemption Model

In chapters 3 and 5, and in lieu with the prevalent literature, we have assumed a fully-preemptive scheduling model. Another project explores possible energy-savings under the limited preemption model based on preemption thresholds. This future work considers energy-aware scheduling under limited preemption constraints for both homogeneous and heterogeneous processors. This involves modifying the Litmus-RT framework to support preemption thresholds and solving the task allocation and threshold assignment concurrently such that the total number of task preemptions as well as total energy is reduced. This work would be valuable in leveraging potential energy savings related to minimizing the use of the underlying bus, cache, and memory subsystems. This project can be paired with research pertaining to cache-aware scheduling and partitioning where reduced cache access and block replacement pave the way to deploy cache energy reduction techniques and reduce the effects of multicore interference on the energy and execution time of jobs.

6.2.3 Enhanced Average and Worst Case Task Energy Estimation

In this thesis and most literature, the power consumed by tasks is mostly measured by running tasks in isolation for a certain amount of time and averaging the results. This approach while simple, does not give any guarantees about the actual energy consumed by tasks at run time especially in complex multicore systems. Interference due to tasks evicting cache lines related to other tasks in shared caches, contention on shared resources, and bus arbitration and access affects the total energy consumed by task instances. An interesting research venue is to provide proper means to measure the average and worst case energy of tasks effectively. Such parameters could prove to be invaluable in designing more refined and task-aware energy-efficient scheduling algorithms.

Bibliography

[1] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*, 1st ed. LeeSehia.org, 2012.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem&mdash;overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[3] "Semiconductor Engineering Website," Nov. 2015. [Online]. Available: https://semiengineering.com/micro-architectural-exploration-for-low-power-design/

[4] A. Colin, A. Kandhalu, and R. R. Rajkumar, "Energy-efficient allocation of real-time applications onto single-isa heterogeneous multi-core processors," *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 91–110, Jul 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-0987-3

[5] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *CoRR*, vol. abs/1401.0765, 2014. [Online]. Available: http://arxiv.org/abs/1401.0765

[6] "Am437x sitara™ processors datasheet," Jan. 2019. [Online]. Available: http://www.ti.com/product/AM4372

[7] "Stm32l series ultra-low-power 32-bit mcu releasing your creativity," Nov. 2018. [Online]. Available: https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/brochure/6c/48/c0/f1/bb/35/4a/b4/brstm32ulp.pdf/files/brstm32ulp.pdf/jcr:content/translations/en.brstm32ulp.pdf

[8] "2017 Embedded Markets Study - Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments," apr 2017. [Online]. Available: https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

[9] "ARM Embedded Segment Market Update," jul 2015. [Online]. Available: https://www.arm.com/zh/files/event/1_2015_ARM_Embedded_Seminar_Richard_York.pdf

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2009, pp. 469–480.

[12] T. Nowatzki, J. Menon, C. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, Nov 2015.

[13] J.-E. Jo, G.-H. Lee, H. Jang, J. Lee, M. Ajdari, and J. Kim, "Diagsim: Systematically diagnosing simulators for healthy simulations," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 4:1–4:27, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3177959

[14] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 577–589.

[15] C. Eichler, T. Distler, P. Ulbrich, P. Wägemann, and W. Schröder-Preikschat, "Taskers: A whole-system generator for benchmarking real-time-system analyses," in *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[16] C.-C. Han and H.-Y. Tyan, "A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, ser. RTSS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 36–. [Online]. Available: http://dl.acm.org/citation.cfm?id=827269.828999

[17] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[18] J. Abella, C. Abella, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "Wcet analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2015, pp. 1–10.

[19] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An open toolbox for adaptive WCET analysis," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems.* Springer, 2010, pp. 35–46.

[20] C. Ferdinand and R. Heckmann, "aiT: Worst-case execution time prediction by static program analysis," in *Building the Information Society.* Springer, 2004, pp. 377–383.

[21] D. Hardy, B. Rouxel, and I. Puaut, "The Heptane Static Worst-Case Execution Time Estimation Tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, vol. 8, 2017, pp. 1–812.

[22] Tidorum-Ltd, "Bound-T time and stack analyser," 2005. [Online]. Available: http://www.bound-t.com/

[23] C. Chen, J. Panerati, I. Hafnaoui, and G. Beltrame, "Static probabilistic timing analysis with a permanent fault detection mechanism," in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2017, pp. 1–10.

[24] C. Chen, L. Santinelli, J. Hugues, and G. Beltrame, "Static probabilistic timing analysis in presence of faults," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–10.

[25] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart, "On the Sustainability of the Extreme Value Theory for WCET Estimation," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), H. Falk, Ed., vol. 39. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 21–30. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4601

[26] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. J. Cazorla, "Measurement-based prob-

abilistic timing analysis for multi-path programs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 91–101.

[27] G. Lima, D. Dias, and E. Barros, "Extreme value theory for estimating task execution time bounds: A careful look," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 200–211.

[28] J. P. Hansen, S. A. Hissam, and G. A. Moreno, "Statistical-based wcet estimation and validation," in *WCET*, 2009.

[29] L. Shen, L. J. Mickley, and E. Gilleland, "Impact of increasing heat waves on u.s. ozone episodes in the 2050s: Results from a multimodel analysis using extreme value theory," *Geophysical Research Letters*, vol. 43, no. 8, pp. 4017–4025, 2016. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016GL068432

[30] F. Longin, *Extreme Events in Finance: A Handbook of Extreme Value Theory and its Applications*. Wiley, 2017. [Online]. Available: https://www.wiley.com/en-us/Extreme+Events+in+Finance%3A+A+Handbook+of+Extreme+Value+Theory+and+its+Applications-p-9781118650196

[31] F. J. Acero, M. C. Gallego, J. A. García, I. G. Usoskin, and J. M. Vaquero, "Extreme value theory applied to the millennial sunspot number series," *The Astrophysical Journal*, vol. 853, no. 1, p. 80, 2018. [Online]. Available: http://stacks.iop.org/0004-637X/853/i=1/a=80

[32] K. P. Silva, L. F. Arcaro, and R. S. d. Oliveira, "On using gev or gumbel models when applying evt for probabilistic wcet estimation," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017, pp. 220–230.

[33] F. J. Cazorla, T. Vardanega, E. Quiñones, and J. Abella, "Upper-bounding program execution time with extreme value theory," in *WCET*, 2013.

[34] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu *et al.*, "Proxima: Improving measurement-based timing analysis through randomisation and probabilistic analysis," in *Digital System Design (DSD), 2016 Euromicro Conference on*. IEEE, 2016, pp. 276–285.

[35] S. Milutinovic, E. Mezzetti, J. Abella, T. Vardanega, and F. J. Cazorla, "On uses of extreme value theory fit for industrial-quality wcet analysis," in *Industrial Embedded Systems (SIES), 2017 12th IEEE International Symposium on*. IEEE, 2017, pp. 1–6.

[36] C. Francis, D. Joëlle, K. Claude, and M. Zoubir, *Scheduling in Real-Time Systems.* John Wiley & Sons, 2003.

[37] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Trans. Comput.*, vol. 47, no. 6, pp. 700–713, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1109/12.689649

[38] J. K. W. Kim and S. L. Min, "quantitative analysis of dynamic voltage scaling algorithms for hard real time systems," in *Proceedings of the SoC Design Conference*, Nov. 2003, pp. 25–34.

[39] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S. L. Min, "Performance evaluation of dynamic voltage scaling algorithms for hard real-time systems," *Journal of Low Power Electronics*, vol. 1, no. 3, pp. 207–216, 2005.

[40] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, 1999, pp. 328–335.

[41] G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[42] C. Dong and H. Zeng, "Minimizing stack memory for hard real-time applications on multicore platforms," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[43] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, 2000, pp. 25–34.

[44] "ThreadX RTOS Website," Sep. 2018. [Online]. Available: https://rtos.com/solutions/threadx/real-time-operating-system/

[45] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

[46] S. K. Dhall and C. L. Liu, “On a real-time scheduling problem,” *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Feb. 1978. [Online]. Available: http://dx.doi.org/10.1287/opre.26.1.127

[47] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

[48] S. Sahni, “Preemptive scheduling with due dates,” *Operations Research*, vol. 27, no. 5, pp. 925–934, 1979.

[49] B. Korte and J. Vygen, “Combinatorial Optimization Theory and Algorithms,” in *Combinatorial Optimization - Theory and Algorithms*, Springer. Springer, 2006.

[50] “Windows CE Website,” Sep. 2018. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/embedded/ee504814(v=winembedded.70)

[51] “ARM RTX5 Website,” Sep. 2018. [Online]. Available: https://arm-software.github.io/CMSIS_5/RTOS2/html/rtx5_impl.html

[52] “MicroC/OS Website,” Sep. 2018. [Online]. Available: https://www.micrium.com/

[53] “FreeRTOS and SafeRTOS Website,” Sep. 2018. [Online]. Available: https://www.freertos.org/

[54] “Vxworks website,” Sep. 2018. [Online]. Available: https://www.windriver.com/products/vxworks/#VxWorks

[55] “Linux SCHED_DEADLINE scheduling algorithm website,” Sep. 2018. [Online]. Available: http://www.evidence.eu.com/sched_deadline.html

[56] “LynxOS Website,” Sep. 2018. [Online]. Available: http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/

[57] “VxWorks Website,” Sep. 2018. [Online]. Available: http://www.rtlinuxfree.com/

[58] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, “Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers,” in *Proceedings of 27th IEEE International Real-Time Systems Symposium, RTSS 2006*, 2006, pp. 111–123.

[59] “LITMUS^{RT} Website,” Oct. 2018. [Online]. Available: https://www.litmus-rt.org/)

[60] M. Chisholm, N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, “Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS ’17. New York, NY, USA: ACM, 2017, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/3139258.3139268

[61] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith, “Allowing shared libraries while supporting hardware isolation in multicore real-time systems,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017, pp. 223–234.

[62] B. B. Brandenburg and M. Gül, “Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016, pp. 99–110.

[63] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, “Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016, pp. 57–68.

[64] D. Compagnin, E. Mezzetti, and T. Vardanega, “Experimental evaluation of optimal schedulers based on partitioned proportionate fairness,” in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 115–126.

[65] N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith, “Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

[66] G. Chen, K. Huang, and A. Knoll, “Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, pp. 111:1–111:21, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2567935

[67] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw, “Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads,” in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design.* ACM, 2002, pp. 721–725.

[68] G. Chen, K. Huang, J. Huang, C. Buckl, and A. Knoll, “Effective online power management with adaptive interplay of DVS and DPM for embedded real-time system,” in *2013 Euromicro Conference on Digital System Design (DSD)*, 2013, pp. 881–889.

[69] J. Kawa and A. Biddle, “FinFET, The Promises and the Challenges,” 2012. [Online]. Available: http://www..com/Company/Publications/SynopsysInsight/Pages/Art2-finfet-challenges-ip-IssQ3-12.aspx

[70] “Intel’s Newsroom:Intel Reinvents Transistors Using New 3-D Structure.” [Online]. Available: http://newsroom.intel.com/community/intel_newsroom/blog/2011/05/04/intel-reinvents-transistors-using-new-3-d-structure

[71] IEEE, “MORE MOORE Whitepaper,” IEEE, Tech. Rep., 2016. [Online]. Available: https://irds.ieee.org/images/files/pdf/2016_MM.pdf

[72] M. A. Hughes, K. P. Homewood, R. J. Curry, Y. Ohno, and T. Mizutani, “An ultra-low leakage current single carbon nanotube diode with split-gate and asymmetric contact geometry,” *Applied Physics Letters*, vol. 103, no. 13, pp. –, 2013. [Online]. Available: http://scitation.aip.org/content/aip/journal/apl/103/13/10.1063/1.4823602

[73] S.-J. Han, J. Tang, B. Kumar, A. Falk, D. Farmer, G. Tulevski, K. Jenkins, A. Afzali, S. Oida, J. Ott, J. Hannon, and W. Haensch, “High-speed logic integrated circuits with solution-processed self-assembled carbon nanotubes,” *Nature Nanotechnology*, vol. 12, p. 861, Jul. 2017. [Online]. Available: http://dx.doi.org/10.1038/nnano.2017.115

[74] J. Tang, Q. Cao, G. Tulevski, K. A. Jenkins, L. Nela, D. B. Farmer, and S.-J. Han, “Flexible CMOS integrated circuits based on carbon nanotubes with sub-10 ns stage delays,” *Nature Electronics*, vol. 1, no. 3, pp. 191–196, Mar. 2018. [Online]. Available: https://doi.org/10.1038/s41928-018-0038-8

[75] C. Qiu, Z. Zhang, M. Xiao, Y. Yang, D. Zhong, and L.-M. Peng, “Scaling carbon nanotube complementary transistors to 5-nm gate lengths,” *Science*, vol. 355, no. 6322, pp. 271–276, 2017. [Online]. Available: http://science.sciencemag.org/content/355/6322/271

[76] D. Yakimets, M. G. Bardon, D. Jang, P. Schuddinck, Y. Sherazi, P. Weckx, K. Miyaguchi, B. Parvais, P. Raghavan, A. Spessot, D. Verkest, and A. Mocuta, “Power aware finfet and lateral nanosheet fet targeting for 3nm cmos technology,”

in *2017 IEEE International Electron Devices Meeting (IEDM)*, Dec 2017, pp. 20.4.1–20.4.4.

[77] M. Mayberry, "Pushing past the frontiers of technology," March 2013. [Online]. Available: http://www.nist.gov/pml/div683/conference/upload/Mayberry_final.pdf

[78] S. Mittal, "A survey of techniques for designing and managing cpu register file," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, p. e3906, 2017.

[79] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "An energy-efficient gpgpu register file architecture using racetrack memory," *IEEE Transactions on Computers*, no. 9, pp. 1478–1490, 2017.

[80] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: Energy efficient partitioned register file for gpus," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on.* IEEE, 2017, pp. 589–600.

[81] A. Aggarwal, R. Segelken, and P. Wasson, "Branch prediction power reduction," U.S. Patent US9 547 358B2, 2017.

[82] A. Peter Greenhalgh, "big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7," 2011.

[83] "ARM Big.Little Website," 2014. [Online]. Available: http://www.thinkbiglittle.com/

[84] "ARM Website," 2014. [Online]. Available: http://www.arm.com/products/processors/technologies/biglittleprocessing.php

[85] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2012, p. 317–328, 00018. [Online]. Available: http://dl.acm.org/citation.cfm?id=2457508

[86] "STM32 32-bit ARM Cortex MCUs Webpage." [Online]. Available: http://www.st.com/web/en/catalog/mmc/FM141/SC1169?sc=stm32

[87] "TI Cortex-A8 processor." [Online]. Available: http://www.ti.com/lsds/ti/arm/sitara_arm_cortex_a_processor/sitara_arm_cortex_a8/overview.page?paramCriteria=no

[88] S. Sankar, U. S. Kumar, M. Goel, M. S. Baghini, and V. R. Rao, "Considerations for static energy reduction in digital cmos ics using nems power gating," *IEEE Transactions on Electron Devices*, vol. 64, no. 3, pp. 1399–1403, March 2017.

[89] K. Agarwal, H. Deogun, D. Sylvester, and K. Nowka, "Power gating with multiple sleep modes," in *Quality Electronic Design, 2006. ISQED '06. 7th International Symposium on*, March 2006, pp. 5 pp.–637.

[90] H. Jiang, M. Marek-Sadowska, and S. Nassif, "Benefits and costs of power-gating technique," in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, Oct 2005, pp. 559–566.

[91] Q. Wu, M. Pedram, and X. Wu, "Clock-gating and its application to low power design of sequential circuits," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 47, no. 3, pp. 415–420, Mar 2000.

[92] R. S. Shelar and M. Patyra, "Impact of local interconnects on timing and power in a high performance microprocessor," in *Proceedings of the 19th international symposium on Physical design.* ACM, 2010, pp. 145–152.

[93] S. K. Khatamifard, L. Wang, W. Yu, S. Köse, and U. R. Karpuzcu, "Thermogater: Thermally-aware on-chip voltage regulation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 120–132.

[94] "ARM/Linaro Energy Aware Scheduling (EAS) for Linux Systems," Online, Oct. 2018. [Online]. Available: https://developer.arm.com/open-source/energy-aware-scheduling

[95] C.-H. Lee and K. G. Shin, "On-line dynamic voltage scaling for hard real-time systems using the EDF algorithm," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International.* IEEE, 2004, p. 319–335, 00066. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1381318

[96] D. Shin and J. Kim, "Intra-task voltage scheduling on DVS-enabled hard real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1530–1549, Oct. 2005, 00032. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1512371

[97] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the Eighteenth ACM Symposium on Operating*

Systems Principles, ser. SOSP '01. New York, NY, USA: ACM, 2001, p. 89–102. [Online]. Available: http://doi.acm.org/10.1145/502034.502044

[98] S. Saewong and R. Rajkumar, "Practical Voltage-Scaling for Fixed-Priority RT-Systems," in *Real-Time and Embedded Technology and Applications Symposium, Proceedings. The 9th IEEE.* IEEE, May 2003, pp. 106–114.

[99] R. Jejurikar and R. Gupta, "Optimized slowdown in real-time task systems," in *16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. Proceedings*, 2004, pp. 155–164, 00000.

[100] ——, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the 42Nd Annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 111–116. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065612

[101] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 31:1–31:23, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1550987.1550994

[102] M. Bambagini, F. Prosperi, M. Marinoni, and G. Buttazzo, "Energy management for tiny real-time kernels," in *2011 International Conference on Energy Aware Computing*, Nov 2011, pp. 1–6.

[103] "Erika Enterprise RTOS v3 Website," Oct. 2018. [Online]. Available: http://www.erika-enterprise.com/

[104] M. Bambagini, M. Bertogna, and G. Buttazzo, "On the effectiveness of energy-aware real-time scheduling algorithms on single-core platforms," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sept 2014, pp. 1–8.

[105] Y.-H. Lee, K. Reddy, and C. Krishna, "Scheduling techniques for reducing leakage power in hard real-time systems," in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings*, Jul. 2003, pp. 105–112, 00102.

[106] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, p. 275–280, 00346. [Online]. Available: http://doi.acm.org/10.1145/996566.996650

[107] J.-J. Chen and T.-W. Kuo, “Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor,” in *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, ser. LCTES ’06. New York, NY, USA: ACM, 2006, p. 153–162, 00054. [Online]. Available: http://doi.acm.org/10.1145/1134650.1134673

[108] L. Niu and G. Quan, “Reducing both dynamic and leakage energy consumption for hard real-time systems,” in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES ’04. New York, NY, USA: ACM, 2004, p. 140–148, 00067. [Online]. Available: http://doi.acm.org/10.1145/1023833.1023854

[109] Y. Pan and M. Lin, “Dynamic leakage aware power management with procrastination method,” in *Canadian Conference on Electrical and Computer Engineering, 2009. CCECE ’09*, 2009, pp. 247–251, 00003.

[110] J.-J. Chen and T.-W. Kuo, “Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems.” in *IEEE/ACM International Conference on Computer-Aided Design, 2007. ICCAD 2007*, Nov. 2007, pp. 289–294, 00067.

[111] M. A. Awan, P. M. Yomsi, and S. M. Petters, “Optimal procrastination interval for constrained deadline sporadic tasks upon uniprocessors,” in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS ’13. New York, NY, USA: ACM, 2013, p. 129–138, 00000. [Online]. Available: http://doi.acm.org/10.1145/2516821.2516837

[112] R. Jejurikar and R. Gupta, “Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems,” in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004. ISLPED ’04*, Aug. 2004, pp. 78–81, 00149.

[113] H. Cheng and S. Goddard, “Integrated device scheduling and processor voltage scaling for system-wide energy conservation,” in *Proceedings of the 2005 workshop on power aware real-time computing*, 2005, 00020.

[114] 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*, 2006, pp. 313–322, 00101.

[115] L. Niu, "Rate-monotonic scheduling for reducing system-wide energy consumption for hard real-time systems," in *2010 IEEE International Conference on Computer Design (ICCD)*, 2010, pp. 159–165, 00002.

[116] ——, "System-level energy-efficient scheduling for hard real-time embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–4, 00007.

[117] T.-H. Chen and C.-F. Kuo, "Energy-aware scheduling for weakly-hard real-time system with i/o device," in *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2012, pp. 31–38, 00000.

[118] 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, 00018.

[119] J. Zhuo, "System-level energy-efficient dynamic task scheduling," in *in 42nd DAC*, 2005, p. 628–631, 00072.

[120] H. Cheng and S. Goddard, "EEDS/spl I.bar/NR: an online energy-efficient I/O device scheduling algorithm for hard real-time systems with non-preemptible resources," in *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, July 2006, pp. 10 pp.–260.

[121] ——, "Online energy-aware I/O device scheduling for hard real-time systems," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, 2006, pp. 6 pp.–, 00046.

[122] C.-Y. Yang, J.-J. Chen, C.-M. Hung, and T.-W. Kuo, "System-level energy-efficiency for real-time tasks," in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on.* IEEE, 2007, p. 266–273, 00010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4208853

[123] D. He and W. Mueller, "Online energy-efficient hard real-time scheduling for component oriented systems," in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2012, pp. 56–63, 00001.

[124] C.-Y. Yang, J.-J. Chen, and T.-W. Kuo, “An approximation algorithm for energy-efficient scheduling on a chip multiprocessor,” in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 468–473 Vol. 1.

[125] F. Kong, W. Yi, and Q. Deng, “Energy-efficient scheduling of real-time tasks on cluster-based multicores,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

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

[127] 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*, Aug 2009, pp. 383–392.

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

[129] E. Seo, J. Jeong, S. Park, and J. Lee, “Energy efficient scheduling of real-time tasks on multicore processors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 11, pp. 1540–1552, 2008, 00091.

[130] D.-S. Zhang, F.-Y. Chen, H.-H. Li, S.-Y. Jin, and D.-K. Guo, “An energy-efficient scheduling algorithm for sporadic real-time tasks in multiprocessor systems,” in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, Sept 2011, pp. 187–194.

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

[132] M. Thammawichai and E. C. Kerrigan, “Energy-efficient scheduling for homogeneous multiprocessor systems,” *CoRR*, vol. abs/1510.05567, 2015. [Online]. Available: http://arxiv.org/abs/1510.05567

[133] W. Sun and T. Sugawara, "Heuristics and evaluations of energy-aware task mapping on heterogeneous multiprocessors," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 599–607.

[134] D. Li and J. Wu, "Minimizing energy consumption for frame-based tasks on heterogeneous multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 810–823, March 2015.

[135] C. Y. Yang, J. J. Chen, T. W. Kuo, and L. Thiele, "An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 694–699.

[136] J. J. Chen and L. Thiele, "Task partitioning and platform synthesis for energy efficiency," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2009, pp. 393–402.

[137] M. U. K. Khan, M. Shafique, A. Gupta, T. Schumann, and J. Henkel, "Power-efficient load-balancing on heterogeneous computing platforms," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1469–1472.

[138] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 15:1–15:26, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2566618

[139] B. N. Alahmad and S. Gopalakrishnan, "Energy efficient task partitioning and real-time scheduling on heterogeneous multiprocessor platforms with qos requirements," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, pp. 314–328, 2011.

[140] C.-F. Kuo and Y.-F. Lu, "Task assignment with energy efficiency considerations for non-dvs heterogeneous multiprocessor systems," *SIGAPP Appl. Comput. Rev.*, vol. 14, no. 4, pp. 8–18, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2724928.2724929

[141] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari, "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms," *Journal of Systems Architecture*, vol. 74, pp. 46 – 60, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138376211730019X

[142] M. A. Awan and S. M. Petters, "Energy-aware partitioning of tasks onto a heterogeneous multi-core platform," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 205–214.

[143] M. A. Awan, P. M. Yomsi, G. Nelissen, and S. M. Petters, "Energy-aware task mapping onto heterogeneous platforms using dvfs and sleep states," *Real-Time Systems*, vol. 52, no. 4, pp. 450–485, Jul 2016. [Online]. Available: https://doi.org/10.1007/s11241-015-9236-x

[144] K. Prescilla and A. I. Selvakumar, "Modified binary particle swarm optimization algorithm application to real-time task assignment in heterogeneous multiprocessor," *Microprocessors and Microsystems*, vol. 37, no. 6, pp. 583 – 589, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933113000823

[145] E. Valentin, R. de Freitas, and R. Barreto, "Towards optimal solutions for the low power hard real-time task allocation on multiple heterogeneous processors," *Science of Computer Programming*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642317301648

[146] M. Thammawichai and E. C. Kerrigan, "Energy-efficient real-time scheduling for two-type heterogeneous multiprocessors," *Real-Time Systems*, vol. 54, no. 1, pp. 132–165, Jan 2018. [Online]. Available: https://doi.org/10.1007/s11241-017-9291-6

[147] A. Elewi, M. Shalan, M. Awadalla, and E. M. Saad, "Energy-efficient task allocation techniques for asymmetric multiprocessor embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2s, pp. 71:1–71:27, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2544375.2544391

[148] S. Pagani, A. Pathania, M. Shafique, J. J. Chen, and J. Henkel, "Energy efficiency for clustered heterogeneous multicores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1315–1330, May 2017.

[149] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: A survey," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 1, pp. 7:1–7:34, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2808231

[150] G. Menghani, "A fast genetic algorithm based static heuristic for scheduling independent tasks on heterogeneous systems," in *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*, Oct 2010, pp. 113–117.

[151] S. Ahmad, E. Munir, and W. Nisar, "PEGA: A performance effective genetic algorithm for task scheduling in heterogeneous systems," in *2012 IEEE 14th International Conference on High Performance Computing and Communication IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, Jun. 2012, pp. 1082–1087.

[152] W. Sun, "A novel genetic admission control for real-time multiprocessor systems," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec 2009, pp. 130–137.

[153] U. ManChon, C. Ho, S. Funk, and K. Rasheed, "Gart: A genetic algorithm based real-time system scheduler," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, June 2011, pp. 886–893.

[154] H. Chen, A. M. K. Cheng, and Y.-W. Kuo, "Assigning real-time tasks to heterogeneous processors by applying ant colony optimization," *Journal of Parallel and Distributed Computing*, vol. 71, no. 1, pp. 132 – 142, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151000198X

[155] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810 – 837, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731500917143

[156] N. Nikitin and J. Cortadella, *Static Task Mapping for Tiled Chip Multiprocessors with Multiple Voltage Islands.* Springer, 2012. [Online]. Available: http://www.bookmetrix.com/detail/chapter/6fa06d07-009a-4be6-bf6b-f4eeb8d75df3#citations

[157] W. Zhang, H. Xie, B. Cao, and A. M. Cheng, "Energy-aware real-time task scheduling for heterogeneous multiprocessors with particle swarm optimization algorithm," *Mathematical Problems in Engineering*, vol. 2014, 2014.

[158] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[159] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE.* IEEE, 2009, pp. 398–409.

[160] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Jul. 2010, pp. 6–11.

[161] W. H. Huang, M. Yang, and J. J. Chen, "Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share?" in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016, pp. 111–122.

[162] Y. Sun and G. Lipari, "A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor global fixed-priority scheduling," *Real-Time Systems*, vol. 52, no. 3, pp. 323–355, May 2016. [Online]. Available: https://doi.org/10.1007/s11241-015-9245-9

[163] D. Calvaresi, G. Albanese, F. Dubosson, M. Marinoni, and M. Schumacher, "A tasksets generator for supporting the analysis of multi-agent systems under general purpose and real-time conditions," 2018.

[164] J. Wu and Y. Huang, "Mcrtsim: A simulation tool for multi-core real-time systems," in *2017 International Conference on Applied System Innovation (ICASI)*, May 2017, pp. 461–464.

[165] M. Dellinger, A. Lindsay, and B. Ravindran, "An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms," *Journal of Experimental Algorithmics (JEA)*, vol. 17, pp. 4–3, 2012.

[166] M. Dellinger, P. Garyali, and B. Ravindran, "ChronOS Linux: a best-effort real-time multiprocessor Linux kernel," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE.* IEEE, 2011, pp. 474–479.

[167] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Real-Time Systems Symposium, 2008.* IEEE, 2008, pp. 157–169.

[168] S. Saha and B. Ravindran, "An Experimental Evaluation of Real-time DVFS Scheduling Algorithms," in *Proceedings of the 5th Annual International Systems and Storage Conference*, ser. SYSTOR '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:12, 00007.

[169] A. Vulgarakis, R. Shooja, A. Monot, J. Carlson, and M. Behnam, "Task Synthesis for Control Applications on Multicore Platforms," in *Proceedings of the 11th International Conference on Information Technology: New Generation.* IEEE, Apr. 2014, pp. 229–234.

[170] A. Lindsay and B. Ravindran, "On Cache-Aware Task Partitioning for Multicore Embedded Real-Time Systems," in *IEEE 11th Intl Conf on Embedded Software and Syst (ICESS),.* IEEE, 2014, pp. 677–684.

[171] Y. De Bock, S. Altmeyer, T. Huybrechts, J. Broeckhove, and P. Hellinckx, "Task-set generator for schedulability analysis using the taclebench benchmark suite," *SIGBED Rev.*, vol. 15, no. 1, pp. 22–28, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3199610.3199613

[172] "ARM info center," 2018. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0460d/CHDCBAAH.html

[173] "PMCTrack website," Sep. 2018. [Online]. Available: https://pmctrack.dacya.ucm.es/

[174] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, F. Castro, and M. Prieto-Matias, "PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler," *The Computer Journal*, vol. 60, no. 1, pp. 60–85, Jan. 2017.

[175] D. He and W. Mueller, "Online energy-efficient hard real-time scheduling for component oriented systems," in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2012, pp. 56–63.

[176] R. Jejurikar and R. Gupta, "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004. ISLPED '04*, Aug. 2004, pp. 78–81.

[177] 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.

[178] F. Kong, Y. Wang, Q. Deng, and W. Yi, "Minimizing multi-resource energy for real-time systems with discrete operation modes," in *Proceedings of the 2010*

22Nd Euromicro Conference on Real-Time Systems, ser. ECRTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 113–122. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2010.18

[179] R. Haupt and S. Haupt, *Practical Genetic Algorithms*, ser. Wiley InterScience electronic collection. Wiley, 2004. [Online]. Available: http://books.google.ca/books?id=k0jFfsmbtZIC

[180] J. Xu, "A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems," in *2010 IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications (MESA)*, Jul. 2010, pp. 288–294.

[181] G. Abu-Lebdeh and R. F. Benekohal, "Convergence variability and population sizing in micro-genetic algorithms," *Computer-Aided Civil and Infrastructure Engineering*, vol. 14, no. 5, pp. 321–334, 1999.

[182] O. Davidyuk, I. Selek, J. Ceberio, and J. Riekki, "Application of micro-genetic algorithm for task based computing," in *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, Oct 2007, pp. 140–145.

[183] A. Alajmi and J. Wright, "Selecting the most efficient genetic algorithm sets in solving unconstrained building optimization problem," *International Journal of Sustainable Built Environment*, vol. 3, no. 1, pp. 18 – 26, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212609014000399

[184] D. Guan, Z. Cai, and Z. Kong, "Reactive power and voltage control using micro-genetic algorithm," in *2009 International Conference on Mechatronics and Automation*, Aug 2009, pp. 5019–5024.

[185] X. Ren, Z. Chen, and Z. Ma, "Differential evolution using smaller population," in *2010 Second International Conference on Machine Learning and Computing*, Feb 2010, pp. 76–80.

[186] D. Mora-Melià, F. J. Martínez-Solano, P. L. Iglesias-Rey, and J. H. Gutiérrez-Bahamondes, "Population size influence on the efficiency of evolutionary algorithms to design water networks," *Procedia Engineering*, vol. 186, pp. 341 – 348, 2017, xVIII International Conference on Water Distribution Systems, WDSA2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877705817313565

[187] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms," *Real-Time Systems*, vol. 52, no. 6, pp. 808–832, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11241-016-9256-1

[188] J. Goossens and R. Devillers, "Feasibility intervals for the deadline driven scheduler with arbitrary deadlines," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, 1999, pp. 54–61.

[189] J. Y.-T. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Information Processing Letters*, vol. 11, no. 3, pp. 115 – 118, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0020019080901234

[190] L. Cucu and J. Goossens, "Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems," in *2007 Design, Automation Test in Europe Conference Exhibition*, April 2007, pp. 1–6.

[191] V. Nelis, P. M. Yomsi, and J. Goossens, "Feasibility intervals for homogeneous multicores, asynchronous periodic tasks, and fjp schedulers," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13. New York, NY, USA: ACM, 2013, pp. 277–286. [Online]. Available: http://doi.acm.org/10.1145/2516821.2516848

[192] L. Cucu-Grosjean and J. Goossens, "Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms," *Journal of Systems Architecture*, vol. 57, no. 5, pp. 561 – 569, 2011, special Issue on Multiprocessor Real-time Scheduling. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762111000300

[193] M. Nasri, G. Nasri, and M. Kargahi, "A framework to construct customized harmonic periods for real-time systems," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014, pp. 211–220.

[194] M. Nasri and G. Fohler, "An efficient method for assigning harmonic periods to hard real-time tasks with period ranges," in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 149–159.

[195] M. Mohaqeqi, M. Nasri, Y. Xu, A. Cervin, and K.-E. Arzen, "On the problem of finding optimal harmonic periods," in *Proceedings of the 24th International Conference*

on Real-Time Networks and Systems, ser. RTNS '16. New York, NY, USA: ACM, 2016, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/2997465.2997490

[196] ——, "Optimal harmonic period assignment: complexity results and approximation algorithms," *Real-Time Systems*, Apr 2018. [Online]. Available: https://doi.org/10.1007/s11241-018-9304-0

[197] H. Fu, J. Liu, Z. Han, and Z. Shao, "A heuristic task periods selection algorithm for real-time control systems on a multi-core processor," *IEEE Access*, vol. 5, pp. 24 819–24 829, 2017.

[198] J. Goossens and C. Macq, "Limitation of the hyper-period in real-time periodic task set generation," in *In Proceedings of the RTS Embedded System (RTS'01.* Citeseer, 2001.

[199] J. Xu, "A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems," in *2010 IEEE/ASME International Conference on Mechatronics and Embedded Systems and Applications (MESA)*, Jul. 2010, pp. 288–294, 00003.

[200] M. Qamhieh, L. George, and S. Midonnet, "A stretching algorithm for parallel real-time dag tasks on multiprocessor systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems.* ACM, 2014, p. 13.

[201] Y. Abdeddaïm, Y. Chandarli, R. I. Davis, and D. Masson, "Response time analysis for fixed priority real-time systems with energy-harvesting," *Real-Time Systems*, vol. 52, no. 2, pp. 125–160, 2016.

[202] R. Bouaziz, L. Lemarchand, F. Singhoff, B. Zalila, and M. Jmaiel, "Multi-objective design exploration approach for ravenscar real-time systems," *Real-Time Systems*, vol. 54, no. 2, pp. 424–483, 2018.

[203] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll, "Task period selection to minimize hyperperiod," in *ETFA2011*, Sept 2011, pp. 1–4.

[204] I. Ripoll and R. Ballester-Ripoll, "Period selection for minimal hyperperiod in periodic task systems," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1813–1822, Sept. 2013. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TC.2012.243

[205] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[206] “EEMBC Website,” Oct. 2018. [Online]. Available: https://www.eembc.org/products/

[207] J. Pallister, S. Hollis, and J. Bennett, “BEEBS: Open benchmarks for energy measurements on embedded platforms,” *arXiv preprint arXiv:1308.5174*, 2013.

[208] S. Kramer, D. Ziegenbein, and A. Hamann, “Real world automotive benchmarks for free,” in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[209] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, “Time-analysable non-partitioned shared caches for real-time multicore systems,” in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC ’14. New York, NY, USA: ACM, 2014, pp. 198:1–198:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593235

[210] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega, and F. J. Cazorla, “Timing analysis of an avionics case study on complex hardware/software platforms,” in *Proceedings of the 2015 Design, Automation &#38; Test in Europe Conference &#38; Exhibition*, ser. DATE ’15. San Jose, CA, USA: EDA Consortium, 2015, pp. 397–402. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755843

[211] F. Cros, L. Kosmidis, F. Wartel, D. Morales, J. Abella, I. Broster, and F. J. Cazorla, “Dynamic software randomisation: Lessons learned from an aerospace case study,” in *Proceedings of the Conference on Design, Automation & Test in Europe.* European Design and Automation Association, 2017, pp. 103–108.

[212] I. Fedotova, B. Krause, and E. Siemens, “Upper bounds prediction of the execution time of programs running on arm cortex-a systems,” in *Doctoral Conference on Computing, Electrical and Industrial Systems.* Springer, 2017, pp. 220–229.

[213] K. P. Silva, L. F. Arcaro, D. B. de Oliveira, and R. S. de Oliveira, “An empirical study on the adequacy of mbpta for tasks executed on a complex computer architecture with linux,” 2018.

[214] S-18 Aircraft And System Development And Safety Assessment Committee, “Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. standard arp4761,” 1996.

[215] N. Brookwood, “Everything You Always Wanted to Know About HSA,” AMD, Tech. Rep., 10 2013.

[216] NVIDIA, “Variable SMP (4-PLUS-1™) – A Multi-Core CPU Architecture for Low Power and High Performance,” NVIDIA, Tech. Rep., 2011.

[217] TI, “OMAP™ 5 mobile applications platform,” Texas Instruments, Tech. Rep., 2011.

[218] ARM, “big.LITTLE Technology: The Future of Mobile - Making very high performance available in a mobile envelope without sacrificing energy efficiency,” ARM, Tech. Rep., 2013.

[219] “Arm DynamIQ technology - Redefining multi-core processing for the next era of computing,” https://developer.arm.com/technologies/dynamiq, 2018, online; accessed 20 June 2018.

[220] M. J. Walker, S. Diestelhorst, A. Hansson, A. K. Das, S. Yang, B. M. Al-Hashimi, and G. V. Merrett, “Accurate and stable run-time power modeling for mobile and embedded cpus,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 106–119, Jan 2017.

[221] ——, “Accurate and stable run-time power modeling for mobile and embedded cpus,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 106–119, Jan 2017.

[222] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba, “A design-space exploration for allocating security tasks in multicore real-time systems,” *CoRR*, vol. abs/1711.04808, 2017. [Online]. Available: http://arxiv.org/abs/1711.04808

[223] J. Abella, C. Hernandez, E. Quinones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, “WCET analysis methods: Pitfalls

and challenges on their trustworthiness," in *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on.* IEEE, 2015, pp. 1–10.

[224] "Introduction to bayesian statistics," Mar. 2001. [Online]. Available: http://ccrma.stanford.edu/~jos/bayes/bayes.html

[225] "MiBench: a free, commercially representative embedded benchmark suite," 2001. [Online]. Available: http://vhosts.eecs.umich.edu/mibench/

[226] "aiT WCET Analyzers," 2002. [Online]. Available: https://www.absint.com/ait/

[227] "OTAWA (open tool for adaptive wcet analyses)," University of Toulouse, 2006. [Online]. Available: http://www.otawa.fr/

[228] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms," *Real-Time Systems*, vol. 52, no. 6, pp. 808–832, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11241-016-9256-1

[229] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll, "Task period selection to minimize hyperperiod," in *2011 IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2011, pp. 1–4, 00001.

[230] M. Fan and G. Quan, "Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform," in *Proceedings of the Conference on Design, Automation and Test in Europe.* EDA Consortium, 2012, pp. 503–508. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492834

[231] "Heptane WCET Analyzers," 2016, national Institute for Computer Science and Applied Mathematics, France. [Online]. Available: https://team.inria.fr/pacap/software/heptane/

[232] A. Suyyagh and Z. Zilic, "Real-time benchmark set synthesis based on pWCET estimation and bounded hyper-periods," in *IEEE International Conference on Circuits, System and Simulation (ICCSS 2017).* London, UK: IEEE, Jul. 2017, pp. 129–133. [Online]. Available: http://ieeexplore.ieee.org/document/8023196/

[233] A. Suyyagh, J. G. Tong, and Z. Zilic, "Analysis of meta-heuristics performance in energy aware scheduling of real-time embedded systems," in *2015 IEEE Conference*

on Applied Electrical Engineering and Computing Technologies (AEECT), Nov 2015, pp. 1–6.

[234] 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)*, Oct 2017, pp. 1–8.

[235] S. Ramanathan, A. Easwaran, and H. Cho, "Multi-rate fluid scheduling of mixed-criticality systems on multiprocessors," *Real-Time Systems*, vol. 54, no. 2, pp. 247–277, Apr. 2018. [Online]. Available: https://doi.org/10.1007/s11241-017-9296-1

[236] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011, aAI3502550.

[237] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[238] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," *SIGPLAN Not.*, vol. 49, no. 4, pp. 161–176, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2644865.2541974

[239] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2833

[240] A. Suyyagh and Z. Zilic, "Real-time benchmark set synthesis based on pwcet estimation and bounded hyper-periods," *2017 International Conference on Circuits, System and Simulation (ICCSS)*, pp. 129–133, 2017.

[241] 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.

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

[243] Y. Pan and M. Lin, "Dynamic leakage aware power management with procrastination method," in *Canadian Conference on Electrical and Computer Engineering, 2009. CCECE '09*, 2009, pp. 247–251.

[244] J. Zhao and H. Qiu, "Genetic algorithm and ant colony algorithm based energy-efficient task scheduling," in *2013 International Conference on Information Science and Technology (ICIST)*, Mar. 2013, pp. 946–950.

[245] S. Baruah, D. Chen, and A. Mok, "Jitter concerns in periodic task systems," in *, The 18th IEEE Real-Time Systems Symposium, 1997. Proceedings*, Dec. 1997, pp. 68–77.

[246] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, p. 275–280. [Online]. Available: http://doi.acm.org/10.1145/996566.996650

[247] H. Cheng and S. Goddard, "Online energy-aware i/o device scheduling for hard real-time systems," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, 2006, pp. 6 pp.–.

[248] J.-J. Chen and T.-W. Kuo, "Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems." in *IEEE/ACM International Conference on Computer-Aided Design, 2007. ICCAD 2007*, Nov. 2007, pp. 289–294.

[249] V. Swaminathan and K. Chakrabarty, "Pruning-based energy-optimal device scheduling for hard real-time systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign, 2002. CODES 2002*, 2002, pp. 175–180.

[250] L. Niu, "System-level energy-efficient scheduling for hard real-time embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–4.

[251] W. Wang, P. Mishra, and S. Ranka, "System-wide energy optimization with DVS and DCR," in *Dynamic Reconfiguration in Real-Time Systems*, ser. Embedded Systems. Springer New York, Jan. 2013, no. 4, pp. 129–163. [Online]. Available: http://link.springer.com/chapter/10.1007/978-1-4614-0278-7_6

[252] D. Simon, *Evolutionary Optimization Algorithms.* Wiley, 2013. [Online]. Available: http://books.google.ca/books?id=gwUwIEPqk30C

[253] R. Jejurikar and R. Gupta, "Optimized slowdown in real-time task systems," *IEEE Trans. Comput.*, vol. 55, no. 12, pp. 1588–1598, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/TC.2006.204

[254] B. A. Mahafzah and B. A. Jaradat, "The hybrid dynamic parallel scheduling algorithm for load balancing on chained-cubic tree interconnection networks," *The Journal of Supercomputing*, vol. 52, no. 3, pp. 224–252, Jun 2010. [Online]. Available: https://doi.org/10.1007/s11227-009-0288-3

[255] ——, "The load balancing problem in otis-hypercube interconnection networks," *The Journal of Supercomputing*, vol. 46, no. 3, pp. 276–297, Dec 2008. [Online]. Available: https://doi.org/10.1007/s11227-008-0191-3

[256] H. Jeon, Y.-B. Kim, and M. Choi, "Standby Leakage Power Reduction Technique for Nanoscale CMOS VLSI Systems," *Instrumentation and Measurement, IEEE Transactions on*, vol. 59, no. 5, pp. 1127–1133, May 2010.

[257] H. Iwai, "CMOS Technology after Reaching the Scale Limit," in *Junction Technology, 2008. IWJT '08. Extended Abstracts - 2008 8th International workshop on*, May 2008, pp. 1–2.

[258] J. Seng and D. Tullsen, "Exploring the potential of architecture-level power optimizations," in *Power-Aware Computer Systems*, ser. Lecture Notes in Computer Science, B. Falsafi and T. VijayKumar, Eds. Springer Berlin Heidelberg, 2005, vol. 3164, pp. 132–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28641-7_10

[259] B. Steigerwald, C. D. Lucero, C. Akella, and A. R. Agarwal, *Energy Aware Computing, Powerful Approaches for Green System Design.* INTEL, 2011.

[260] D. Li, J. Wu, K. Li, and K. Hwang, “Energy-aware scheduling on multiprocessor platforms with devices,” in *Cloud and Green Computing (CGC), 2013 Third International Conference on*, Sept 2013, pp. 26–33.

[261] A. Suyyagh, J. G. Tong, and Z. Zilic, “Performance evaluation of meta-heuristics in energy aware real-time scheduling problems,” *Journal of Computers and Information Technology*, vol. 2, no. 1, pp. 68–85, 2016.

[262] P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat, “Benchmark generation for timing analysis,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017, pp. 319–330.

[263] “Tms570ls1227 16- and 32-bit risc flash microcontroller,” Feb. 2015. [Online]. Available: http://www.ti.com/lit/ds/symlink/tms570ls1227.pdf