Patrick Eitschberger

Energy-efficient and Fault-tolerant Scheduling for Manycores and Grids

Mathematik und Informatik

Dissertation

FernUniversität in Hagen

FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Parallelität und VLSI

Energy-efficient and Fault-tolerant Scheduling for Manycores and Grids

Dissertation
zur Erlangung des akademischen Grades
DOKTOR DER NATURWISSENSCHAFTEN
(Dr. rer. nat.)

von
Dipl. Inform. Patrick Eitschberger, M. Comp. Sc.
Dorsten

Hagen, 2017

Promotionskommission:

Erster Gutachter:	Herr Prof. Dr. Jörg Keller (FernUniversität in Hagen, Germany)
Zweiter Gutachter:	Herr Prof. Dr. Christoph Kessler (Linköping University, Sweden)
Vorsitzender der Promotionskommission:	Herr Prof. Dr. Friedrich Steimann
Promovierte Mitarbeiterin:	Frau Dr. Daniela Keller

Tag der Disputation:
05. Oktober 2017

For my wife Katrin and
my children Milian and Jannis

Abstract

Parallel platforms like manycores and grids consist of a large number of processing units (PUs) to achieve high computing power. To enable a fast execution of complex applications, they must be decomposed into several tasks and scheduled efficiently onto the parallel platform.

The reliability of these platforms is crucial in order to avoid high costs in sense of money, time or life-critical situations. A failure of a PU can be tolerated by task duplication, where each task is duplicated onto another PU. In case of a failure, the schedule execution can be continued by running the duplicates instead of the faulty original tasks. But integrating duplicates into a schedule often results in performance overhead already in the fault-free case.

Additionally, a low energy consumption for a schedule is desired to reduce costs and to protect the environment. Dynamic voltage and frequency scaling (DVFS) is one approach to reduce the energy consumption. However, scaling the frequencies and voltages of PUs to an efficient level often leads to performance overhead, because tasks are usually slowed down. In addition, including duplicates into a schedule increases the energy consumption because they are typically executed beside the original tasks. This leads to a three-dimensional optimization problem of performance, fault tolerance and energy consumption.

In this thesis the interplay between these three objectives is explored. Several new fault-tolerant and energy-efficient heuristics and strategies are presented that offer a user the opportunity to set various preferences. Additionally, a prototype runtime system is presented that tolerates a failure and also uses DVFS to improve the energy consumption. Finally, the different strategies are evaluated with various test sets in general but also with real-world applications on real parallel platforms.

Zusammenfassung

Parallele Systeme, wie beispielsweise Manycores und Grids, bestehen aus einer großen Anzahl an Verarbeitungseinheiten (PUs), um eine hohe Rechenleistung zu erzielen. Um eine schnelle Abarbeitung komplexer Applikationen zu ermöglichen, müssen diese in mehrere Tasks zerlegt und effizient auf die parallele Plattform eingeplant werden.

Die Zuverlässigkeit dieser Plattformen ist entscheidend, um hohe Kosten im Sinne von Geld, Zeit oder lebensbedrohlichen Situationen zu vermeiden. Ein Ausfall einer PU kann durch Task-Duplikation toleriert werden, bei der jede Task auf einer anderen PU dupliziert wird. Im Falle eines Ausfalls kann die Schedule-Verarbeitung fortgesetzt werden, indem die Duplikate anstelle der ausgefallenen Original-Tasks ausgeführt werden. Allerdings führt das Einfügen von Duplikaten in einen Schedule bereits im fehlerfreien Fall häufig zu einem Leistungsverlust.

Zusätzlich wird ein niedriger Energieverbrauch für einen Schedule angestrebt, um Kosten zu reduzieren und die Umwelt zu schonen. Dynamische Spannungs- und Frequenzskalierung (DVFS) ist eine Möglichkeit, um den Energieverbrauch zu reduzieren. Jedoch führt die Skalierung der Frequenzen und Spannungen von PUs häufig zu einem Leistungsverlust, da Tasks für gewöhnlich verlangsamt werden. Außerdem erhöht das Einfügen von Duplikaten in den Schedule den Energieverbrauch, da sie in der Regel neben den Original-Tasks ausgeführt werden. Dies führt zu einem drei-dimmensionalen Optimierungsproblem zwischen Leistung, Fehlertoleranz und Energieverbrauch.

In dieser Arbeit wird das Zusammenspiel zwischen diesen drei Zielen erforscht. Mehrere neue fehlertolerante und energieeffiziente Heuristiken und Strategien werden präsentiert, die einem Benutzer die Möglichkeit bieten unterschiedliche Präferenzen einzustellen. Darüber hinaus wird ein prototypisches Laufzeitsystem vorgestellt, das einen Fehler toleriert und DVFS benutzt, um den Energieverbrauch zu verbessern. Zum Abschluss werden die unterschiedlichen Strategien mit verschiedenen Testmengen im Allgemeinen aber auch mit realen Anwendungen auf realen parallelen Plattformen evaluiert.

Acknowledgement

I would like to thank in particular Prof. Dr. Jörg Keller for his great supervision and the countless fruitful discussions, helpful hints and his untiring patience (especially in the final phase of my Ph.D.) during my time as a research assistant and Ph.D.-student in his research group.

Many thanks to Prof. Dr. Christoph Kessler for interesting discussions, ideas and insights in related topics, which have positively influenced the development of this thesis. Thanks for agreeing to be the second reviewer of this thesis.

I am grateful for the detailed discussions with Dr. Simon Holmbacka related to several parts of this thesis, like power modeling of real systems, trade-off analysis and estimation of lower/upper bounds. This helped me to find new ideas and to improve the quality of this thesis. Thanks for also proof-reading most of the parts of this thesis.

I like to thank Prof. Dr. Wolfram Schiffmann and Dr. Jörg Lenhardt for the discussions and feedbacks about energy modeling. Special thanks to Dr. Jörg Lenhardt for his feedback to several questions related to LaTeX and for proof-reading this thesis.

I especially would like to thank my lovely family for the support and motivation during my Ph.D.-study. Special thanks to my wife Katrin, who gave me the necessary freedom to write down this thesis and who always listened to me, when I had problems. Warm thanks to my children Milian and Jannis, who brought a smile to my face whenever I was stressed or unmotivated.

Thanks to all not explicitly mentioned persons, who supported me during this time.

Patrick Eitschberger
Hagen
June 5, 2017

Publications and Previous Work

Parts of the work described in this thesis have been published in the following articles. After a short description of the content, the contribution of the authors is described for each publication. All parts in this thesis that are related to these articles are explicitly cited. The sections that cover the work in the articles are given for each article.

Patrick Cichowski, Jörg Keller, Christoph Kessler: *Modelling Power Consumption of the Intel SCC*. 6th Many-core Applications Research Community Symposium (MARC '12), Toulouse, France, Jul. 2012, pp. 46-51 [45] (covered in Sect. 6.3.2).

> In this article, a power model for the Intel SCC (Single-chip Cloud Computer) is presented. Based on the results of several micro-benchmarks and workload distributions over all cores the tuning parameters for the proposed power model are determined by a least squares analysis.
>
> This publication was written by Patrick Cichowski, Jörg Keller and Christoph Kessler. Jörg Keller suggested the power model and the micro-benchmarks for the Intel SCC. Christoph Kessler contributed several details about the Intel SCC and related work. Patrick Eitschberger implemented and evaluated the micro-benchmarks and gave further information of the SCC.

Patrick Eitschberger, Jörg Keller: *Efficient and Fault-tolerant Static Scheduling for Grids*. 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC '13), Boston, Massachusetts, May 2013, pp. 1439-1448 [53] (covered in Sects. 2.4.3, 4.2.1 and 4.2.3).

> In this article, a fault-tolerant duplication-based scheduler is presented that guarantees no overhead in a fault-free case. Based on the approach of Fechner et al. [59] where no communication time is considered, the influence of the communication time between tasks onto the placement of duplicates is discussed and different strategies are presented. Additionally, in this paper another kind of fault is introduced where tasks can be slowed down because of a high usage rate of a processor. In this case, the already placed duplicates can be used to speedup the execution.
>
> This publication was written by Patrick Eitschberger as most of the results were based on his German diploma thesis. Jörg Keller proof-read the paper and checked the writing style. He furthermore contributed the idea of using the duplicates for slowed down tasks. Patrick Eitschberger implemented and evaluated this extension.

Patrick Eitschberger, Jörg Keller: *Energy-efficient and Fault-tolerant Task Graph Scheduling for Manycores and Grids*. 1st Workshop on Runtime and Operating Systems for the Many-core Era (ROME '13), Aachen, Germany, Aug 2013, pp. 769-778 [54] (covered in Sects. 4.3.1 and 4.3.4).

> In this article, as an extension, frequency scaling is included into the previous approach [53] to either improve energy consumption in the fault-free and fault case or the performance overhead in case of a failure. The frequency for tasks is scaled down without prolonging the makespan of the original schedule. In case of a failure tasks can also be speeded up to reduce the performance overhead.
>
> This publication was written by Patrick Eitschberger. He also contributed the main idea of integrating energy efficiency aspects, i.e. frequency scaling into the fault-tolerant scheduler and the corresponding heuristic. Jörg Keller contributed details for a generalized power model and suggested to also use frequency scaling for speeding up tasks in case of a failure. The implementation and evaluation was done by Patrick Eitschberger.

Christoph Kessler, Nicolas Melot, Patrick Eitschberger, Jörg Keller: *Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Mallable Streaming Tasks*. 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS '13), Sept 2013, pp. 215-222 [104] (used as related work in Sect. 2.5.3).

> In this article, an energy-efficient optimization approach, called crown scheduling, for malleable streaming tasks is presented. Crown scheduling is based on integer linear programming (ILP) and combines next to a separate consideration the resource allocation, mapping and discrete voltage/frequency scaling under a given throughput constraint.
>
> This publication was written by Christoph Kessler and Jörg Keller. Christoph Kessler contributed an ILP model for the separate and combined crown scheduling. Patrick Eitschberger developed a task set generator for the experimental section and participated in combining the software for the separate phases. Nicolas Melot implemented and combined the separate phases of crown scheduling. He also adapted details of the ILPs and evaluated crown scheduling.

Nicolas Melot, Christoph Kessler, Jörg Keller, Patrick Eitschberger: *Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems*. ACM Transactions on Architecture and Code Optimization (TACO '15), vol. 11, no 4, pp. 62:1-62:24, Jan. 2015 [126] (used as related work in Sect. 2.5.3).

In this article, the previous approach [104] is extended by several phase-separated and integrated heuristics for Crown Scheduling. The longest Task, lowest Group (LTLG) heuristic is introduced to balance the load for mapping parallel tasks. A Height heuristic is presented for frequency scaling. The allocation is optimized by heuristics based on binary search and simulated annealing.

This publication was written by Nicolas Melot. He designed the optimal allocation and the phase-separated and integrated heuristics for Crown Scheduling. Christoph Kessler contributed the concrete task set in the extended experimental section. Christoph Kessler, Patrick Eitschberger and Jörg Keller proof-read the paper and checked the writing style.

Patrick Eitschberger, Jörg Keller: *Energy-efficient Task Scheduling in Manycore Processors with Frequency Scaling Overhead*. 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '15), Turku, Finland, March 2015, pp. 541-548 [55] (covered in Sect. 2.5).

This article focuses on evaluating the overhead in time and energy that is spent for frequency scaling. Especially for small time scales the frequency scaling overhead can have a significant influence on the runtime and energy of a schedule. With the help of a bin packing heuristic the frequency scaling overhead is considered and optimized schedules are created.

This publication was written by Patrick Eitschberger and Jörg Keller. Jörg Keller contributed the idea to use bin packing and participated in interpreting the results. Patrick Eitschberger contributed to the cost function for the bin packing. He also implemented and evaluated the bin packing heuristic.

Patrick Eitschberger, Jörg Keller: *Fault-tolerant Parallel Execution of Workflows with Deadlines*. 25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '17), St. Petersburg, Russia, March 2017, pp. 541-548 [56] (covered in Sects. 4.3.6, 4.3.7, 4.3.8 and 4.5).

In this article, the previous work in [53] and [54] is extended by several approaches to improve the energy consumption in case of a failure. Next to various heuristics also ILPs for optimal solutions are presented.

This publication was written by Patrick Eitschberger and Jörg Keller. Jörg Keller contributed the idea and implementation of the CP-heuristic (Constant Power). Patrick Eitschberger contributed the idea and implementation of the LFR-heuristic (Lazy Frequency Re-scaling). He also integrated both approaches into the already existing scheduler from the previous work and did the evaluation. The main idea for the ILPs to optimize the energy consumption

for a schedule was given by Christoph Kessler for the publication about Crown Scheduling above [104]. Jörg Keller and Patrick Eitschberger adapted this idea for energy-efficient and fault-tolerant schedules.

The author of this Ph.D.-thesis already wrote his German diploma thesis in the context of fault-tolerant scheduling [44] where the primary version of the scheduler was implemented that is used, partly re-implemented and significantly extended in this Ph.D.-thesis. All parts of this Ph.D.-thesis that are related to this previous work are strictly separated from the contributions and marked as previous work. The organization in Chap. 2 (*Background*) is in Sects. 2.3 (*Scheduling*) and 2.4 (*Fault Tolerance*) partially based on the presentation in the German diploma thesis. In Tab. 0.1 the mentioned previous work is summarized in the context of this Ph.D.-thesis to provide the reader a clear distinction.

Table 0.1: Summary of Previous Work Done for the German Diploma Thesis in Comparison to the Work Done for the Ph.D.-thesis.

Topic	**German Diploma thesis**	**Ph.D-thesis**
Scheduling	✓	✓(Chap. 2; additional rubrics in the classification and related work)
Fault Tolerance	✓	✓(Chap. 2; additional related work and fault tolerance in MPI (Message Passing Interface))
Fault-tolerant Scheduling Heuristics	✓	✓(Chap. 4; additional heuristic and fault type)
Strategies	✓(fault-tolerant)	✓(Chap. 4; additional fault-tolerant and energy-efficient strategies)
Detailed Description of Parallel Platforms	-	✓(Chap. 2)
Designing Parallel Applications	-	✓(Chap. 2)
Explanations and Definitions in the Context of Energy Efficiency	-	✓(Chap. 2)
Trade-off Description	-	✓(Chap. 3)
Estimation of Upper/Lower Bounds	-	✓(Chap. 3)
Energy-efficient Heuristics	-	✓(Chap. 4)
Energy Optimization with ILPs	-	✓(Chap. 4)
Runtime System	-	✓(Chap. 5)
Power Modeling	-	✓(Chap. 5)
Real-world Scenarios	-	✓(Chap. 6)

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

3SAT	3-SATisfiability
ABAC	Algorithm Based on Application Checkpointing
ABSC	Algorithm Based on System Checkpointing
ACO	Ant Colony Optimization
ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
b-level	bottom level
BER	Buffer for Energy Reduction
BLCR	Berkeley Lab Checkpoint/Restart
BNP	Bounded Number of Processors
CASPER	Combined Assignment, Scheduling, and PowER management
CBF	Change Base Frequency
COW	Cluster of Workstations
CP	Constant Power
CP	Critical Path
CP/MISF	Critical Path/ Most Immediate Successors First
CPU	Central Processing Unit
CPUFreq	Central Processing Unit Frequency
CT	Completion Time
D	Duplicate
DAG	Direct Acyclic Graph
DBUS	Duplication-based Bottom-Up Scheduling
DD	Dummy Duplicate
DFTS	Distributed Fault-Tolerant Scheduling
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DRFT	Dynamic Replication of Fault-Tolerant scheduling
DSC	Dominant Sequence Clustering
DUPS	Duplication-based scheduling Using Partial Schedules
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
DYTAS	Dynamic TAsk Scheduling
E	Energy consumption
EOTD	Energy Optimization scheduling for Task Dependent graph
EP	Energy for Performance
FT	Fault Tolerance
FT-MPI	Fault-Tolerant Message Passing Interface

FTWS	Fault-Tolerant Workflow Scheduling
GEV	Generalized Extreme Value
GHz	GigaHertz
HA	High Availability
HCP	Heterogeneous Critical Path
HLF	Highest Level First
HLFET	Highest Level First with Estimated Times
HPC	High Performance Computing
IDA*	Iterative Deepening A*
ILP	Integer Linear Programming
J	Joule
LAN	Local Area Network
LDCP	Longest Dynamic Critical Path
LFR	Lazy Frequency Re-scaling
LLREF	Largest Local Remaining Execution First
LP	Longest Path
LPT	Longest Processing Time
LTB	Large Test Benchmark
MC	Memory Controller
MHz	MegaHertz
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMI	Maximum Makespan Increase
MP-SoC	MultiProcessor System-on-Chip
MPI	Message Passing Interface
MPP	Massively Parallel Processor
ms	milliseconds
MSR	Model-Specific Register
MultiOp	Multiple Operation
NDFS	Nearest Deadline First Served
NUMA	NonUniform Memory Access
OpenMP	Open Multi-Processing
P	Power consumption
PAPI	Performance Application Programming Interface
PCAM	Partition Communicate Agglomerate Map
PDS	Pruned Depth-first Search
PE	PErformance
PKG	Package
POSIX	Portable Operating System Interface for uniX
PP0	PowerPlane0
PP1	PowerPlane1
PU	Processing Unit
PY	Papadimitriou Yannakakis
RAID	Redundant Array of Independent Disks

RAM	Random Access Memory
RAPL	Running Average Power Limit
RUPS	Runtime system for User Preferences-defined Schedules
s	seconds
sb-level	static bottom level
SCC	Single-chip Cloud Computer
SDE	Scheduling → Duplicates → Energy
SED	Scheduling → Energy → Duplicates
SIFT	Software Implemented Fault Tolerance
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SLS	Simple List Scheduler
SMT	Simultaneous MultiThreading
SPMD	Single Program Multiple Data
SSE	Streaming Single instruction multiple data Extension
SSF	Standard Schedule Format
STG	Standard Task Graph
t-level	top level
TB	Test Benchmark
TDB	Task Duplication-Based
UHPO	Use Half PUs for Originals
ULFM-MPI	User Level Failure Mitigation Message Passing Interface
UMA	Uniform Memory Access
UNC	Unbounded Number of Clusters
VLIW	Very Long Instruction Word
W	Watt
WQR	WorkQueue with Replication
WQR-FT	WorkQueue with Replication Fault-tolerant
Ws	Watt-second

1 Introduction

Today's computer systems typically consist of several processing units (PUs) in order to achieve high computing power, starting from common desktop computers, where usually four to eight cores are integrated on a single processor, up to manycores, clusters, grids and clouds that include dozens, hundreds or thousands of PUs. In addition, many problems like weather forecasting, simulating rush hour traffics or modeling vehicle constructions are complex and massively parallel. Corresponding algorithms can be parallelized onto several PUs to speed up the execution or to increase the degree of details. Therefore, corresponding applications are decomposed into several distinct parts and scheduled onto various PUs. Often, dependencies between the parts of an application exist, because results of parts are used as input for successor parts. In this case, these parts are called tasks. The schedule of such a complex parallel application can be created statically prior to or dynamically during the execution. Static scheduling is typically performed with the help of a task graph that represents the tasks and dependencies between them.

However, with a larger number of PUs and tasks the probability of a fault or failure during the execution of an application increases. As a failure of a PU often causes high economic costs or life-critical situations, the reliability of a parallel platform is crucial. A failure can result from both a hardware fault (e.g. a damaged processor core or a corrupted network connection) or from a software fault (e.g. the program stops or is interrupted for some reason). Typically, faults are tolerated by redundancy. One kind of fault tolerance is the task duplication where for each task a copy – a so-called duplicate – is created on another PU. In case of a failure, the duplicate is used to continue the schedule execution. The performance of the system in the fault case will then benefit from the duplicates, since the progress of the schedule can seamlessly be continued by the tasks' duplicates.

To maximize performance in static schedules, it is critical to minimize the length of a schedule, the so-called makespan. However, integrating fault tolerance techniques typically results in performance overhead. This leads to increasing makespans. Therefore, a trade-off exists for this two-dimensional scheduling problem.

The problem of minimizing the energy consumption is another issue emerging especially in recent years, as high energy consumption causes high costs and negatively influences the environment. In a computer system, the processor is one of the most energy consuming components. To reduce the energy consumption of a processor and thus for a schedule execution, computer systems typically support energy saving features like scaling voltage and frequency of a PU dynamically according to its usage rate. While the energy consumption is improved by scaling the clock frequency and voltage of a PU to an energy-efficient level, the performance of a schedule is usually decreased because typically the task execution is slowed down. Therefore, another trade-off exists for this two-dimensional scheduling problem.

Additionally, when integrating fault tolerance into the schedule, the tasks' duplicates require extra resources because the task is actually executing simultaneously on various PUs. In the fault-free case this is regarded as energy wasting. Therefore, in this thesis the interplay between the three criteria performance, fault tolerance and energy consumption is explored and discussed for manycores and grids as two common parallel platforms with a large number of PUs.

The following research questions can be formulated:

Is it possible to combine all three criteria?
How do these criteria influence each other?
What options do users have to reach their preferences?
How to realize a corresponding runtime system?
How to model the power consumption to predict the energy?

The main objectives of this thesis are therefore to integrate fault tolerance into schedules by task duplication and to minimize the energy consumption by using frequency and voltage scaling. Additionally, the preferences of a user are represented by different strategies and options. Next to these objectives, a performance loss of a schedule by integrating fault tolerance and energy efficiency aspects is avoided as far as possible. A power model for real systems with an acceptable accuracy is proposed, to predict the energy consumption. A runtime system is provided by using existing methods without changing the operating system of PUs.

Contributions

The contribution of this thesis can be subdivided into three parts:

Fault-tolerant and Energy-efficient Scheduler Although the optimization for all two-dimensional combinations of performance, fault tolerance and energy consumption is well researched in the literature, the three-dimensional optimization is rarely addressed. There exist a few exceptions that focus on real-time systems where tasks have to be executed in predefined time frames or within a certain deadline. Therefore, the performance in corresponding approaches is the major objective. In addition, typically transient faults are considered in these systems. In this work, permanent faults are considered and innovative scheduling heuristics and strategies are presented that combine all three criteria without a real-time constraint. Hence, in this work a broader range is considered, which is not yet addressed in previous work. In each strategy, different criteria are dominating to provide a scheduler with options for various preferences of a user. Next to strategies that focus on the fault-free case also strategies for the fault case are presented. In related works, the focus usually is only put on heuristics or strategies for one of both cases. In this work, all proposed scheduling strategies are constructed to be used with an already existing schedule (and task graph). Therefore, the classical scheduling is separated from the fault-tolerant and energy-efficient extensions. This has the advantage that the strategies can be used independently without a restriction to a specific scheduler.

Trade-off Study An overall and detailed trade-off study between the three criteria does not yet exist in the literature. Often only smaller parts are considered separately. Additionally, another indirect trade-off between the fault-free and fault-case is left out in previous work. With this thesis, all the above-mentioned trade-offs are analyzed and visualized in order to obtain detailed insights into the interplay between performance, fault tolerance and energy consumption in general, but also for common computer systems. This is helpful for developers to find appropriate algorithms and for system architects in order to plan and design new platforms.

Runtime System As an operating system lacks information about a task graph, it cannot scale the frequencies efficiently or handle with failures of PUs during the execution of a schedule. In this work, a prototype runtime system is therefore presented that combines the information about task graph and schedule with supporting fault

tolerance and frequency scaling. The concepts and methods used in the runtime system are explained in detail to show how such a runtime system can be developed without changing the operating system.

Structure of the thesis

Chap. 2 explains all relevant basics for this thesis. Starting from various types and classifications of parallel platforms the process of developing parallel applications to benefit from those platforms is described. Scheduling as part of the parallel application design is separately discussed in detail. Then, the chapter introduces fault tolerance and energy efficiency as two major objectives next to the classical performance objective in scheduling. Chap. 3 discusses the trade-off between the three objectives, where firstly the two-dimensional optimization for all combinations is explained. Secondly, the three-dimensional optimization is considered. Finally, the chapter investigates in an estimation of upper and lower bounds for all objectives. Chap. 4 presents several energy-efficient and fault-tolerant heuristics and options after a short review of the previous work. Then, nine strategies are formulated (including two strategies from the previous work) that represent different user preferences in both the fault-free and fault case. The chapter also describes energy optimal solutions by using integer linear programming. In Chap. 5 a fault-tolerant and energy-efficient prototype runtime system is presented with all its components like a system check tool, the runtime system itself and also a power model for actual multicore processors. Chap. 6 discusses the experiments and evaluation of the strategies with various test sets for a generalized power model followed by experiments on common Intel processors in the mobile, desktop and server field and for the Intel SCC as an example manycore processor. Next to the evaluation of the strategies, an analysis and visualization of the trade-off between all objectives follows. Chap. 7 concludes the thesis and finally, Chap. 8 introduces several future research directions.

2 Background

In recent decades, both computer systems and applications were in an ongoing change. The performance of a computer was improved over the years by increasing the clock speed of the processor, introducing pipelining and superscalar execution and using fast memories (caches) to bridge the gap between the slow *RAM (Random Access Memory)* access times and the high processor speed [73]. While older computer systems until the mid-80's consisted of a single *PU (Processing Unit)*, so called *uniprocessor* systems, newer computer systems include several PUs to increase the performance. In addition, computers were and still are increasingly interconnected not only by *LANs (Local Area Network)* but also through the Internet since its introduction in the late-80's. This trend is due to physical limitations on the one hand, a further increase of the clock speed of a processor is very cost intensive and ineffective because it leads to disproportional high power consumption and heat. On the other hand, the distributed nature of many problems that can be subdivided into several smaller parts and solved in parallel, motivated the decision of developing and using parallel computing platforms to improve the performance [73].

However, in terms of software, the resulting parallel applications have to be distributed suitably onto the multiple PUs. Therefore, an efficient mapping and scheduling is essential. Next to the performance, the reliability of a system is of central interest. In several parallel computing platforms like clusters, grids and clouds, fault tolerance is indispensable, because the probability of an error or a failure increases with the number of PUs. Also the network used to communicate between the PUs might be fault-prone.

In recent years, another challenge was arisen besides fault tolerance and performance, which is energy efficiency. Computer systems consume significant power and thus energy. To reduce the power consumption, modern computer systems support several processor features to save energy like *DVFS (Dynamic Voltage and Frequency Scaling)* or *DPM (Dynamic Power Management)*. Other components e.g. the hard disk, the screen, or a couple of ports support sleep states or can be switched off completely, when the components are not needed for the moment.

In Sect. 2.1 a short overview of different parallel platforms is given and some classifications are introduced. The developing process of parallel applications is described in Sect. 2.2. Scheduling is explained in Sect. 2.3. Different fault tolerance techniques are presented in Sect. 2.4. Finally, the energy efficiency is described in Sect. 2.5.

2.1 Parallel Platforms

2.1.1 Types

Today's computer systems can be subdivided into several types of parallel platforms. The most common types are explained in the following:

Uniprocessor/Singlecore Processor Computer systems with one PU are called *uniprocessor* or *singlecore processor* systems. This type is not related to parallel platforms but it is mentioned for the sake of completeness. In such systems, parallelism is achieved by multitasking or *SMT (Simultaneous MultiThreading)* where several programs are executed quasi simultaneously on one PU. When an instance of a program is running, it is called a *process*. A process is assigned to an address space and typically consists of several light weight processes called *threads* [172].

Multicore Processor A processor that consists of multiple cores in a single chip with a shared memory is a so-called *multicore processor*. Those computer systems are used for both executing several sequential programs on different cores or parallel programs on several cores simultaneously. This type of parallel platform is the most common in general-purpose computers. In todays' multicore processors the number of cores is typically between two and 16 cores.

Manycore Processor *Manycore processors* consist of dozens or hundreds of cores on a single device. There is no specific limit in the number of cores to differ between manycore and multicore systems. But in contrast to multicore processors, the hardware is usually optimized for the execution of parallel programs.

Multiprocessor When several processors are included in a computer system (also with a shared memory), it is called a *multiprocessor system*. The processor type in those systems can vary from singlecore to multicore processors. Thus, there does not

exist a restriction to the type of used processors. One special kind of multiprocessors is the so-called *MP-SoC (MultiProcessor System-on-Chip)* [96]. In such system, all or most components of a computer system are integrated within a single chip. In MP-SoCs, several other components next to the processors are included onto the die like a high-speed network or different memories. MP-SoCs are mainly used in mobile or embedded systems, due to the lack of space and to minimize the power consumption.

Array-/Vector-Processor An *array* or *vector processor* is a microprocessor that executes one instruction on an array or vector of data elements instead of on single data elements at a time. Vector processors are typically used to improve the performance of numerical simulations [81].

VLIW Processor *VLIW processors (Very Long Instruction Word)* achieve high levels of instruction level parallelism by executing long instruction words that consist of multiple operations, so-called MultiOps. A *MultiOp* is a set of multiple control, arithmetic and logic operations that are executed concurrently on a VLIW processor [134].

Multicomputer *Multicomputer* is a wide spread term. Under this type of parallel platform several systems are considered where the computers within a system are interconnected by a LAN or by the Internet. These systems consist of a distributed memory:

- Computer Cluster: In a *computer cluster*, several computers are tightly connected through a LAN to work together. Each computer has its own operating system, and the communication between those is done by message passing. The administration of all computers is directed to a single organization. The most common types of computer cluster are *HPC-Cluster (High Performance Computing)* and *HA-Cluster (High Availability)*. If the computers are interconnected by a high-speed proprietary interconnection network, the cluster is also called *MPP (Massively Parallel Processor)* or *supercomputer*. Another type of computer cluster is the so-called *COW (Cluster of Workstations)*. As the name already indicates, in this type of computer cluster, the computers are regular PCs or workstations that are connected to each other. In contrast to conventional computer clusters, the computers are usually cheaper, but more

place is necessary for the whole system and the system is more distributed over an area and not related to a single room or building [171].

- Grid System: A *grid system* consists of several computers or computer clusters that are interconnected through a LAN or through the Internet. A main difference between computer clusters and grids is that the administration of the computers in a grid system is directed to different organizations. Users can submit their programs to the grid system where a grid middleware coordinates and distributes the programs to the different grid nodes (computers). One of the main challenges in grids is to offer computing power as needed by the user like electricity from a socket.

- Cloud Computing: *Cloud computing* is a kind of internet-based computing where a user can rent computer systems, applications, storages etc. on demand as needed and accesses to it remotely. A cloud is, from the perspective of a parallel platfrom, a multicomputer system that consists of interconnected computer nodes like computer clusters. In contrast to grid computing, virtualization is typically used in cloud computing.

2.1.2 Classifications

The huge number of different parallel platforms that were built over the years led to many approaches of classifying these into categories. An early classification of parallel architectures was given by Flynn in the year 1972 [61]. He classified computer systems based on the number of instruction and data streams, where a *stream* is a sequence of instructions or data. Parallel architectures can be subdivided in four categories:

- SISD – Single Instruction (stream), Single Data (stream):
  Conventional sequential machines fall into this category. The *CPU (Central Processing Unit)* acts on a single instruction stream and one data item is processed per cycle.

- SIMD – Single Instruction (stream), Multiple Data (stream):
  Several processors execute the same instruction but with different data elements. Array-, vector- and VLIW-processors are examples assigned to this category.

- MISD – Multiple Instruction (stream), Single Data (stream):
  The type of parallel computers that would be related to this category is very uncommon. Therefore, in general this category applies to be empty.

- MIMD – Multiple Instruction (stream), Multiple Data (stream):
  Here, multiple processors or threads execute different instructions on different data elements. One important subclass in this category is SPMD – Single Program, Multiple Data. Each processor executes the same program but on different data elements. The execution of the program can be at any point for different processors. Multicomputer, multiprocessor, multi- and manycore processors are examples that fall into this category.

In Fig. 2.1 the four categories are depicted. The columns represent the number of data streams, the rows represent the number of instruction streams.

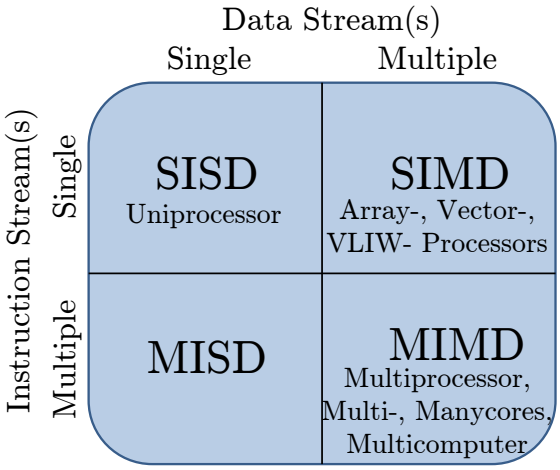

Figure 2.1: Flynn's Classification of Computer Systems [139].

Parallel platforms can also be classified into categories according to the organization of the memory, so-called *shared memory systems* and *distributed memory systems*:

- Shared Memory System:
  In a shared memory system, all processing units access one memory. Shared memory systems can be subdivided into *UMA (Uniform Memory Access)* and *NUMA (NonUniform Memory Access)* machines. In the former, all PUs can access every memory location in the same time. In contrast, in the latter, the memory modules are close to the PUs and therefore the access time differs between close and distant modules.

- Distributed Memory System:
  In these systems, every PU has its own memory. The communication between the PUs (and memories) is done via messages.

Following Tanenbaum [171] in general, a combination of both classifications with the different types of parallel platforms leads to a detailed taxonomy, as shown in Fig. 2.2. In this thesis, the major focus is directed to the parts of the figure that are connected by reinforced lines.

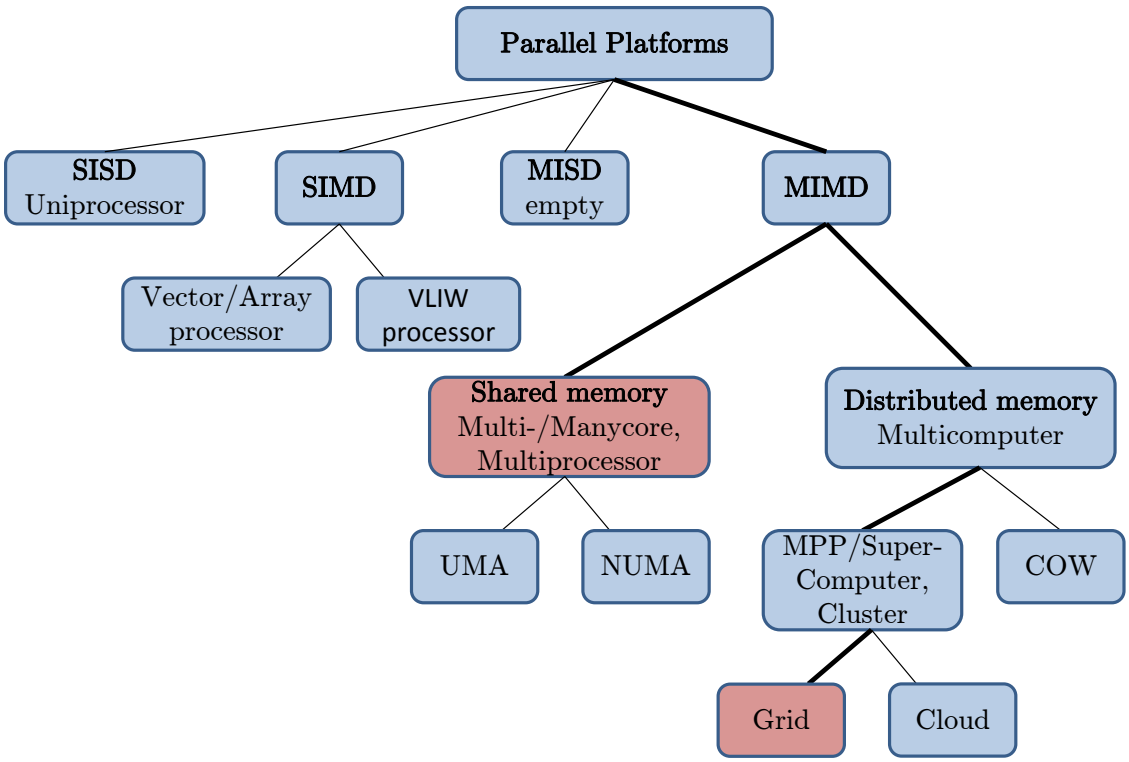

Figure 2.2: Tanenbaum Classification of Parallel Platforms [171].

Sometimes parallel platforms are classified in *homogeneous* and *heterogeneous* systems, according to the composition of processing units. Homogeneous systems consist of identical processing units where as in heterogeneous systems different types of processing units are used. Examples can be found for shared memory systems like the big.LITTLE technology [11] in ARM's multicore processors and for distributed memory systems like clusters, grids and clouds. Another categorization is related to the application field like *embedded systems* or *real-time systems*. An embedded system is a system that is integrated into an electronic device. It executes repeatedly a single program and reacts on changes in the system's environment. It often has to produce results in real-time. Therefore, an embedded system is usually

a real-time system [179]. A major characteristic of a real-time system in comparison to a non-real-time system is the time constraint. Each program (part) in a real-time system usually has a release time at which the program becomes available and also a *deadline*, at which the completion of the program execution must be guaranteed. Thus, not only the logical correctness of an execution is important, but also the time frame for the execution has to be met. Real-time systems are typically subdivided into *hard* and *soft real-time systems*. In a hard real-time system, a failure occurs when any deadline is not met, i.e. all deadlines are guaranteed to be met. For example, the brake control system of a vehicle must be a hard real-time system, as an execution delay can lead to life-critical situations. In contrast, in soft real-time systems, a completion of the execution within the deadline is desired but an exceeded deadline does not directly lead to a failure. In image processing for example, a delayed completion of an image leads to a lower quality but does not cause a life-critical situation [118].

2.2 Parallel Applications

Many real-world situations are complex and massively parallel, e.g. simulating climate changes, rush hour traffics and planetary movements or modeling vehicle constructions and forecasting the weather [22]. Numerous other examples exist in all sciences like in mathematics, physics, chemistry, biology, computer science, or economics. Such complex problems can usually be parallelized and executed onto several computers, e.g. to speed up the execution or to include more details in a model. The design and implementation of corresponding parallel applications or algorithms are an important challenge to benefit from the underlying parallel platforms.

The general process of developing parallel applications is depicted in Fig. 2.3, modeled after Sinnen [164]. Starting with the specification of an application, the design of the application is generated by decomposing the computation into parts and mapping/scheduling the resulting parts to PUs. Then, the implementation follows to obtain an executable parallel program. The generation of the design is sometimes subdivided into more than two phases. Foster [63] for example describes four phases in his PCAM method to design a parallel program. *PCAM* is an acronym for the different phases, the partition-, communication-, agglomerate- and map phase.

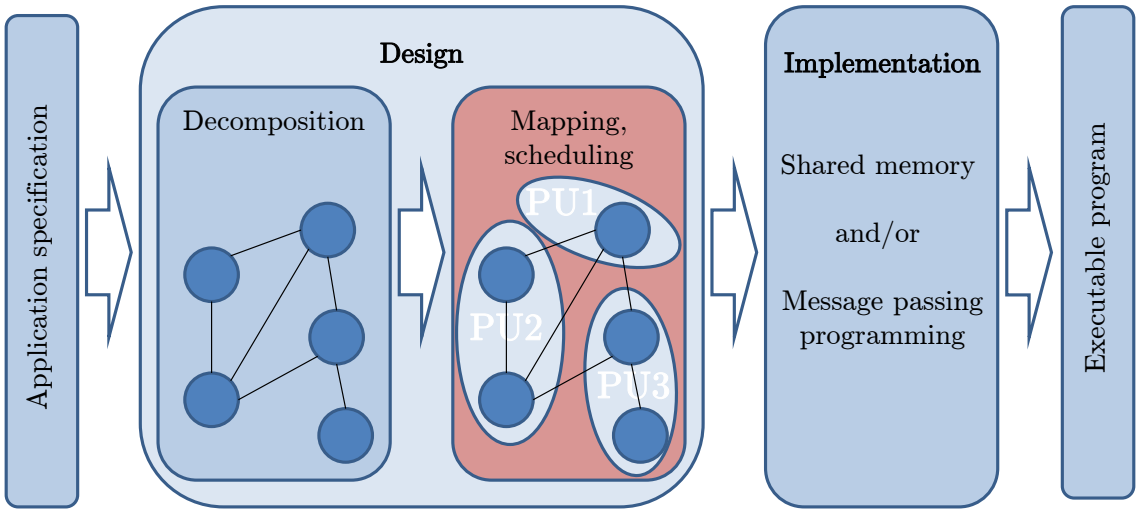

Figure 2.3: Process of Developing a Parallel Application [164].

2.2.1 Design

Typically, two steps are performed to design a parallel program.

Decomposition In a first step, the computation has to be divided into smaller parts that can be processed concurrently. This process is called *decomposition* or *partitioning*. Depending on the number and size of the parts, the *granularity* of a decomposition can be subdivided into *fine-grained*, a large number of small parts, and *coarse-grained*, a small number of large parts. Decomposition techniques can be broadly subdivided into four categories: recursive, data, exploratory and speculative decompositions. *Recursive decomposition* is usually used for problems that can be solved by a divide-and-conquer strategy, i.e. dividing the problem recursively into smaller independent subproblems followed by a combination of their results. *Data decomposition* is desired for algorithms operating on a large data structure. Depending on the type of the problem, the partitioning is applied to the input, intermediate or output data or a combination of these data. *Exploratory decomposition* is related to problems, where the underlying computations correspond to a search of a space for solutions. The search space is partitioned into smaller parts that can be searched for solutions concurrently. Finally, the *speculative decomposition* is considered for applications, whose computation branches are dependent from the output of preceding computations. Therefore, all (or a subset of all) possible branches are executed concurrently in advance so that the result of a proper branch can be chosen, when the corresponding output of the preceding computation is available [73].

Mapping/Scheduling In a second step, the parts have to be assigned to various processors, called *mapping* and the execution order has to be determined, called *scheduling*. While the determination of the execution order can only be accomplished for an existing mapping, generally both the spatial (i.e. mapping) and temporal assignment of the parts to PUs is considered, when referring to scheduling [164]. One of the major challenges to achieve a good performance is to balance the load over all processors, i.e. scheduling the parts of the computation to processors so that the execution on all processors finishes at the same time. The parts of the computation can be independent of each other so that the order of the parts is not important for the execution. Then, the parts are called *jobs*. If there are dependencies between the parts, for example the results of parts are used as input for next ones, they are called *tasks*. Next to this, the time for transferring the results has to be considered. While the transfer time on a single core is very small, it grows significantly up when using different processors that are interconnected via a LAN or the Internet. A minimization of the interaction overhead is then additionally striven, to get a good performance. Also the job and task flexibility can differ. For example, when jobs or tasks require a fixed number of resources, then they are called *rigid*. When they can be parallelized onto several PUs once during the execution, they are called *moldable*. If the number of PUs can be changed during the execution, the jobs or tasks are called *malleable* [120]. The resulting schedule is then used to create the parallel program. In Sect. 2.3 scheduling is described in more detail.

2.2.2 Models

The structure of a parallel application or algorithm is typically given by a *parallel algorithm model* that combines various techniques and strategies for the decomposition, mapping, scheduling, and for the minimization of the interaction overhead. The most common parallel algorithm models are described in the following [73]:

Task Graph Model Task graphs model a variety of parallel applications. A *task graph* $G = (V, E)$ is typically a *DAG (Direct Acyclic Graph)* that consists of nodes $v \in V$ and edges $e \in E$. The nodes represent the tasks where the workload, e.g. the execution time or number of instructions, is given as a node weight. The edges represent the dependencies between the tasks. Here, a given edge weight describes the communication costs, usually considered as transfer time. Task graphs can be statically or dynamically mapped onto PUs.

Work Pool Model In a *work pool model*, the work given as jobs or tasks is dynamically mapped onto PUs. When a PU finishes its execution, a new available job or task is assigned to it. The information of the work order is usually stored in a shared list, priority queue, hash table, or in a tree. New tasks or jobs can be dynamically added to the pool.

Master-Slave Model In this model, typically one *master*-process generates the work and allocates it to the *slave*-processes. The slave-processes execute the work and send the results back to the master. The work can be generated statically before allocating it to the slave-processes or dynamically, while the slave-processes are busy. In this model, the master-process might become a bottleneck, when the tasks are too small or the slave-processes are too fast so that the slave-processes have to wait for next tasks and cannot continue the execution directly.

Pipeline or Producer-Consumer Model The *pipeline model* is based on a chain structure. The tasks are once allocated onto the PUs. Then, each PU receives some input data, executes the corresponding task that is assigned to it and generates some output data for the next task on the next PU. Thus, a data stream is processed by this model, where the mapping of the tasks is fixed and only data changes during the execution of the whole program. This kind of application structure is sometimes called *stream parallelism*.

Data-Parallel Model In the *data-parallel model*, tasks perform similar operations on different data. Typically, the tasks are mapped and scheduled statically onto the PUs. The execution is sometimes done in different phases, where synchronization between the phases is necessary to receive new data. One example of a data-parallel algorithm is a matrix multiplication.

2.2.3 Implementation

The implementation of parallel programs is typically based on two paradigms. As parallel platforms can be classified into shared memory and distributed memory platforms (see sect. 2.1), the programming of parallel applications can be subdivided into shared memory programming and distributed memory programming, the latter called message passing.

Shared Memory Programming *Shared memory programming* is typically done with threads that use a shared address space to communicate to each other. A thread is a lightweight process that consists of a single stream of instructions. When using a shared address space over all threads, an appropriate synchronization of the memory access is necessary to guarantee the correctness of the data. Common *APIs (Application Programming Interfaces)* for shared memory programming are *POSIX* threads *(Portable Operating System Interface for uniX)* [21] and *OpenMP (Open Multi-Processing)* [136].

Message Passing Programming In contrast, in *message passing programming*, the communication between the processes is explicitly conducted using messages. Each process has its private address space for the data. The communication is basically done with send and receive operations to transfer data from one process to another. The *MPI (Message Passing Interface)* [127] is typically used for this kind of programming.

These two paradigms are related but not restricted to its corresponding class of parallel platforms. Thus, it is possible to use message passing on a shared memory platform or vice versa shared memory programming on a distributed memory platform. But usually, this leads to a lower performance. Often both paradigms are combined in one program, e.g. when using a multicomputer with multiprocessor nodes [73].

2.3 Scheduling

[1]In general, scheduling is the process of assigning activities to resources in time. Examples can be found in various application fields like in production planning and manufacturing, booking systems, or in a simple diary. In computer science, scheduling often describes the spatial and temporal assignment of computational parts onto different PUs (see Sect. 2.2). The corresponding *scheduler* that manages the scheduling process can be realized in either hardware or software. The information about where and when the parts of the computation should be executed, are then stored in a *schedule* [44].

Scheduling can be used for several challenges like minimizing the overall completion time of an application (*makespan*), for throughput constraints, or minimizing

[1]The organization of this section is partially based on the presentation in my German diploma thesis [44].

the energy consumption of a computer system, to mention only a few. Often a combination of multiple challenges, usually two or three, is desired.

A corresponding scheduling algorithm is typically based on a model that contains at least particulars of the target system architecture and of the parallel application like explained in Sect. 2.1 and Sect. 2.2. Such particulars for example include whether the target system is a shared memory or a distributed memory system, whether the system consists of homogeneous or heterogeneous PUs, or how the PUs are interconnected to each other. For the parallel application, particular features are important like whether there are dependencies between the parts of the computation or if there exist transfer times or costs. [44]

Scheduling is a NP-complete problem. Finding an optimal solution is typically very compute intensive, especially for problems with a large number of computational parts and PUs. Therefore, instead of finding an optimal solution, often approximation algorithms or heuristics are used to get a satisfying solution that can be found within an acceptable period of time.

Every year, hundreds of new scheduling approaches for different parallel systems and applications with various constraints and objective functions are published so that a summarization of all scheduling variants is impossible. Also several taxonomies with various focuses are proposed in the literature to categorize the scheduling algorithms like in [38], [50], [110], [120], [156] or [173]. However, a complete description of all taxonomies would go beyond the scope of this thesis. Instead, in the remaining part of this section a classification of scheduling algorithms for parallel applications is described that includes differentiations important for this thesis. Furthermore, exemplary representatives for each class are given or it is sometimes referred to corresponding techniques in the literature. Finally, key figures are presented to validate different scheduling techniques.

2.3.1 Classification

Scheduling algorithms can be subdivided into several classes. In Fig. 2.4 a taxonomy of scheduling algorithms essentially based on [84] is shown. The classes that are connected by reinforced lines represent the focus of this thesis and, therefore, are described in more detail. For the remaining classes, typical properties are described. A further subdivision of these classes is left out for reasons of clarity.

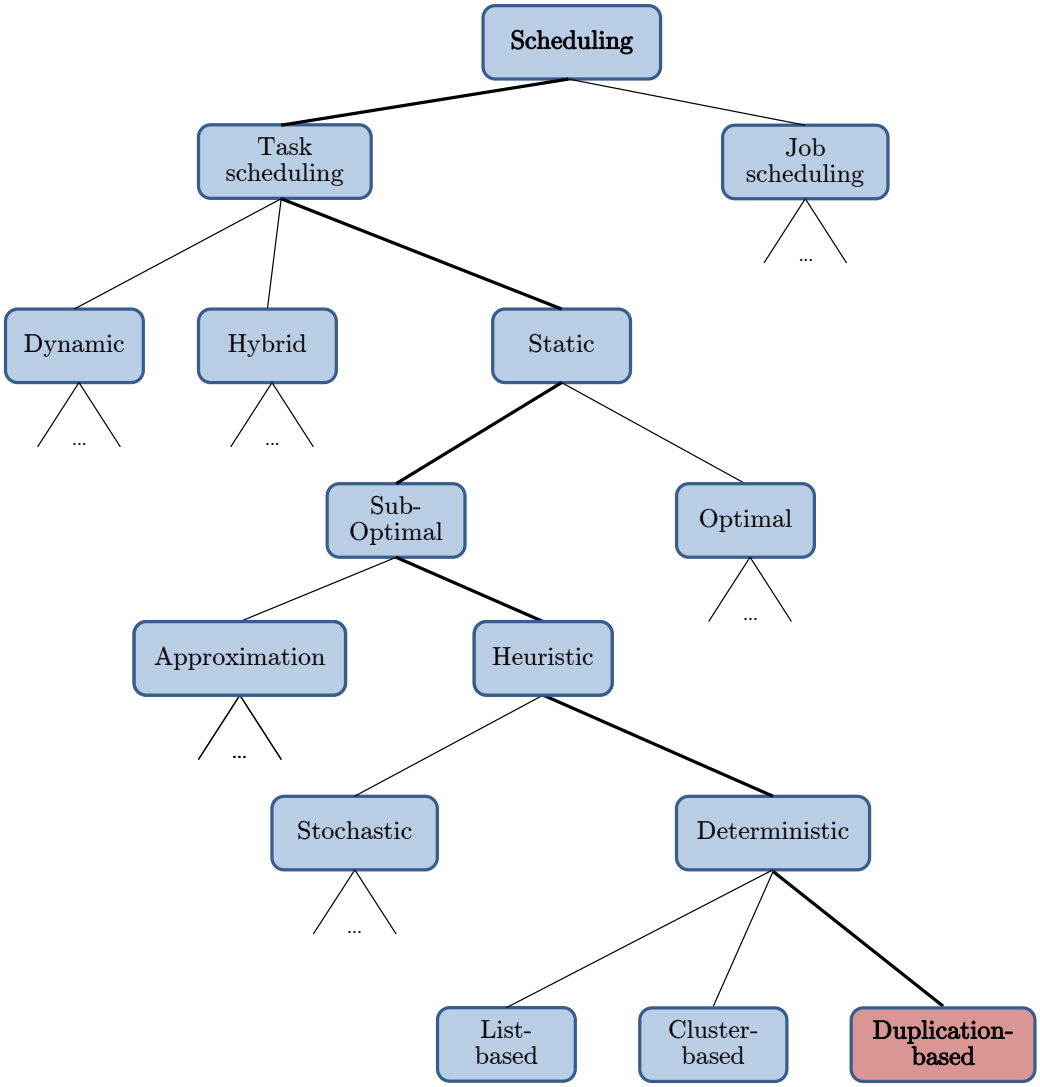

Figure 2.4: Taxonomy of Scheduling Algorithms [84].

Task vs. Job Scheduling A first categorization of scheduling algorithms is related to the dependencies between the parts of a computation. If the computation consists of independent parts (jobs), then the scheduling is also called *job scheduling*. A job can be one part of a parallel program, but it also can for example be related to a group of combined parts like bags of tasks, to a whole program or to a workflow. However, jobs can usually be scheduled without any specific order[2]. Therefore, a typical goal of job scheduling is the optimization of the overall system performance [110]. Often (meta) heuristics are applied to find an optimized solution.

[2]There exist a few exceptions e.g. in real time systems.

Braun et al. [34] present eleven different job scheduling heuristics for heterogeneous distributed computing systems. The scheduling is done statically, i.e. prior to execution. The goal of the heuristics is to minimize the total execution time.

For computational grids, Subramani et al. [168] present another job scheduling algorithm. They use distributed scheduling algorithms with multiple simultaneous requests to improve the performance.

Gao et al. [64] present adaptive job scheduling in grid environments based on two algorithms. They use an algorithm for the system-level to decide on which node a single job should be executed in a shortest time and a genetic algorithm for the application-level that is used to minimize the average completion time of all jobs. Two models for a service Grid are designed to predict the completion time of jobs.

In [188], Zaharia et al. present two job scheduling techniques for multi-user map reduce clusters. With these techniques, they try to improve the data locality and the throughput. One proposed approach is delay scheduling, where jobs with local data are chosen first, before also considering jobs with non-local data. The other approach is copy-compute splitting, where the jobs are divided into two types (copy tasks and compute tasks), according to their operations.

For job scheduling on computational grids, Pooranian et al. [143] present a hybrid meta heuristic algorithm. They combine a genetic algorithm for searching the problem space globally and gravitational emulation for local search. With their approach, the runtime and number of submitted tasks that miss deadlines are decreased.

Shojafar et al. [161] present a meta-heuristic job scheduling approach for cloud environments to minimize the makespan. The scheduling algorithm, called FUGE[3], is based on a genetic algorithm combined with fuzzy theory to optimize the load balancing in terms of execution time and cost.

In [72], Goswami et al. present a deadline stringency based job scheduling approach for computational grids that is based on an already existing so-called *NDFS (Nearest Deadline First Served)* algorithm. They try to improve the dynamic load balancing by simultaneously receiving job requirements and collecting runtime status informations of the resources for the allocation of the jobs.

Lopes and Menascé [120] propose a current approach to categorize job scheduling algorithms. They present the trends of job scheduling research for the last decade based on over 1,000 analyzed job scheduling papers and provide a taxonomy of job scheduling. They classify the most cited 100 problems with their taxonomy into ten groups, that differ next to others in the used environment and the structure

[3]The authors omit a long form of the term FUGE.

of jobs. Additionally, they give for each group the proposed solutions with typical properties.

When there exist dependencies between the parts (tasks), the scheduling is called *task scheduling*. Task scheduling is often done with the help of a task graph that represents the structure of a parallel program (see Sect. 2.3). The information of the task graph is then used to create an appropriate schedule. Therefore, task scheduling is sometimes called *task graph scheduling*. A common challenge of task scheduling is the minimization of the makespan. (Task) Scheduling can further be subdivided by the time, when the scheduling process is done, into *static* and *dynamic* scheduling.

Static vs. Dynamic Static scheduling is done prior to execution, sometimes called *offline*. In contrast, dynamic scheduling is done during the execution, called *online*. While the static scheduling does not influence the runtime of the corresponding application, it cannot react as the dynamic scheduling on unpredictable situations that might occur during the execution. However, the calculations for the dynamic scheduling prolong the execution of the parallel application and thus the makespan of the schedule [84][110]. Dynamic scheduling is mainly used for problems, where no information about the tasks is known prior to execution. Therefore, the goal of dynamic scheduling is usually Load Balancing.

For example, Ramamritham et al. [151] describe a dynamic task scheduling approach for hard real-time distributed systems. They assume that each node within the system has its own scheduler and a set of periodic tasks that are guaranteed to meet their deadlines. Additionally, aperiodic tasks can arrive at any time on any node in the system. The scheduler on the corresponding node checks whether the task can be assigned to that node without violating the deadline constraints. If the task cannot be allocated suitably on the local node, the scheduler interacts with the schedulers of other nodes by using a bidding scheme to determine on which node the task can better be allocated. Then, the task is sent to the corresponding node.

Manimaran and Murthy [125] present another approach for real-time systems. They assume to have a multiprocessor system and aperiodic tasks that can arrive at any time. A dynamic scheduler is used to allocate the tasks onto the processors. Instead of restricting on non-parallelizable tasks, the tasks can be parallelized onto several processors. The scheduler takes advantage of the task parallelization to meet the deadline constraints.

In [150], Rahman et al. present a dynamic scheduling algorithm for workflow applications on global grids. They extend an already existing scheduling approach

for homogeneous systems to include heterogeneous and dynamic environments. The scheduling is done stepwise by calculating dynamically the critical path in the workflow task graph and prioritizing tasks to respect the dependencies.

For grid computing systems, Zhang et al. [192] present a dynamic task scheduling algorithm. They extend a two phase algorithm that originally is used for static scheduling. The algorithm first selects the tasks and then the processors to allocate the tasks. The tasks are related to levels, so that equal levels indicate independent tasks, otherwise there exist dependencies between the tasks.

Amalarethinam et al. [8] present a DAG based dynamic task scheduling algorithm called *DYTAS (DYnamic TAsk Scheduling)* for multiprocessor systems. The scheduling is done by using several different task queues: One initial queue, a dispatch and completed queue, and individual processor task queues. After starting the scheduler, the tasks are ordered by their dependencies. Then, the processor queue is dynamically chosen, where the next task can finish at the earliest time. When the processor queues become empty, the algorithm stops.

In contrast, static scheduling can only be done, when information about the tasks and there dependencies are known in advance. Sometimes also a *hybrid* variant is used, i.e. a combination of static and dynamic scheduling like in [9], [116], [133] or [191]. (Static) scheduling can further be subdivided by the quality of the solutions, into *optimal* and *sub-optimal* scheduling.

Optimal vs. Sub-optimal Most of the scheduling problems for parallel applications are NP-hard [164]. This means that no polynomial-time algorithm exists to solve the problems optimally (unless P = NP). While for small instances of those problems optimal solutions might be found within an acceptable period of time, it is infeasible for larger instances.

Wang et al. [181] present an optimal task scheduling approach for streaming applications on MP-SoCs. They describe the problem as an *ILP (Integer Linear Programming)* formulation. The objective is to minimize the makespan by avoiding inter-core communication overhead. Furthermore, they also try to reduce the energy consumption by using *DVS (Dynamic Voltage Scaling)*.

Shioda et al. [160] present an optimal task scheduling algorithm for parallel processing. They propose an ILP formulation, where the objective is the minimization of the makespan. With their approach, they also try to minimize idle times.

In [85], Hönig and Schiffmann present a fast optimal task graph scheduling approach for homogeneous computing systems. They use a parallel variant of an

informed search algorithm based on an *IDA**-algorithm *(Iterative Deepening A*)*, which is a memory-saving derivative of the well known A*-algorithm [78]. The objective is to optimize the makespan. Their approach is not restricted to task graph problems with a small number of tasks, but complex task graphs can be computed by their IDA*-algorithm as well.

Cho et al. [42] propose an optimal real-time scheduling algorithm for multiprocessors called *LLREF (Largest Local Remaining Execution First)*. The algorithm is based on a fluid scheduling model, i.e. each task executes at a constant rate at all times. To describe the task execution behavior on multiprocessors, the algorithm uses time and local execution time domain planes (T-L planes). They show that scheduling for multiprocessors can be viewed as repeatedly occurring T-L planes, where a feasible solution for a single T-L plane results in an optimal schedule.

Next to optimal algorithms, two sub-optimal approaches are usually used to find at least optimized solutions in a short period of time. The first one is to use *approximation algorithms* to find an approximation of the optimal solution, the second one is to use *heuristics* that make their decisions based on predefined assumptions.

Approximation vs. Heuristic In approximation algorithms, the objective to find an optimal solution is slightly relaxed for finding solutions that are close to the optimal. However, instead of searching the whole solution space to find an optimal one, the algorithm stops, when a solution is found that is rated as "good" enough. Typically a factor p is given that describes the guarantee on how close the solution to the optimal at least will be. Therefore, these algorithms are sometimes called *p-approximation algorithms*.

Giroudeau et al. [69] present in their work an approximation for precedence constrained scheduling problems with large communication delays. They focus on multiprocessor systems and try to minimize the makespan. A lower bound for the performance of an approximation algorithm is given by using a variant of the problem *3SAT (3-SATisfiability)* and an impossibility theorem. They propose a polynomial-time algorithm with a performance ratio of $\frac{2(c+1)}{3}$ with $c \leq 2$, where c denotes the communication delay between two tasks.

Chen and Chu [40] present another approach for scheduling malleable tasks. They assume that the processing time of a malleable task is non-increasing and the work non-decreasing in the number of processors. Their polynomial-time approximation algorithm can yield an approximation ratio of 3.4142 and for strictly decreasing processing times in the number of processors the ratio is 2.9549.

For scheduling malleable tasks under precedence constraints, Lepère et al. [114] present an approximation algorithm. They demonstrate a close relationship to the allotment problem and design a polynomial time approximation algorithm with a performance guarantee close to 2.61803 for different special cases and with a guarantee of 5.23606 for the general case.

In [87], Hunold presents a scheduling algorithm for moldable tasks on multiprocessors. The objective is the minimization of the makespan. He tries to reduce the critical path while keeping the overall work small by introducing a relative runtime threshold that defines the minimum runtime improvement of a larger allotment.

For scheduling malleable tasks under general precedence constraints, Jansen and Zhang [94] present an approximation algorithm. They developed an improved version of their previous approximation algorithm with a ratio of 4.730598.

Günther et al. [74] present in their work a scheduling and packing approach for malleable tasks with precedence constraints of bounded width.

In contrast, heuristics make their decisions based on predefined assumptions, i.e. a heuristic follows fixed rules that are defined based on experiences with the treated problem type. Therefore, heuristics are usually very fast in finding a solution, but there is no guarantee to find an optimal or at least a near to the optimal solution. The results of a heuristic are usually compared to results of other heuristics or to optimal solutions to get informations about the quality of the heuristic. Heuristics can further be subdivided into *stochastic* and *deterministic* methods.

Stochastic vs. Deterministic Stochastic methods try to find a possibly optimal solution with the help of random-based algorithms. During the runtime, either exactly one solution can be considered or several alternative solutions [44]. Techniques that only consider one solution are Tabu Search [70] and Simulated Annealing [158]. Representatives for considering several solutions are Genetic Algorithms [62] and Swarm Algorithms [137].

Blum [29] presents in his work an introduction and recent trends of a swarm algorithm, the so-called *ACO* algorithm *(Ant Colony Optimization)*. In the nature, ants produce messenger substances, so-called pheromones, to find a way to some food and back to their colony. Over time, a shorter path results in a more dominant trace and the probability increases that other ants will also follow this trace. This behavior leads to an optimized solution. In the ACO algorithm, the ants are represented by agents and the pheromones by values depending on the quality of a solution. The values are then stored in a so-called pheromones-matrix.

Singh [162] for example presents a genetic algorithm for task scheduling in parallel systems. He considers dependencies between tasks and assumes to have no communication costs. As target system, he uses multiprocessor systems with heterogeneous processors.

For scheduling tasks with precedence constraints, Omara and Arafa [135] present two genetic algorithms. They try to improve the performance of the genetic algorithms by adding heuristic principles. For the first algorithm, two fitness functions are used in a row with the objectives to minimize the makespan and to balance the load. The second algorithm is extended by a duplication-based technique to reduce the communication overhead.

Daoud and Kharma [48] present another genetic algorithm for task scheduling in heterogeneous distributed systems. They use a customized genetic algorithm to produce optimized schedules. The algorithm starts with a scheduling heuristic for the initial population. The schedule is then located at an approximate area in the search space that is used for the genetic algorithm to improve the schedule.

Porto and Ribeiro [144] present a tabu search approach for heterogeneous processors. They consider task scheduling and show that their approach results in solutions that are improved by up to 30 percent compared to the most appropriate algorithms.

Deterministic methods, in contrast, can further be classified into *list-based*, *cluster-based* and *duplication-based* algorithms.

List- vs. Cluster- vs. Duplication-based In list-based method, nodes (tasks) are ordered with priorities in a scheduling list. To prioritize tasks, different approaches are proposed like *CP (Critical Path)*, *HLF (Highest Level First)*, *LP (Longest Path)* or *LPT (Longest Processing Time)*. List scheduling can be done statically or dynamically. In the static variant, the tasks are obtained prior to execution from the beginning of the list and assigned to PUs where they can earliest start. However, the dynamic variant consists of three steps: prioritize unscheduled tasks, select the task with the highest priority and allocate it to a PU. After each allocation, the scheduling list is rearranged. The priorities are assigned with the help of task attributes. The most common are the so-called *t-level (top level)* and *b-level (bottom level)*. The t-level is the longest path between an entry node and the current node, by considering the node and edge weights. The b-level is the longest path between the current node and an exit node and is bounded by the critical path [110].

Adam et al. [2] for example present list-based approaches. They propose a *HLFET*-method *(Highest Level First with Estimated Times)*. In this method, a static b-level, so-called *sb-level*, is used to prioritize the tasks in a scheduling list. In contrast to the common b-level, the sb-level only considers the node weights. After prioritizing the tasks, they are allocated to the PUs by a greedy mapper. A *greedy mapper* assigns each task to the PU that can finish the execution of a task first.

Kasahara and Narita [101] present an enhanced version of the HLFET-method. They propose a so-called *CP/MISF*-method *(Critical Path/ Most Immediate Successors First)*. In contrast to the HLFET-method, tasks with same priorities are sorted in the scheduling list by the number of direct successor tasks and not randomly.

In [49], Daoud and Kharma present a list-based algorithm for heterogeneous distributed systems. They consider dependent tasks with communication costs and try to minimize the total execution time. The so-called *LDCP (Longest Dynamic Critical Path)* algorithm consists of three phases. In each scheduling step, a task selection, processor selection and a status update is done. The status update is necessary, because only when a task is assigned to a processor, the computation time is known and communication costs of succeeding tasks can be updated.

Macey and Zomaya [124] present a performance evaluation of list scheduling algorithms. They compare the performance of different popular list scheduling heuristics like HLFET and others. Their results demonstrate the inadequacies of the list-based heuristics in communication-intensive cases.

For heterogeneous target systems, Bjørn-Jørgensen and Madsen [27] present another approach. They propose an algorithm called *HCP*-algorithm *(Heterogeneous Critical Path)* that considers both concurrently the interprocessor communication and the heterogeneity of the target system.

In [165], Sllame and Drabek present a list-based scheduling algorithm for high-level synthesis. This algorithm produces near-optimal schedules by including information extracted from a data flow graph (like the number of successors and predecessors) into the priority functions of the scheduler.

In cluster-based task scheduling, the makespan of a schedule is optimized by combining tasks suitably to reduce communication costs. Therefore, the tasks are combined to so-called *clusters* that are merged together until no further reduction of the makespan is possible. In general, there are two different methods. The *UNC-method (Unbounded Number of Clusters)* and the *BNP-method (Bounded Number of Processors)*. In the former method, an unlimited number of processors is assumed to get a high reduction of the makespan. A post processing step is then necessary

to map the clusters onto the available processors. In contrast, in the BNP-method, the existing target system is directly considered so that no post processing step is required [84][110].

For an unbounded number of processors, Yang and Gerasoulis [186] present a cluster-based algorithm. In the so-called *DSC*-algorithm *(Dominant Sequence Clustering)*, a task is assigned to a cluster when the longest path between an entry task and the current task is minimized by saving the communication costs. Therefore, the critical path of the partial schedule (called *dominant sequence*) is used. If the task is not on the critical path, it must be proved whether the assignment leads to a worsening of tasks with a higher priority. Is it not possible to assign a task to an existing cluster, the task is assigned to a new cluster. Gerasoulis and Yang present in a further work a comparison of clustering heuristics for scheduling DAGs on multiprocessors [68].

Cirou and Jeannot [46] propose an approach for heterogeneous systems. They present a multi-step scheduling algorithm called *triplet*. Initially, the tasks are combined to clusters. Then, platforms with same characteristics are grouped to workstation clusters. Finally, the clusters are mapped to the workstation clusters.

In duplication-based scheduling, so-called *TDB (Task Duplication-Based)*, also a reduction of the communication costs is desired. In contrast to the cluster-based scheduling, the tasks are replicated. If dependent tasks are on the same PU, no communication costs are needed. If a task has several descendants, only duplicates can avoid communication costs (and serialization). Some algorithms replicate only direct predecessor tasks, others in turn all predecessors. The replicated tasks (duplicates) are then mapped onto different PUs. If a successor task is placed on the same PU like the duplicate, the communication costs between them can be neglected [44].

Papadimitriou and Yannakakis [138] present a duplication-based algorithm called *PY*-algorithm *(Papadimitriou Yannakakis)*. The tasks are assigned to PUs by using a so-called *e-value* that represents the earliest start time of a task. If a task can only be started after the e-value because of dependencies, the predecessors are duplicated and placed onto the same PU like the considered task. Thus, the communication costs are eliminated and the makespan is reduced.

Bozdag et al. [33] present a task duplication-based algorithm called *DUPS (Duplication-based scheduling Using Partial Schedules)*. The algorithm consists of two phases: one for minimizing the schedule length by using partial schedules and one for reducing the number of required processors by merging and eliminating the schedules. Duplicates are created in the first phase to construct partial schedules.

In [32], Bozdag et al. present a further duplication-based algorithm called *DBUS (Duplication-based Bottom-Up Scheduling)* for heterogeneous environments.

In [109], Kwok presents a task duplication-based approach for heterogeneous clusters. To minimize the makespan, tasks on the critical path are replicated onto faster machines.

Ahmad et al. [4] present a comparison of different algorithms for task graphs on parallel processors. Next to task duplication-based algorithms, list- and cluster-based algorithms are considered.

Kaur and Kaur [103], and Gupta et al. [75] present a review of duplication-based scheduling algorithms.

Often a combination of list-, duplication- and cluster-based methods is used. Kruatrachue et al. [108] and Bansal et al. [19] present combinations of duplication- and list-based scheduling algorithms.

2.3.2 Performance and Cost Metrics

In order to make a statement about the quality of a scheduling algorithm, various factors can be considered. Depending on the application field and the resulting type of scheduling, different factors are important, e.g. the performance, scalability, cost or throughput. Several metrics are proposed in the literature like in [36], [60], [90], [102] and [107] to validate the quality and thus to make a scheduling algorithm comparable against existing ones. In the following, necessary performance and cost metrics for this work are described.

Makespan The *makespan* or *schedule length* of a schedule is the overall completion time CT of a schedule. It describes the length between the starting point of an execution and the ending point. In a parallel program, where the tasks of the program are executed on different PUs, the makespan is the length between the earliest starting task of the schedule execution and the latest ending task over all PUs [39]. The makespan can be described by equation 2.1, where i is the task index:

$$Makespan = max(CT_i), \tag{2.1}$$

Speedup A common performance metric for scheduling algorithms is the *speedup*. Important for the speedup is the *schedule length* (i.e. the makespan of a schedule). The speedup of a parallel program and thus of a schedule is then the ratio between

the sequential schedule length SL_{seq} and the parallel schedule length SL_{para} [73][184]. It can be described by equation 2.2:

$$Speedup = \frac{SL_{seq}}{SL_{para}} \tag{2.2}$$

In the literature, different types of speedups like absolute or relative speedup are described, depending on different assumptions to the sequential schedule length. In this work, SL_{seq} is defined as the sum of all task execution times.

Efficiency The efficiency of a scheduling algorithm is defined by the ratio between the speedup and the number of PUs. Thus, it represents the average usage rate of all PUs of the system during the execution of a schedule [73].

$$Efficiency = \frac{Speedup}{\#PUs} \tag{2.3}$$

Scheduling Time A common cost metric for scheduling is the *scheduling time*. It represents the running time of an algorithm to find a resulting schedule [174].

2.4 Fault Tolerance

[4]A *fault* is the cause of an error. Examples are a transistor defect or an incorrect line of program code. An *error* is the visible effect of a fault like a wrong behavior of an application due to the incorrect code line. Faults can occur at any time in a computer system. Some faults have a major effect on the behavior of a system, e.g. a damaged or overheated processor leads to a *failure*, while others affect the system only marginally, e.g. a wrong presentation of some results in an application [51][167].

Therefore, it is important to classify faults to handle them appropriately [52]. In addition, the application field of a system is important for the handling of a fault. For example in a medical or industrial field a faulty system or at least a faulty component (in hardware or software) leads to high costs in sense of time, money up to life-critical situations. In such application fields, a correct system behavior is mandatory.

[4]The organization of this section is partially based on the presentation in my German diploma thesis [44].

In general, there exist two approaches for the handling of faults, the fault avoidance and the fault tolerance. The *fault avoidance* is typically used in the planning and development phase of hardware and software in order to avoid faults directly in advance, e.g. with additional tests and systematic designs. In contrast, *fault tolerance* is used to detect and treat faults after they have occurred, so that the system still functions correctly (despite faults) [51][52][167].

In parallel systems with a large number of PUs like in grids or manycores, where a failure of a PU can occur during the execution of a scheduled parallel program, fault tolerance is very important. In such systems, unexpected faults often occur during operation, e.g. failures due to hardware, software and connection faults or due to the shutdown of PUs by their owner (in a grid), so that a fault avoidance is impossible in advance. In addition, the probability of a fault increases with the number of PUs [51]. Therefore, several approaches for fault-tolerant scheduling are presented in the literature.

Whenever fault tolerance is used, some kind of *redundancy* is necessary, e.g. by including additional components in hard- or software, by supporting special functions, or by using extra time for fault-tolerant techniques. Basically, redundancy describes the existence of additional resources that are dispensable for a correct functioning of a system in general, i.e. in a fault-free case [30][52]. To develop and evaluate fault-tolerant techniques, *failure models* are often used. A failure model describes the assumptions about the system in use and the tolerable faults that should be supported by the system [66].

In this section, firstly a classification of faults is given and different failure models are explained. Secondly, fault-tolerant scheduling and corresponding approaches are presented. Finally, difficulties in real implementations by using MPI are described.

2.4.1 Classification of Faults

There exist different faults that can occur in a computer system. Faults are usually categorized into several classes according to their properties. These classes can be used to develop fault-tolerant techniques for a group of faults. In the following, the most common categories are described. A first classification can be done according to the cause of a fault into *design*, *production* and *operation faults* [51][52]:

- Design Fault:
  A design fault is caused in the planning and designing phase of a system by a wrong specification, documentation or mistakes in the implementation.

For example an incorrect algorithm or architectural specifications that do not match the requirements for the application field of a system.

- Production Fault:
  Faults that are caused in the manufacturing of a system are called production faults. For example inaccuracies in the fabrication process of hardware like processor chips or faults from the replication process of software, when copying an application to another medium.

- Operation Fault:
  Operation faults are caused during the runtime of a system, i.e. the system was initially faultless. For example wear-out and aging failures, mistakes in the use of programs or a wrong handling in the maintenance.

Faults can also be subdivided into *software* and *hardware faults* [51]:

- Hardware Fault:
  A hardware fault is caused by physical defects in the system or component. For example broken pins of a processor, defect memory cells in a RAM module, or short-circuits.

- Software Fault:
  Faults that are caused in the software are related to this category. The software can be for example a standalone application, an operating system or drivers for hardware components. Most software faults are caused by implementation mistakes in the design phase or due to incorrect updates.

Another differentiation of faults is based on the duration of a fault into *temporary* (transient and intermittent) or *permanent faults* [51]:

- Temporary Fault:
  Faults that occur for a short time during the life cycle of a hard- or software are called temporary faults. If a fault occurs only once it is also called a *transient fault*. These faults typically arise due to external influences like cosmic rays or electrical power drops. If a fault occurs periodically, it is also called *intermittent fault*. These faults are usually based on implementation mistakes or unexpected operation conditions.

- Permanent Fault:
  A permanent fault exists until the state of the corresponding hard- or software is changed by restarting, changing or replacing the component.

2.4.2 Failure Models

In the literature, several failure models are presented, e.g. in [51], [66], [166] and [176]. Therefore, only the most common are described in the following [66]:

- *Fail-stop* Model:
  In this model, a failure of a PU can occur at any time. The information about a failure is distributed to the other PUs automatically, as the faulty PU cannot send or receive messages anymore. A recovery of the faulty PU is impossible.

- *Crash* Model:
  The crash model is similar to the fail-stop model. But the information about a failure is not distributed automatically. The remaining PUs only get the information about a failure, when they do not receive any response of the faulty PU to a request within a time frame.

- *Crash Recovery* Model:
  In contrast to the crash model, a faulty PU can be recovered in this model after a while.

2.4.3 Fault-tolerant Scheduling

In fault-tolerant scheduling, the most common software techniques are checkpointing and replication. *Checkpointing* is a *backward error recovery* technique. With this technique the system state is saved periodically in a checkpoint. When a fault occurs, the faulty system state is replaced by the last fault-free checkpoint and the execution is continued [148].

In *replication*, the fault tolerance is achieved by copying parts of an application onto other PUs. In case of a failure of one PU, either the original or the copy finishes. The execution of the application can continue. This technique can be subdivided into active and passive replication. In *active* replication, multiple copies of a program part are mapped onto different PUs and executed in parallel. In this case, a fixed number of failures is tolerated. *Passive* replication uses only one backup copy per program part. The backup copy is not executed in parallel with the original. Instead, it only runs if the original fails [16].

If only one copy is used in general, replication is also called *duplication* and the copy of each program part is called a *duplicate*. This scheduling approach is then

related to task duplication-based scheduling (see Sect. 2.3.1). Sometimes hybrid techniques that combine checkpointing and replication are used.

As a fault or failure can only be survived or not, often the (performance) *overhead* in both the fault-free and the fault case are used to evaluate the fault-tolerant scheduling approaches. The overhead is the difference between the makespan without any fault-tolerant aspects m and the makespan with fault tolerance included m_{ft} in percent and can be expressed by [44][53]:

$$Overhead = \frac{m_{ft} - m}{m} \cdot 100. \tag{2.4}$$

In this equation, no difference between the fault-free and fault case is considered. Therefore, the relation of the overhead to the corresponding case is always specified in this thesis, if it is not clear in the context.

Fault-tolerant scheduling has been studied over decades, thus numerous approaches exist in the literature for both checkpointing and replication. Therefore, only some examples are presented in the following:

Approaches using Checkpointing Poola et al. [141] present a fault-tolerant scheduling approach for workflows in cloud computing. They minimize the execution costs by using different cloud instances for a workflow and tolerate failures by including checkpoints.

In [17], Balpande and Shrawankar present a fault-tolerant job scheduling approach for computational grids. They propose two different checkpointing schemes. The so-called *ABSC (Algorithm Based on System Checkpointing)* scheme is based on a genetic algorithm and used for non-intensive applications. In the so-called *ABAC (Algorithm Based on Application Checkpointing)* scheme a computational intensive application is extended to make and store checkpoints.

For computational workflows, Aupy et al. [13] present another approach. They propose a polynomial-time optimal algorithm for fork DAGs. A *fork* DAG is a task graph with one entry task, n exit tasks and n edges from the entry task to each exit task. Their goal is to minimize the expected execution time for a workflow by making decisions about the execution order of tasks and when to checkpoint.

Prashar et al. [145] present a fault-tolerant approach in grid computing by combining checkpointing with an ACO algorithm. The ACO algorithm is used to balance the load over nodes in a grid. Checkpointing is considered to handle faults of resources.

For computational grids, Babu and Rao [14] present an automated checkpointing strategy. They assume a heterogeneous system and independent jobs to be scheduled. MPI and *BLCR (Berkeley Lab Checkpoint/Restart)*, a package to support interactive checkpointing on MPI applications, is used.

In [163], Singh presents a fault-tolerant scheduling algorithm for grids with checkpointing. He considers the failure rate and the computational capacity of resources to minimize the makespan and cost of a schedule.

Nazir et al. [132] present an adaptive checkpointing strategy for economy based grids. They propose a fault index of grid resources that is dynamically updated based on successful or unsuccessful execution of tasks. The fault index is used for the intensity of checkpointing, i.e. the checkpoint interval.

Further approaches using checkpointing can be found e.g. in [65], [146] or [149].

Approaches using Replication Litke et al. [117] present an efficient static fault-tolerant task replication approach for mobile grid environments. They consider multiple replications per task depending on the probability of a failure. The goal is to maximize the utilization of the grid resources.

For multiprocessors, Jun et al. [98] present another fault-tolerant approach. They propose two different scheduling methods for both active and passive replication. The first method is based on an ILP formulation to find optimal schedules, the second one is a heuristic algorithm that can achieve close to optimal solutions.

In [177], Tsuchiya et al. present a dynamic fault-tolerant replication-based scheduling technique for real-time multiprocessors. They assume aperiodic tasks and consider one copy per task. The copy is divided into two parts: one part that can be executed simultaneously to the original task (so-called *redundant part*) and another part (so-called *backup part*) that runs after the original task. Thus, the backup part has only to be executed when the original task fails.

Ahn et al. [5] present a fault-tolerant scheduling approach for hard real-time systems. They use passive replication and also consider aperiodic tasks. In contrast to Tsuchiya et al., an overlapping of an original task and a copy of another task is allowed in their dynamic scheduling heuristic.

In [1], Abawajy presents a dynamic fault-tolerant scheduling approach for grid systems called *DFTS (Distributed Fault-Tolerant Scheduling)*. He assumes a fail-stop model and a certain number of replicas given by the user.

In [183], Wensley et al. present a fault-tolerant approach for aircraft control called *SIFT* (*Software Implemented Fault Tolerance*). They use triplication of tasks to

tolerate a single failure of a processing unit or bus. Iterative tasks are assumed and redundantly executed. Consequently, the results of each iteration are initially voted before they are used.

Poola et al. [142] present a fault-tolerant approach for workflows in cloud environments. They propose a just-in-time scheduling heuristic that uses replication to utilize different pricing models offered by clouds.

Further investigations on fault-tolerant scheduling with replication can be found in [25], [31] and [182].

Combinations Nandagopal and Uthariaraj [130] present a combined fault-tolerant approach for computational grids. They use checkpointing to tolerate failures and replication to increase the availability of checkpoints.

Nazir and Khan [131] present a fault-tolerant job scheduling approach for computational grids. They consider checkpointing and replication. In their approach the intensity of both is dependent on the fault occurrence history of a resource.

In [43], Chtepen et al. present an efficient fault-tolerant approach that combines checkpointing with replication. They propose to dynamically switch between both techniques according to the runtime information and system load. Next to their proposed algorithm several checkpointing and replication approaches are presented that dynamically adapt the checkpointing interval and the number of replicas.

Further combined fault-tolerant approaches can be found in [15] and [187].

2.4.4 Fault Tolerance in MPI

We consider a fault-tolerant MPI-application that can survive a failure. In the basic version, MPI does not support fault tolerance. When an MPI-process fails for some reason, the typical behavior of MPI is to abort the whole application, i.e. all remaining MPI-processes. This behavior can be changed by using an additional flag (`-gmca orte_abort_on _non_zero_status 0`), when starting the MPI-application with `mpirun *`. The execution of an MPI-application is then continued. But in this case, no further collective functions can be used, as the standard intra-communicator is MPI_COMM_WORLD, that includes all started MPI-processes and is static during the runtime. In every MPI-application the parallel part must be started and ended with the collective functions `MPI_Init()` and `MPI_finalize()` that are related to the standard intra-communicator. If an MPI-process fails within this part, the `MPI_finalize()` function leads to an endless loop, because the failed

MPI-process did not reach this function. One option to avoid the collective function in the end of the MPI-application is to use an explicit call of `MPI_Abort(Comm)` for each remaining process. But the typical behavior of MPI is to abort the whole MPI-application when the first `MPI_Abort(Comm)` is executed. Also in this case, the application terminates with an error message. Thus, there exist several difficulties why MPI in its standard version is unsuitable to execute fault-tolerant applications.

Fagg and Dongarra [58] present an approach to control the behavior of MPI within an application. The so-called *FT-MPI (Fault-Tolerant MPI)* is an extension of MPI, that supports different failure modes for checkpointing and replication techniques by including semantics into MPI. It is used in combination with a core library called *HARNESS (Heterogeneous Adaptive Reconfigurable Networked SyStem)*, to build necessary services for FT-MPI. The main feature of FT-MPI is to handle failures by modifying the standard communicator of MPI. Different options are given, like shrinking the communicator size and reordering the ranking of MPI-processes or to fill-up the MPI-application with new MPI-processes and to rebuild the communicator.

Bland [28] presents another approach called *ULFM-MPI (User Level Failure Mitigation MPI)*. It is based on Open MPI and also includes several semantics and constructs, to control the behavior of MPI during runtime. In contrast to FT-MPI, an additional runtime system like HARNESS is unnecessary and the constructs allow a more detailed behavior control. ULFM-MPI can be interpreted as a successor variant of FT-MPI, that is built by a working group of the MPI forum with the goal to be included in the MPI standards in the future.

2.5 Energy Efficiency

In computer systems like manycores or grids that are typically used for computational intensive applications, the processor (core) is one of the most power consuming components. Several hardware and software mechanisms have been integrated into the systems to reduce the power consumption and therefore the energy consumption of those components, and finally for the entire system. Today, most processors support different frequency and voltage levels as well as multiple sleep states. Operating systems must provide appropriate software energy management functions, to use such hardware features of a processor. A common standard for these functions is given by the *ACPI (Advanced Configuration and Power Interface)* [82]. In addition, the operating system is typically responsible for adequately utilizing these functions.

Especially in grids and manycores, where the PUs are controlled by separate operating systems, an efficient utilization of the energy management functions is impossible because the operating systems are not aware about the parallel application to be executed. In such a case, the scheduler or the underlying runtime system must support corresponding features. Therefore, numerous scheduling approaches are proposed in the literature, which consider energy efficiency.

Typically, energy efficiency can either be described as using less energy to provide the same amount of service or as using the same amount of energy to provide an increased service [35][89]. Both definitions are correct depending on the perspective. In this thesis, the first definition is considered and corresponding energy-efficient scheduling heuristics and strategies are presented in Chap. 4.

To provide energy-efficient scheduling, i.e. reducing the energy consumption for a task or schedule, a prediction of the energy consumption is necessary. Therefore, an appropriate power model for the system in use must be defined.

In the remainder of this section, firstly energy consumption is defined and power consumption explained. Secondly, typical aspects of modeling the power consumption of a processor (core) are described. Then, energy-efficient scheduling and corresponding approaches are presented. Finally, limitations and issues in real systems are described.

2.5.1 Energy Consumption

The *energy consumption* E for a certain time span $t = [t_1; t_2]$ is the power consumption P integrated over time and is expressed by:

$$E = \int_{t_1}^{t_2} P(t)dt \tag{2.5}$$

If P is constant over time, then this reduces to $E = P \cdot (t_2 - t_1)$. The energy is typically measured in J (*Joule*), which corresponds to Ws (*Watt-second*).

The *power consumption* (measured in W (*Watt*)) of a processor (core) is subdivided into a *static* part P_{static} that is frequency independent and a *dynamic* part $P_{dynamic}$, which depends on both the frequency and voltage. The total power consumption is then the sum of the static and dynamic power consumption:

$$P = P_{static} + P_{dynamic} \tag{2.6}$$

A processor consists of several billion transistors that are combined to gates and circuits to form different function units. A certain number of transistors must be switched in a clock cycle depending on the instruction to be executed and thus the function units in use. Additionally, the *clock cycle time* is dependent on the frequency. As the dynamic part mainly results from switching these transistors and thus from charging and discharging the load capacitance, it is expressed by:

$$P_{dynamic} = ACV^2 f, \tag{2.7}$$

where A is the percentage of active gates, C is the complete load capacitance of the chip, V is the supplied voltage and f is the frequency [67]. The influence of temperature is caused by the dynamic power, i.e. by scaling frequency and voltage.

The static part results from leakage currents, i.e. mainly from *subthreshold leakage* and *gate-oxide leakage* that exist due to shrinking the transistor size and integrating an increasing number of transistors on a single chip. In Fig 2.5 both leakage currents are demonstrated with arrows within a transistor.

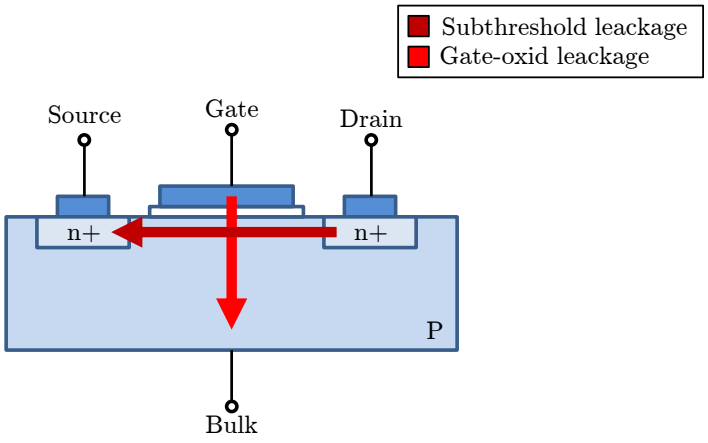

Figure 2.5: Leackage Currents in a Transistor [154].

A transistor has a defined threshold voltage, above which it is turned on. The subthreshold leakage is a current flowing from drain to source when the gate voltage is below the threshold voltage [106][154]. The subthreshold leakage is dependent on the temperature. As the temperature increases with an increased dynamic power, it also influences the subthreshold leakage. In contrast, the gate-oxide leakage results from the decreasing thickness of the isolator between gate and substrate (bulk), where electrons can (with a small probability) tunneling through the isolator [111].

2.5.2 Modeling

To predict the energy consumption for a task and finally for an entire schedule, an appropriate power model of the processor (core) is mandatory or at least a table with real frequency/power consumption pairs. Basically, a *model* is a simplified representation of the reality. The complexity of a model increases significantly with its accuracy. As the power consumption of a processor depends on several factors, like the temperature, instruction mix, usage rate and technology of the processor, there exist numerous approaches in the literature to model the power consumption of a processor with varying complexities and accuracies, like in [23], [45], [71] or [170]. In general, without considering a specific processor, typically a quadratic or cubic frequency function is assumed, as the frequency and voltage are loosely linearly correlated[5]. Therefore, the power consumption for a given frequency f is expressed by:

$$P(f) = c + f^{\alpha} : 2 \leq \alpha \leq 3, \tag{2.8}$$

where c is a constant for the static power consumption and f^{α} is for the dynamic power consumption [7]. For a certain task workload w_i, the runtime r_i of a task at a given frequency f_i results in:

$$r_i = \frac{w_i}{f_i}. \tag{2.9}$$

The energy consumption for an entire schedule (without considering the energy consumption for idle times) is the sum of the energy consumptions for all tasks and is expressed by:

$$E = \sum_i E_i = \sum_i r_i \cdot P(f_i). \tag{2.10}$$

In such a generalized model, usually a continuous frequency scaling is assumed. While a processor typically supports only several discrete frequencies, each modeled frequency f can be "simulated" by a linear interpolation as demonstrated by Eitschberger and Keller [55]. A task i then runs for a fraction β of the runtime r_i with a higher supported frequency f_{high} and for the remaining time $1 - \beta$ with a lower supported frequency f_{low}, so that $f = \beta \cdot f_{high} + (1 - \beta) \cdot f_{low}$. In Fig. 2.6, an example is demonstrated, where a task runs for five time units at a modeled frequency of 1.8 *GHz (GigaHertz)* and the corresponding distribution of the runtime with discrete frequencies 1 GHz and 2 GHz.

[5] For a given voltage there is a maximum frequency and for a desired frequency there is a minimum voltage required.

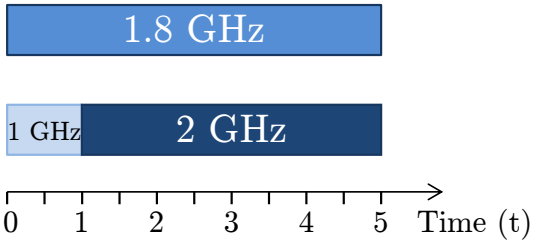

Figure 2.6: Task Runtime with a Modeled Continuous Frequency (Top) and with Discrete Frequencies (Bottom).

The energy consumption for a task i with runtime r_i at a simulated frequency f_i can then be expressed by:

$$E(f_i, r_i) = E(f_{low}, \beta \cdot r_i) + E(f_{high}, (1 - \beta) \cdot r_i). \tag{2.11}$$

The difference in energy consumption for continuous and discrete frequency scaling is very small, so that it often is neglected. In Fig. 2.7, the energy curves for continuous (modeled) and discrete (simulated) frequencies for one second are shown.

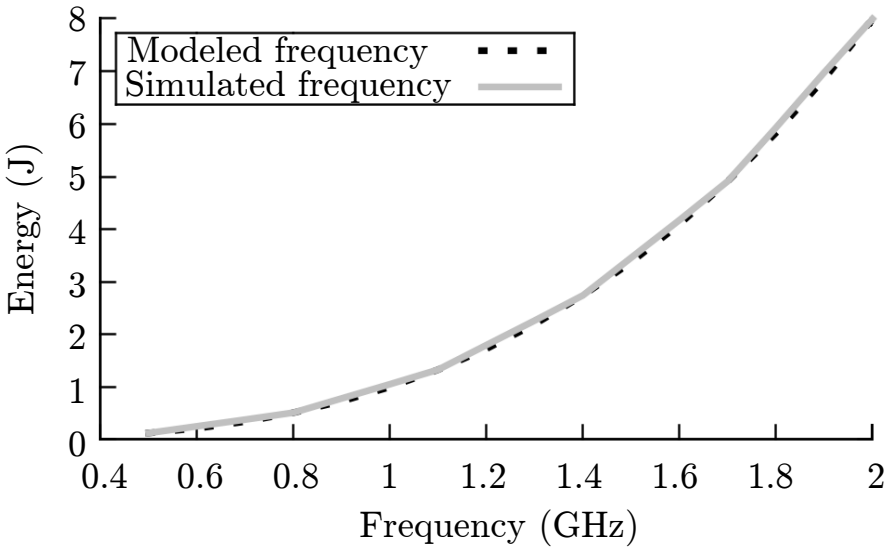

Figure 2.7: Energy Consumption for Continuous and Discrete Frequencies.

In this example a frequency range between 0.5 and 2 GHz is used with discrete frequencies of 0.5, 0.8, 1.1, 1.4, 1.7 and 2.0 GHz. As power model $P(f) = f^3$ is assumed. Between each discrete frequency pair $((f_i, E(f_i)), (f_{i+1}, E(f_{i+1}))$, the energy for simulated frequencies results in a straight line, that connects both frequency

points. Idle times are often neglected, as modern processors typically supports sleep states with a very low power consumption (compared to the power consumption of a task).

2.5.3 Energy-efficient Scheduling

The information about the power consumption of a processor (core) at a certain frequency, either given by a power model or by a table of frequencies with corresponding real power values, can be included into the scheduling process to optimize the energy consumption for a task and finally for an entire schedule. Basically DVFS and DPM are used to construct an energy-efficient schedule. While DVFS is used to scale the frequency and voltage of a processor during runtime, DPM is usually considered for longer idle times to switch a processor into a sleep state. To further reduce the energy consumption, combinations of both techniques are used in scheduling. Next to the use of these techniques also approaches exist that reduce the energy consumption by shutting down processors completely and scheduling the tasks onto the remaining processors. As there exist a huge number of energy-efficient approaches in the literature, only exemplary representatives are presented in the following:

Approaches using DPM Lu et al. [121] present an online low-power task scheduling approach for multiple devices that uses DPM. They focus on clustering idle periods to increase the time for shutdowns and to reduce state-transition delays. They consider the difference in energy for idle periods, when switching to a sleep state mode or when staying in an active mode.

In [122], Ma et al. present an energy-efficient scheduling algorithm for tasks onto a cluster system without DVFS functionality. They propose a combination of task clustering and task duplication to reduce the time and energy for communication and DPM to decrease the static power of processing elements in idle phases. The power consumption is calculated by using technical parameters of the test system.

Baptiste [20] presents a polynomial time algorithm. He focuses on scheduling unit tasks, i.e. the processing time for all tasks remains one time unit, and on minimizing the number of idle periods. He assumes for each task a release date and a deadline.

Rizvandi and Zomaya [153] present a survey of DPM and DVFS scheduling algorithms for cloud systems as well as Jha [97] for uniprocessor and distributed systems.

Approaches using DVFS In [157], Ruan et al. present an energy-efficient scheduling approach called TDVAS[6] for parallel applications on clusters. They assume precedence constraints between the tasks and use DVFS to reduce the voltage and frequency for tasks followed by a gap or directly during idle periods without increasing the makespan. As power model $P = C \cdot V^2 \cdot f$ is assumed. The voltage and frequency levels are given by the system parameters of the used test system.

Lee [112] presents another scheduling approach for multicore processors. He considers multiple periodic real-time tasks and assumes situations with more cores than running tasks. In the proposed scheduling heuristic, tasks are parallelized onto several cores, frequencies are scaled down if possible and unused cores are shut down to minimize the energy consumption. A table of power, frequency and voltage values is given for a test system and the energy consumption per cycle is calculated by the power consumption divided by the frequency.

Huang et al. [86] present an energy-efficient mixed-criticality scheduling approach. They consider DVFS to speedup critical tasks that do not meet their deadlines and to slow down tasks in situations, where tasks finish before the deadlines. They use a cubic power model and assume continuous frequencies.

In [185], Yang et al. present an approximation algorithm for energy-efficient scheduling on a MP-SoC. They assume homogeneous cores that can either be switched off or have to operate at the same voltage level. They consider frame-based tasks that are ready at time 0 and share a common deadline. Their approximation algorithm yield an approximation ratio of 2.371.

For malleable streaming tasks, Kessler et al. [104] present an approach called crown scheduling. They propose a static combined optimization of resource allocation, mapping and DVFS to optimize energy efficiency under throughput constraints with the help of an ILP. They consider dynamic (crown) rescaling when not all tasks are data ready. As power model, a cubic frequency function is used. As extension, Melot et al. [126] present several additional heuristics for crown scheduling.

Mishra et al. [129] present an energy aware scheduling approach for distributed real-time systems. They consider both static and dynamic energy management. They assume tasks with precedence constraints and integrate DVFS in already existing schedules to statically generate energy-efficient schedules. During execution, they further use DVFS dynamically in case of unexpected execution delays.

Further investigations on energy-efficient scheduling using DVFS can be found e.g. in [26], [119] or [189].

[6] A long form of TDVAS is not defined by the authors.

Approaches using DPM and DVFS In [41], Chen et al. present an energy optimization for real-time MP-SoCs with DVFS and DPM. They assume tasks with precedence constraints that share a common deadline. They consider both static and dynamic power consumption and discrete voltage/frequency levels. A mixed ILP is proposed to find energy optimized schedules when using DVFS and DPM.

He and Müller [80] present an online energy-efficient scheduling approach for component oriented systems. They consider hard real-time based tasks and propose a simulated annealing based optimization algorithm. In contrast to many other approaches, they also consider multiple sleep states and switching overhead.

Rong and Pedram [155] present a power-aware scheduling approach for hard real-time systems. They assume periodic tasks with precedence constraints. They present an ILP-formulation and propose also a three-step heuristic to minimize the energy consumption. In a first step the tasks are ordered, then the voltage is assigned, and finally a refinement step follows.

In [18], Bambagini et al. present a survey of energy-aware scheduling approaches for real-time systems. They differentiate between DPM, DFVS, and combined algorithms for uniprocessor and multiprocessor systems and propose a taxonomy of energy-aware scheduling algorithms.

Zhuravlev et al. [194] present a survey of energy-cognizant scheduling techniques. Next to DPM and DVFS algorithms, also thermal aware approaches and scheduling algorithms for heterogeneous systems are considered.

Albers [7] presents energy-efficient algorithms. She considers DVFS and DPM and presents algorithms for various scheduling goals like minimizing the temperature, response time, or fulfill deadline constraints. Additionally, she presents approaches for wireless networks.

Other Approaches For example, Pinel et al. [140] present a two-phase heuristic for mapping independent tasks onto a distributed system to minimize the makespan. They consider energy efficiency by proposing to add low power computing nodes to the distributed system.

In [113], Lenhardt presents approaches to distribute the load in server farms to save energy. Not only an efficient load level in terms of energy savings without significant performance losses is presented, but also fluctuations in the load are considered that can be balanced by switching off some servers or transferring them to a sleep state mode and re-distributing the load to the remaining servers.

Cong and Yuan [47] present an energy-efficient scheduling approach for heterogeneous multicore systems. They map programs onto the most appropriate core based on program phases and use a combination of static analysis and runtime scheduling.

2.5.4 From the Model to the Real World

In real computer systems, several limitations and issues exist in hard- and software that reduce possibilities to utilize energy-efficient scheduling. Continuous frequency scaling for example is impossible, as processors only support several discrete frequencies. The ACPI standard recommends up to 16 frequency levels [82]. The voltage range decreases with a reduced transistor size. Therefore, energy improvements resulting from using DVFS will be getting smaller in the future, as the voltage is the most significant factor that influences the power consumption. Additionally, by shrinking the transistor size and integrating further transistors into a single chip, the heat is getting higher, resulting from switching transistors. This leads to so-called *dark silicon*, i.e. not all parts of a processor can be used at the same time. Therefore, the trend is changing towards processors including different cores, e.g. low power and high performance cores that are switched off, when they are idle [57].

Depending on the timescale that is assumed for a schedule, also the switching times between different frequencies and voltages are becoming important, like for streaming applications as demonstrated by Eitschberger and Keller [55]. Additionally, in some operating systems the accessibility to control DVFS options is restricted. Linux-based systems typically support different *governors* that can be set to change the frequency scaling behavior or to make DVFS accessible to applications that are built on top of the operating system. A corresponding API is the so-called *CPUFreq (Central Processing Unit Frequency)* [88]. In contrast, Windows-based systems do not allow any access to DVFS by other applications or by a user. Also the kind of processor is important for energy-efficient scheduling. For example the voltage for most processors can only be scaled for all cores together within the processor, but not separately, as the voltage regulator for the whole processor is typically placed on the mainboard. One type of processors, where a separate voltage scaling per core is possible, are the Haswell-based server processors from Intel [76]. In this architecture the voltage regulators for each core are directly included on the processor chip.

Modern processors support a so-called *turbo* frequency that is significantly higher than the other frequency levels. The turbo frequency is influenced by several factors, like the number of active cores or the temperature of the processor and therefore it

varies over time. It can only be used for a certain time span because of temperature reasons. Otherwise, *throttling* effects can occur, i.e. switching between a lower frequency and a turbo frequency during execution to reduce the temperature of the processor [82].

2.5.5 Measuring Power Consumption

The classical way to measure the power consumption of a processor is to use an external multimeter. Next to this, several modern processors directly support measurement features that can be used to evaluate the power consumption. For example the *RAPL* (*Running Average Power Limit*) interface in Intels' processors [91] that was introduced in the Sandy-bridge architecture. The RAPL interface is basically used to set and control power limitations of a processor with the help of *performance counters* and *MSRs* (*Model-Specific Registers*). Some of these performance counters provide information about the power and energy consumption of different processor components by using a software power model. In Tab. 2.1 all components are listed that are (partly) supported by modern Intel processors. The MSRs are updated

Table 2.1: RAPL Domains and Corresponding Processor Components.

Domain	**Component**
Package (PKG)	Complete processor chip
PowerPlane0 (PP0)	All processor cores
PowerPlane1 (PP1)	Processor uncore, mainly the GPU
Dynamic RAM (DRAM)	Memory controller

every millisecond and the accuracy of the power and energy information is high in comparison to real measurements, like presented in [76], [77] and [190].

To access the MSRs with the power and energy information within a program, the *PAPI* (*Performance Application Programming Interface*)[7] can be used [178]. As the MSRs are only accessibly in the kernel level, the user must either have root permissions or the executable. In combination with MPI only the latter one is possible, as MPI cannot be used as root-user.

[7]PAPI is a project of the University of Tennessee to design, standardize, and implement an API to access hardware performance counters.

3 Trade-off between Performance, Fault Tolerance and Energy Consumption

In general, a *trade-off* describes a balancing of factors that are unattainable simultaneously. Improving one of these leads to a worsening of the others. The behavior is similar related to a combination of the three optimization criteria: performance, fault tolerance, and energy consumption. A trade-off already exists if two of these three are considered. While there are several approaches in the literature for these two-dimensional optimizations, a combination of all three criteria, especially for duplication based task graph scheduling, has not been considered yet. Sect. 3.1 firstly describes the trade-offs for all two-dimensional combinations. Then, for each combination exemplary related existing approaches are presented. In Sect. 3.2 the indirect trade-off between the fault-free case and the fault case is described. In Sect. 3.3 the three-dimensional optimization problem is explained. Finally, in Sect. 3.4 upper and lower bounds for the three criteria are given.

3.1 Two-dimensional Optimization

A common practice for two-dimensional optimizations with a trade-off between the optimization criteria is to fix one criterion, while the other is optimized. For scheduling in general this means that either a fixed deadline (or makespan) or a fixed energy budget is assumed.

3.1.1 Performance vs. Fault Tolerance

Scheduling goals either focus on minimizing the makespan and thus maximizing the performance of a schedule, or on minimizing the energy consumption for the execution of a schedule (see Sect. 2.3). Here, the first approach is assumed. In

contrast, a main challenge of fault tolerance, next to dealing with a failure, is the minimization of the performance overhead both in a fault-free case and in case of a failure (see Sect. 2.4). As long as the number of available PUs is at least twice as high as the number of required PUs for a performance optimal solution, the whole schedule can be copied and run simultaneously on the remaining half of the PUs. In this case, if only considering the performance and the fault tolerance, a trade-off does not exist and finding an optimal solution is trivial. Fig. 3.1a illustrates an example, where a schedule is copied onto unused PUs without prolonging the makespan. The original tasks are mapped onto PUs 0 and 1, the duplicates are placed onto PUs 2 and 3.

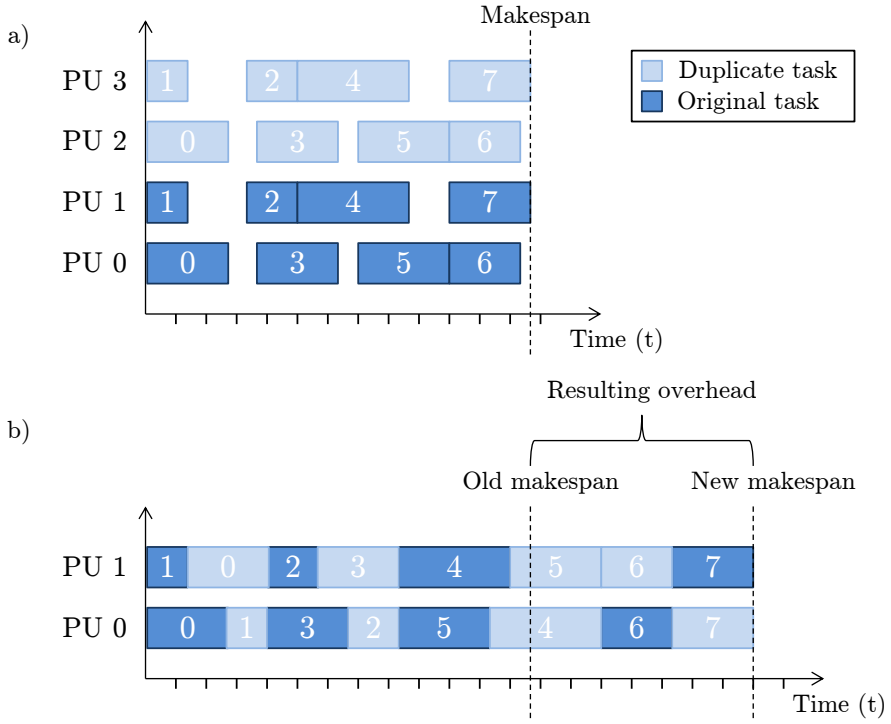

Figure 3.1: Fault-tolerant Schedule: a) Using Free PUs for the Placement of Duplicates, b) No Free PUs are Available.

When the number of available PUs is lower, the problem becomes more complex and a trade-off between performance and fault tolerance must be made. However, the overhead of duplication-based scheduling then is determined by the duplicates that have to be included into the schedule. One reason is the additional execution time of the duplicates. Secondly, the dependencies between all tasks (original tasks and duplicates) has to be considered because each task requires the results of its predecessors, independently of whether the tasks are original tasks or duplicates.

Fig. 3.1b illustrates the fault-tolerant schedule when free PUs are unavailable. Now, the makespan is increased by placing the duplicates directly between the original tasks resulting in a performance overhead. Usually, the focus lies on the fault-free case, because in most environments a failure occurs infrequently. Therefore, minimizing the performance overhead in this case is desired.

Fechner et al. [59] present a fault-tolerant duplication-based scheduling approach for grids that guarantees no overhead in a fault-free case. They assume to have an already existing schedule (and task graph) and extend this schedule by including a duplicate for each original task. They focus on homogeneous PUs and assume a fail-stop model. To reduce the performance overhead of the schedule, an original task sends a commit message to its corresponding duplicate after it has finished so that the duplicate can be aborted.

Hashimoto et al. [79] present a fault-tolerant duplication-based scheduling algorithm for multiprocessor systems. They assume to have homogeneous processing elements and a fail-stop model. In their work, a single processor failure is considered. The algorithm consists of two phases, one for the partitioning of the parallel program into subsets of tasks, the other for duplicating and scheduling the tasks in the subset. The duplication is done in two steps: In the first step, the tasks are duplicated to minimize the communication delays and thus to increase the performance. To reach the fault tolerance, a second step is used to duplicate tasks that are not duplicated in the first step.

In [83], Hongxia and Xin present a fault-tolerant scheduling algorithm with dynamic replication, called *DRFT*-algorithm *(Dynamic Replication of Fault-Tolerant scheduling)*. They combine an active and passive mode of backup tasks to reduce the performance overhead in case of a failure. For each task a backup level is calculated to decide how long the active and passive backups should be.

Jayadivya et al. [95] present a fault-tolerant workflow scheduling approach for Cloud Computing, called *FTWS (Fault-Tolerant Workflow Scheduling)*. They assume to have a predefined deadline, within which the schedule must be completed. To get a good performance while tolerating failures, a combination of replication and resubmission of tasks is used. Therefore, a heuristic metric is calculated to find the trade-off between replication and resubmission.

Izosimov et al. [93] present an algorithm that handles the transparency/performance trade-offs of fault-tolerant embedded systems. They focus on transient faults in safety-critical systems. The transparency requirements are handled at the application level and give the designer the opportunity to trade-off between memory

requirements and debuggability and performance. It reduces the overhead due to fault tolerance.

In [10], Anglano and Canonico extend a knowledge-free scheduler called *WQR (WorkQueue with Replication)* scheduler that does not base its decisions on the status of resources or characteristics of applications. They focus on bag-of-tasks grid applications and try to maximize the performance by using task replication. They consider heterogeneous systems and let the replicated tasks run on processing units, whenever they are in idle mode. In order to achieve fault-tolerance and good application performance, they use automatic restarting of tasks and checkpointing in their fault-tolerant extension of WQR, so-called WQR-*FT (Fault-Tolerant)*.

Further investigations on performance and fault-tolerance in scheduling can be found e.g. in [3], [24] and [169].

3.1.2 Performance vs. Energy Consumption

The performance of a schedule is dependent on the mapping of the tasks. The more an application can be parallelized the better is the performance. Additionally modern processors support several frequencies at which a processor can run. Thus, the tasks should be accelerated as much as possible, i.e. use the highest supported frequency of a processor to achieve the highest performance. As described in Sect. 2.5, the dynamic power consumption of a processor (core) is usually modeled by a quadratic or cubic frequency function. From the perspective of energy consumption a low frequency typically leads to better results than a high frequency. However, this is highly dependent on the processor type, because in real systems there are several other factors that influence the power consumption of a processor and thus the energy consumption of a schedule (e.g. the static power or the idle power etc.) In Fig. 3.2 an example schedule is given (a) and the resulting performance overhead is shown when the frequency is scaled down to save energy (b).

Pruhs et al. present in [147] an approximation-algorithm to optimize the performance of a schedule by scaling the processor frequencies, when a certain energy budget is given. They consider continuous frequencies and tasks with precedence constraints. As power model a cubic function f^3 is used. They show that with these assumptions the aggregate powers of the processors are constant, i.e. the sum of the processors' power consumptions is constant over time in any locally optimal schedule. However, to find an optimized solution the problem can be restricted to schedules with those properties (so called constant power schedules).

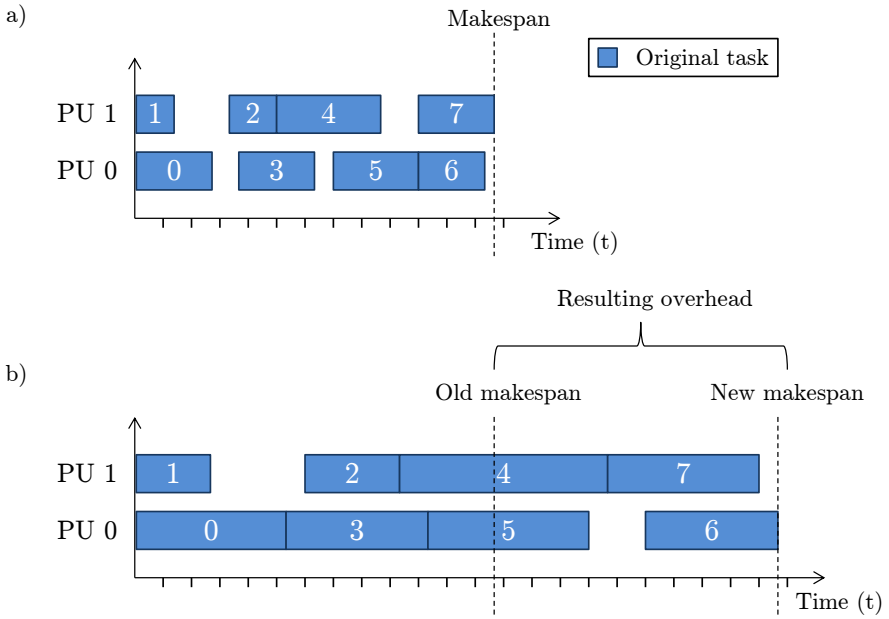

Figure 3.2: Example Schedule: a) Running at a High Frequency (e.g. 2 GHz), b) Reducing Energy Consumption by Scaling Down the Frequencies (e.g. to 1 GHz).

In [105], Kianzad et al. present a framework, called *CASPER (Combined Assignment, Scheduling, and PowER management)*, that uses a genetic algorithm to find an optimized schedule. They consider both homogeneous and heterogeneous systems and use frequency scaling. The focus in their presented work lies on the integration of the scheduling and scaling steps.

Singh and Auluck [162] present an integrated DVFS and duplication-based approach for optimizing power and performance in heterogeneous multiprocessors. They use a combination of DVFS i.e. let the tasks run on low voltages and frequencies and duplicating tasks on multiple processors to reduce the communication delay between the tasks. A Mixed Integer Programming (MIP) formulation is given to get a better power consumption and performance at the same time. They show that their approach leads to significant improvements for both the processor power and the total power, i.e. the processor power and the communication.

Ma et al. [123] present an algorithm, called *EOTD (Energy Optimization scheduling for Task Dependent graph)* that uses task clustering to reduce both communication time and energy consumption. DVS is considered to decrease dynamic power of processing elements without violating a user predefined deadline. Also static power is reduced by dynamic power management and binary search techniques.

In [180], Wang et al. discuss the relationship between energy consumption and performance in clusters with DVFS. They propose scheduling heuristics to minimize the energy consumption of a schedule without prolonging the makespan. They reduce the energy consumption by using the slack time for non-critical tasks, to slowdown the tasks by scaling down the frequencies. Additionally, they consider energy-performance trade-off scheduling by increasing the makespan of a schedule within an affordable limit to reduce the energy consumption.

Li [115] presents a performance analysis of power-aware task scheduling algorithms and shows that on a multiprocessor computer, the problem of minimizing the schedule length with energy constraint and the vice versa problem, minimizing energy consumption with schedule length constraint are equivalent to the sum of powers problem. He analyses the performance of list scheduling and equal-speed algorithms by comparing the performance of the algorithms with optimal solutions analytically.

Aupy et al. [12] present a survey of different energy-efficient approaches. They describe models for data centers and energy models next to a case study about task graph scheduling and replica placements.

Further investigations on performance and energy consumption in scheduling can be found e.g. in [99], [128] and [159].

3.1.3 Fault Tolerance vs. Energy Consumption

In any case where fault tolerance is warranted, some kind of redundancy is necessary (see Sect. 2.4). In duplication-based scheduling the redundancy is used in the form of task copies, so-called *Ds (Duplicates)*. A D can only be started, when it has received all results from its predecessors and when no other task is running on the corresponding PU. We assume that a D can be aborted when the corresponding original task has finished correctly. To save energy, a D can also be placed with runtime (and thus energy) zero at the end of the corresponding original task, because it never has to be started in the fault-free case, i.e. when the completion of the original task is successful. Such a D with runtime zero is then called *DD (Dummy Duplicate)*. To get a better fault tolerance (in terms of performance overhead in case of a failure) the duplicates should be executed simultaneously to the original tasks or at least as long as possible. However, increasing the execution time of duplicates leads to an increased energy consumption. Fig. 3.3 shows this behavior with an example schedule.

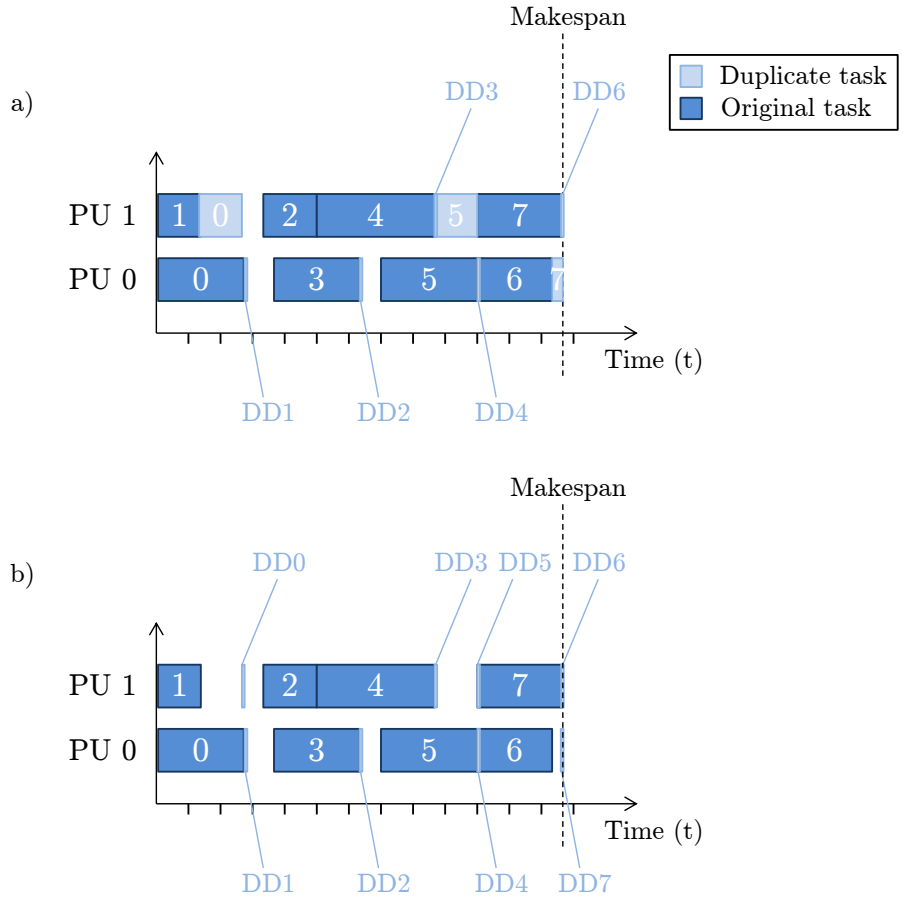

Figure 3.3: Fault-tolerant Schedule: a) Using Dummies and Duplicates to Increase the Fault Tolerance, b) Only Using Dummies to Get a Better Energy Consumption.

In the upper case, duplicates are executed as long as possible without prolonging the makespan. In the bottom case, only DDs are used. Hence, in the fault-free case no energy has to be consumed for the DDs.

In terms of energy consumption the tasks should run with a low frequency to save energy. But slowing down the tasks without changing the mapping of the tasks results in different gaps and thus to another placement of the duplicates that might result in a higher performance overhead in case of a failure. Additionally, also the duplicates have to be slowed down because otherwise a duplicate would finish earlier than the corresponding original task. Thus, the duplicates are prolonged and gaps might become too small for the placement of whole duplicates. In this case, either the succeeding tasks have to be shifted and the makespan is significantly increased or the duplicates can only be executed for a short time before the next original task

starts. For this reason, the remaining part of the duplicates, that has to be executed in case of a failure, grows and leads to a higher performance overhead. In Fig. 3.4 an example schedule is illustrated (a) and the resulting schedule, when the frequency is scaled down to save energy (b).

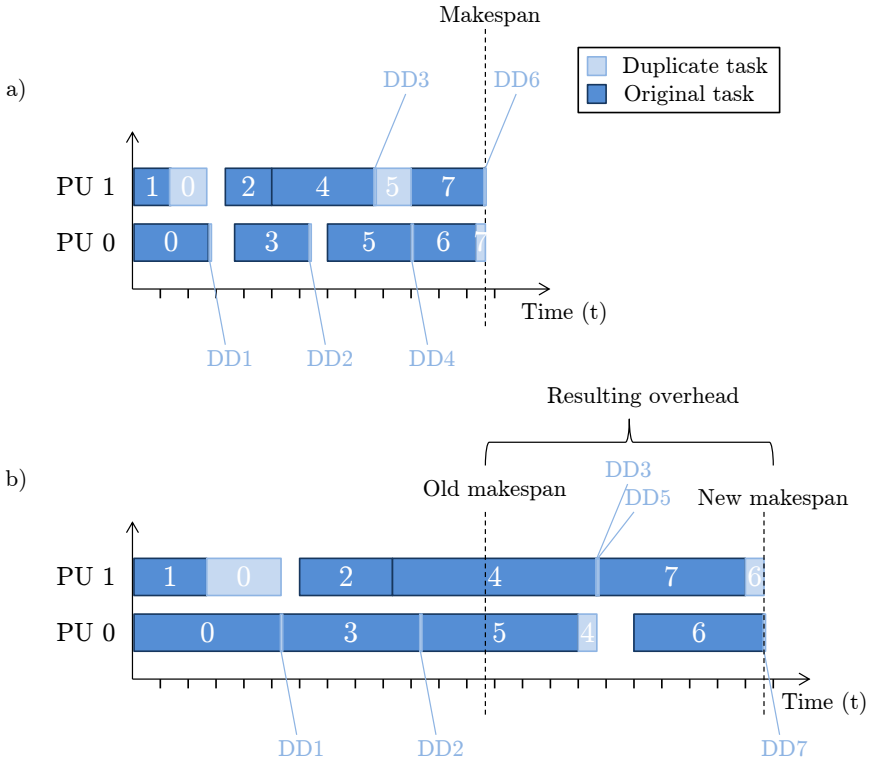

Figure 3.4: Placement of Duplicates: a) Running at a High Frequency (e.g. 2 GHz), b) Reducing Energy Consumption by Scaling Down the Frequencies (e.g. to 1 GHz).

In the upper schedule, the duplicates of tasks 0, 5, and 7 can be executed for a short time. In the bottom case, when using a lower frequency, the duplicates of tasks 0, 4, and 6 can be executed partly.

Zhao et al. [193] present a fault-tolerant scheduling algorithm, called MaxRe, with a dynamic number of replicas for heterogeneous systems. They focus on cloud environments and consider crash faults. Typically n replicas are needed to tolerate n failures. In their work, they show that a higher number of replicas does not always lead to a higher reliability. The longer a resource is used and the higher the usage rate of a resource is, the higher is the probability of a failure. Thus, filling up a schedule with replicas might lead to more failures. They try to minimize the number of replicas while satisfying the user's reliability requirements.

Aupy et al. [12] also present some works about fault tolerance and energy efficiency, but with checkpointing instead of task duplication.

3.2 Fault-free Case vs. Fault Case

Yet another trade-off exists between the fault-free case and the fault case. Most improvements in the fault-free case lead to some worsening in the fault case and vice versa. For example focusing on the performance in the fault-free case (not considering the marginal case in Sect. 3.1.1) means using placeholders (so called Dummies) as duplicates where necessary and consider the communication times between all duplicates and original tasks only in case of a failure. Thus, the worsening of the performance is shifted from the fault-free case into the fault case. When focusing on the energy consumption in the fault-free case, the tasks (and thus the duplicates) are prolonged because the frequencies of the PUs are scaled down. In case of a failure, these duplicates cause a higher performance overhead, because they still run with a low frequency. Focusing on the fault tolerance in the fault-free case improves on the one hand the performance overhead in case of a failure, but on the other hand increases the energy consumption in the fault-free case. However, either this trade-off must be considered directly when focusing on the fault-free case, or there have to be other options and strategies for the fault-free case and in case of a failure, so that the preferences of a user can be set independently in both cases. As the approaches in the literature focus only on one of both cases, there does not exist any work about a combined approach that considers directly this trade-off.

3.3 Three-dimensional Optimization: Performance vs. Fault Tolerance vs. Energy Consumption

As seen in Sect. 3.1, a trade-off already exists in all two-dimensional cases. A combination of all three criteria, performance, fault tolerance and energy consumption leads to several additional optimization parameters. While one criterion is improved, either one or both of the others are worsened. But it is unclear how this worsening should be distributed to the other criteria. The main focus varies in different situations. For example, in a time critical environment, the performance is the most important criterion next to the fault tolerance. Thus, in this situation the performance and the fault tolerance is usually favored over minimizing the energy

consumption. Another situation is that a failure occurs extremely rarely and thus the energy consumption in a fault-free case becomes more important. Another example exists in mobile devices, such as smartphones and laptops, where the energy consumption is the most important criterion next to the performance. Thus, the main focus is put on the energy consumption and on the performance while fault tolerance is neglected.

Especially for real-time systems, approaches are given in the literature, that combine all three criteria. The real-time constraint itself can be interpreted as the performance criterion, because in such systems the execution of a task or schedule has to be finished before a deadline. Thus, the performance is fixed while the other criteria are improved. In real-time systems, the focus typically lies on transient faults, where checkpointing or backup mechanisms are used to circumvent a fault.

For example, Cai et al. [37] present a greedy heuristic to reduce the energy consumption in fault-tolerant distributed embedded systems with time-constraints. The fault tolerance is given by a combination of re-executing tasks and using replication. They assume a heterogeneous system environment where the processors are connected via a bus. They focus on transient faults and they use a reserved slack for the re-execution. Their greedy heuristic searches for the task that consumes the most energy and try to either change the mapping of the task or to include (or delete) a replica. This procedure is repeated until all tasks are considered or until no further energy improvement can be achieved. Thus, in their work the makespan of a schedule and the fault tolerance is fixed while the energy consumption is improved.

In [6], Alam and Kumar present another approach. They assume that only one specific transient fault could occur during the execution of a task. Checkpointing is used to tolerate faults and a fault is detected within the next checkpointing interval. Then, the last checkpoint is used to execute the remaining part of the task. The minimization of the energy consumption is done by DVS, i.e. using a discrete voltage level with its corresponding clock speed. To achieve a good fault tolerance, they find the optimal number of checkpoints to be included into the tasks. In their work, the performance is fixed, while the fault tolerance and energy consumption is being optimized.

Tosun et al. [175] present a framework that is used to map a given real-time embedded application under different optimization criteria onto a heterogeneous chip multiprocessor architecture. They consider energy consumption, performance, and fallibility (the opposite of reliability) as objective functions or constraints for their ILP formulation. Duplicating tasks is used to detect faults and DVS for energy

savings. The processors are connected via a shared bus system and communication overheads are neglected. A result checker after each task and its corresponding duplicate is used to find a fault. They focus on the trade-off between increasing the number of duplicates, energy consumption, and performance.

However, the alignment of the optimization is very situational and ultimately depends highly on the user preferences.

3.4 Estimation of Upper/Lower Bounds

3.4.1 Performance

Focusing on performance PE, the best solution is to parallelize an application as much as possible. Furthermore, the highest available frequency $f_{highest}$ should be selected, if the system in use supports different frequencies. Then, a simple general upper bound for the performance can be described as the following equation, where m_{seq} is the makespan in cycles, when all tasks are running in sequence and $p_{max} \in PU$ is the maximum number of PUs used:

$$PE_{best} = \frac{m_{seq}}{p_{max} \cdot f_{highest}} \tag{3.1}$$

A lower bound for the performance can be achieved by running all tasks in sequence on one PU with the lowest possible frequency f_{lowest}. Equation 3.2 shows this context:

$$PE_{worst} = \frac{m_{seq}}{1 \cdot f_{lowest}} \tag{3.2}$$

3.4.2 Fault Tolerance

While a schedule is either fault-tolerant or not, in this thesis, the fault tolerance FT is rated by the performance overhead in case of a failure. Therefore, when focusing on the fault tolerance the best solution is to copy the whole schedule and execute it simultaneously (completely independent) to the original one on other PUs. Then, both the performance, i.e. the makespan m_{ft} in case of a failure and in a fault-free case m are equal. Accordingly, the performance overhead results to zero percent.

$$FT_{best} = \frac{m_{ft} - m}{m} \cdot 100 = 0\% \tag{3.3}$$

However, the worst solution is when the schedule is not fault-tolerant and a failure occurs directly before the end of the schedule execution. Then, the whole schedule has to be repeated on $p-1$ PUs and the makespan $m_{ft} = 2 \cdot m$ in case of a failure is at least doubled in comparison to the fault-free case m. Thus, the performance overhead in case of a failure results in 100 percent.

$$FT_{worst} \geq \frac{m_{ft} - m}{m} \cdot 100 = \frac{2 \cdot m - m}{m} \cdot 100 = 100\% \tag{3.4}$$

3.4.3 Energy Consumption

The energy consumption for a schedule execution is highly dependent on the power consumption of the processors in use. As the power consumption of different processors varies significantly, a general upper or lower bound cannot be determined. Only when a specific processor or power model is considered, the boundaries can be defined. However, instead of finding general boundaries, some factors are given that influence the energy consumption and thus the upper and lower bounds. Then, exemplary the lower and upper bounds for a simple cubic power model are described.

The power consumption of a processor (core) can be subdivided into a static part that is frequency independent and a dynamic part that depends on both the frequency and the supplied voltage (see Sect. 2.5.2). When the static power consumption is high in comparison to the dynamic power consumption, using a certain frequency influences the total power consumption a little. Thus, speeding up a task by running at a high frequency is more energy-efficient than running at a low frequency. In contrast, when the static power consumption is low, it is better to scale down the frequency and thus slowing down a task to get a low energy consumption.

The idle power consumption influences the lower and upper bounds significantly. When the idle power is high, a reduction of the idle times within a schedule would lead to better results than having a lot of gaps between the tasks. Thus, trying to fill up the gaps within a schedule with duplicates and shutting off all unused processors would be better than using a free processor for the duplicates. If the idle power can be neglected, because it is very low or cores can go to sleep mode, both solutions would lead to the same energy behavior.

Another reason results from different instructions. For example, a task with many load and store instructions leads to a different energy consumption than a task with many arithmetic calculations and only a few load and store instructions.

It is important whether only the power consumption of a processor core is considered or the power consumption of the whole processor. If the focus lies on the processor core, the static power consumption is very low, because there are only a few components within the core that influence the power consumption. From the perspective of a processor in total, there are many components like the memory controller, the low-level caches, graphic units etc. that also influence the power consumption. Additionally, some components might only be switched on when at least one processor core is under load. So that the relation between running in idle mode and executing something on one core might be more complex than focusing only on a processor core and its components.

Starting from a simple cubic power model for the dynamic power consumption, i.e. at frequency f, a core consumes power proportional to f^3. The energy consumption of a task t_i is then the product of power and task runtime which is inversely proportional to the frequency. For simplicity, constant factors and low-order terms are left out. Then an upper and lower bound for the energy consumption can be given by:

$$E_{best} = \sum_{i=0}^{n-1} \frac{t_i}{f_{lowest}} \cdot f_{lowest}^3, \tag{3.5}$$

$$E_{worst} = \sum_{i=0}^{n-1} \frac{t_i}{f_{highest}} \cdot f_{highest}^3. \tag{3.6}$$

In this model, the idle power and power for communication is neglected and thus only the sum of all tasks has to be considered but not the concrete mapping or scheduling or the resulting makespan.

4 Fault-tolerant and Energy-efficient Scheduling

In this chapter, various approaches that include fault-tolerant and energy-efficient aspects into the scheduling process are presented. Starting with the assumptions in Sect. 4.1, fault-tolerant scheduling heuristics are given in Sect. 4.2. Next to the previous work, new fault-tolerant heuristics are introduced. In Sect. 4.3 several extensions, i.e. heuristics and options are presented that combine energy savings and investments with the fault-tolerant approaches for both the fault-free case and in case of a failure. As these extensions affect different parts of the heuristics, possible combinations of those are discussed in Sect. 4.4 and new combined strategies are proposed that reflect different preferences of a user. Finally, energy-optimal solutions are presented in Sect. 4.5.

4.1 Assumptions

In this work, homogeneous systems with a fully meshed network are considered. Starting from an already existing schedule (and task graph), the following scheduling algorithms are used afterwards to include fault tolerance and energy efficiency aspects. Thus, these algorithms are not restricted to specific schedulers and therefore can be combined with every static scheduling method. To be independent from a concrete system, one assumption and also goal is to support the scheduling and execution without any changes of the operating system. Hence, a corresponding scheduler and runtime system is built on top of the operating system.

As failure model, the crash model is assumed (see Sect. 2.4) and a single permanent fault can occur during the runtime of a schedule. For simplification, possible bottlenecks e.g. in memory or network bandwidth are not considered. The probability of a fault is neglected in the following presentation.

A maximum load level of a PU is considered by assuming computational tasks with a standard instruction mix. Influences in the power consumption due to the

temperature are ignored because the temperature is assumed to be kept constant by active cooling. Also influences due to the voltage are neglected as the voltage is always scaled with the frequency to the lowest possible level.

A task can only run with one frequency, i.e. a frequency change during the runtime of a task is prohibited. Switching times between different frequencies and the corresponding energy consumption are neglected as the time and energy for executing tasks is much higher in comparison.

4.2 Fault-tolerant Scheduling Heuristics

4.2.1 Previous Work

In the following, the static task duplication-based approach of Fechner et al. [59] (see Sect. 3.1.1) is described in more detail, as it is used, extended and combined with energy-efficient options and heuristics. The main goal of Fechner et al. is to guarantee no overhead in a fault-free case and only a small overhead in case of a failure. Their assumptions are included in the assumptions given above (see Sect. 4.1), except the failure model, where they assume a fail-stop model. As in the implemented runtime system, described in Chap. 5, all PUs get the information about a failure shortly after its occurrence, the behavior is nearly equal to the fail-stop assumption. Therefore, their approach can also be used with the assumptions described above.

To be fault-tolerant, each original task within a schedule is copied and its duplicate (D) is placed onto another PU. In case of a failure, the execution of a schedule can be continued by running the duplicates instead of the faulty original tasks. If an original task has finished it sends a commit message to its corresponding duplicate, so that it can be aborted. Fig. 4.1 illustrates this context.

Typically, in a schedule several gaps exist because of dependencies between tasks, i.e. a task must send its results to successor tasks that use them as input to start their execution. A duplicate can be placed either in those gaps or directly between two succeeding tasks. To avoid an overhead in the fault-free case, in all situations where a duplicate would lead to a shift of all its successor and succeeding tasks[1] only a placeholder, a so called *dummy duplicate* (DD) is placed. A DD cannot be

[1] To differ between successor tasks because of dependencies and successor tasks because they are placed in sequence on a common PU, in this work the term "successor" is used for the former kind, the term "succeeding" for the latter kind.

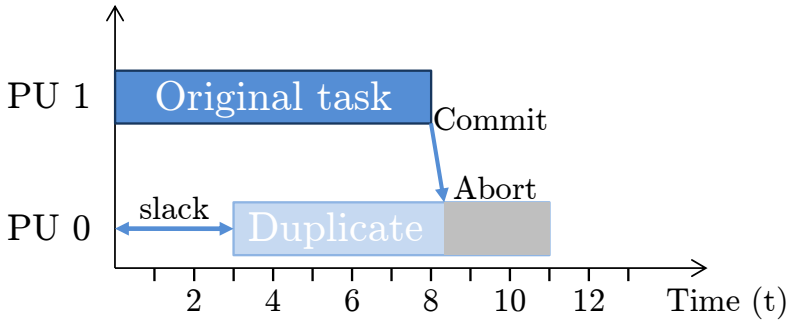

Figure 4.1: Abort Duplicate After Finishing Original Task [59].

placed earlier than the end of its corresponding original task, as the information about the correct execution is then confirmed by the commit message so that a DD never has to start in the fault-free case. The DD is only extended to a fully duplicate in case of a failure, when the commit message is not received. The communication overhead is increased for duplicates that start and finish at the same time as their corresponding original tasks, because both send its results to the same successor tasks. To reduce the communication overhead, duplicates are placed with a short delay, so called *slack* (see Fig. 4.1). Thus, either the results of an original task are sent to its successor tasks (in a fault-free case) or the results of the corresponding duplicate (in case of a failure), but not both.

Fig. 4.2a illustrates an example task graph. For a better understanding the communication times and the slack are omitted. Figs. 4.2b and 4.2c show the resulting schedules of two strategies.

The first strategy uses only DDs and takes not much advantage of gaps. As the DDs are placed with runtime zero, the performance overhead in a fault-free case can be guaranteed to be zero. In case of a failure, the DDs of the corresponding faulty original tasks are converted to Ds and executed. Therefore, a moderate overhead arises in the fault case. In the second strategy, all DDs within a gap are checked whether they can be directly converted to Ds. If a DD is placed at the end of its corresponding original task and if there exists an idle time before the DD, it can be converted to a D. The size of the D is then bounded by the idle time before the D. In this strategy, gaps are used more efficiently and no performance overhead exists in the fault-free case. In case of a failure, Ds only have to be extended for a short fraction so that the performance overhead can be decreased.

In [53], Eitschberger and Keller show that the consideration of communication times is important for the placement of Ds and DDs. If two tasks are placed on

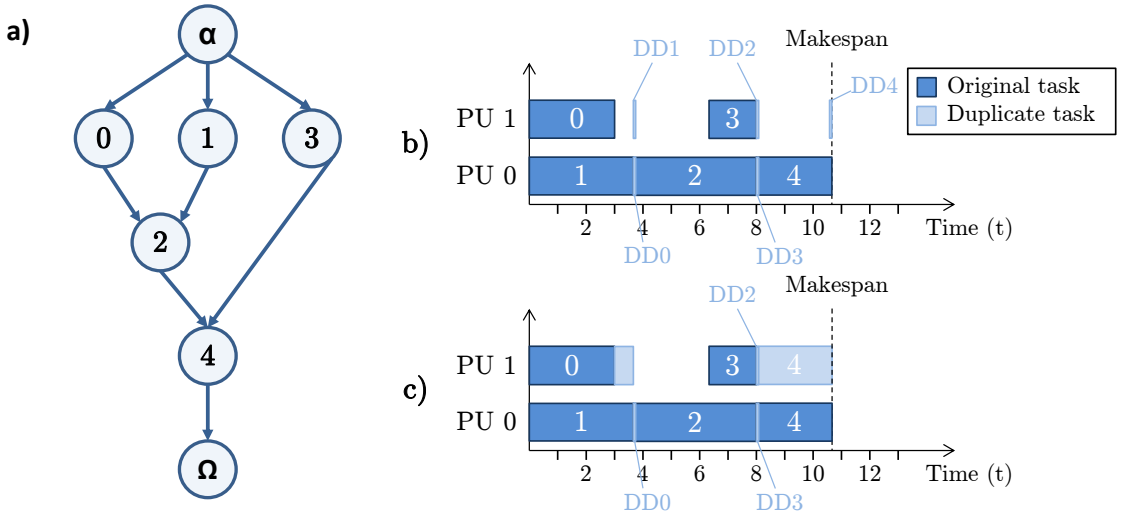

Figure 4.2: a) Simplified Taskgraph, b) Strategy 1: Use Only DDs, c) Strategy 2: Use Ds and DDs [59].

the same PU, the communication time can be neglected and the above explained strategies can be used without any changes. But if they are placed onto different PUs, the communication time has to be considered. Therefore, Ds often must be started delayed, depending on the schedule structure. The influence of communication times is illustrated in Fig. 4.3. We assume that task 0 requires five time units

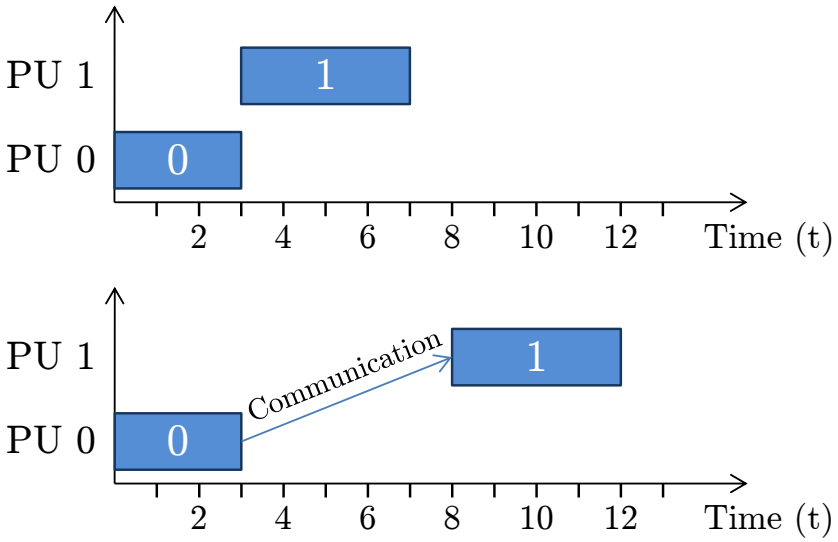

Figure 4.3: Influence of Communication Times [53].

to transfer its results to task 1. In the upper case, the placement without any communication time is presented. Task 1 can directly be started at the end of task 0.

In the bottom case, the resulting placement by considering the communication time is shown. Hence, task 1 cannot be started anymore at the end of task 0, because it has to wait for the results of task 0. Thus, it can only be started after five time units. As duplicates and original tasks are placed onto different PUs, at least one of them is on another PU than its predecessor.

The consequences are demonstrated exemplary for the first strategy by extending the initial task graph and schedule example. We assume there is a further task (task 5), that is a successor of task 2 and a communication time ct between task 1 and task 2 that is longer than the execution time of task 2, e.g. $ct = t2+x$. Fig. 4.4a shows the resulting task graph and the schedule without considering the communication time (Fig. 4.4b) and with the communication time included (Fig. 4.4c).

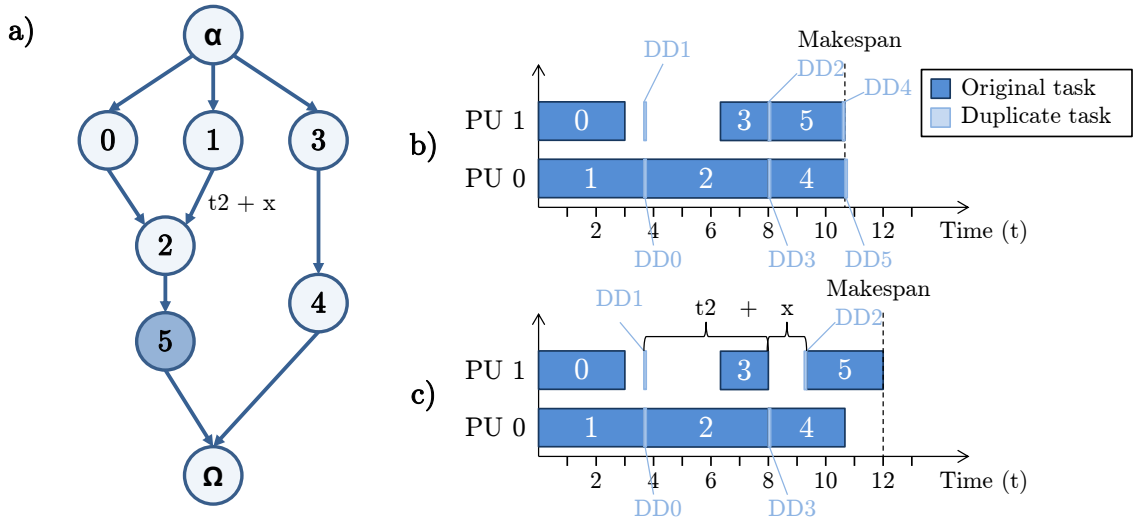

Figure 4.4: a) Extended Task Graph, b) Schedule without and c) with Communication Time [53].

The makespan in the upper case is not changed and no overhead exists. In the bottom case, when considering at least one communication time, the makespan is increased and an overhead directly arises. Thus, the above explained strategies cannot be used in its current version to guarantee no overhead in a fault-free case, when communication times are included.

Eitschberger and Keller present an adapted version [53] that guarantees no overhead in a fault-free case, even when considering communication time. They propose to handle the fault-free and fault case separately. In the fault-free case, some communication times can be neglected, as some are only needed in case of a failure. Therefore, communication times between DDs and original tasks (and only in this

order) are not considered in the fault-free case. With this extension, no overhead in a fault-free case can be guaranteed. Only in case of a failure, these communication times have to be considered, because then some of the DDs are converted to Ds and executed. Hence, their results have to be sent to the successor tasks, which leads to a few more shifts of tasks in the fault case.

However, in the fault-free case DDs sometimes have to be placed during the runtime of original tasks on the same PU. Thus, there exist some overlapping of tasks. But as the information about a failure is known prior to the DD execution, these overlapping can be solved in the fault case. To support this behavior, each PU needs the information about the complete task graph. A memory buffer is required to hold the results of tasks that have finished, so that the transmission to successor tasks on other PUs is continued in case of a failure. Fig. 4.5 illustrates the placement of DDs for the old and for the new version.

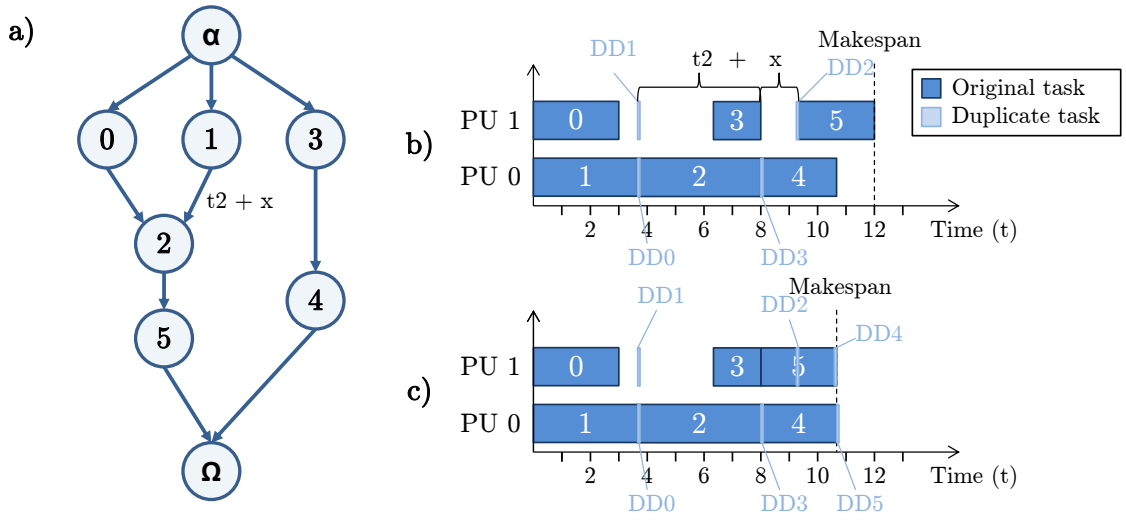

Figure 4.5: a) Example Taskgraph, b) DD Placement Old Version, c) DD Placement New Version [53].

In Fig. 4.5b the old schedule with an overhead in a fault-free case is shown. In Fig. 4.5c the new version is depicted. Here, an overlapping of DD2 and task 5 exists and the makespan remains constant.

An implementation of a corresponding scheduler that supports the above explained duplicate placement strategies is presented by Eitschberger and Keller [53]. The scheduler consists of two parts: One part for the schedule generation, i.e. the placement of duplicates, the other for simulating the execution of a schedule in the fault-free and fault case to predict the overhead. The scheduler has various options

that can be set like using different strategies, setting the time for the slack and considering free processors for the placement of duplicates that are available but unused. The simulator firstly predicts the performance overhead in the fault-free case. It secondly simulates a failure for each task (at the end of a task) to calculate the overhead in a fault case. A detailed description can be found in [44] and [53]. This scheduler (and simulator) is used and extended in this work by the heuristics and options explained in the following subsections and in Sect. 4.3.

The overhead of the scheduling strategies is evaluated with a synthetic benchmark suite of Hönig [84]. In total nearly 34,000 (performance optimal) schedules and task graphs are considered with different properties like the number of tasks, the number of PUs, or the edge density. As this benchmark suite is used in this work, a detailed description is given in Sect. 6.1.1. The overhead in the fault-free case of both versions, i.e. the version of Fechner et al. and the extended version of Eitschberger and Keller is presented in Tab.4.1.

Table 4.1: Overhead in the Fault-free Case [53].

	Version Fechner et al.	Version Eitschberger and Keller
Strategy 1	2.52%	0.00%
Strategy 2	2.52%	0.00%

In the previous version (left column) a relative overhead of around 2.5% on average arises for both strategies. In contrast, the new version (right column) guarantees no overhead in the fault-free case. As the second strategy is based on the first one, the general placement of DDs is the same. Only if possible, DDs are converted to Ds. Therefore, the overhead of both strategies is equal in the fault-free case. In Tab. 4.2 the minimum, average (averaged over all failure points) and maximum overhead for both strategies and versions in the fault case are shown.

Table 4.2: Overhead in the Fault Case [53].

	Minimum	Average	Maximum
Strategy 1 (Fechner et al.)	7.43%	24.23%	39.61%
Strategy 2 (Fechner et al.)	6.41%	22.08%	37.18%
Strategy 1 (Eitschberger and Keller)	5.28%	22.89%	39.05%
Strategy 2 (Eitschberger and Keller)	4.19%	20.87%	36.83%

The improvements of using Ds (if possible) directly in the fault-free case for the second strategy are for all versions around 2%. The improved version of the strategies leads to better results in case of a failure compared to the results of the previous

version. While for the previous version the overhead is on average 24% for strategy 1 and 22% for strategy 2, the overhead of the improved version is around 1.3% lower. The maximum overhead of both versions is for all strategies nearly the same, but the minimum overhead is up to 2.2% lower in the improved version. The large difference in general between the minimum and maximum overhead results from the time at which a failure occurs. Is a failure directly in the beginning of a schedule, several duplicates have to be extended, tasks have to be shifted and a high overhead arises. In contrast, when a failure occurs shortly before the end of a schedule, only few duplicates must be extended and fewer shifts must be performed. Therefore, the overhead in the latter situation is small.

In this work, only the new version proposed by Eitschberger and Keller is used for the placement of Ds and DDs (Strategy 1 and 2). Therefore, no difference is made in the following between the versions.

4.2.2 Use Half PUs for Originals (UHPO)

A simple way to achieve a high FT is to use half of the available PUs for original tasks and the other half for placing duplicates. In this case, the already existing schedule is not used anymore. Instead, a simple list scheduler generates a new schedule with the reduced number of PUs. A copy of the schedule can then be placed onto the remaining PUs. Dependencies between original tasks and duplicates are not considered, because both schedules are executed synchronously. Therefore, the performance of a schedule is not influenced in case of a failure. In Fig. 4.6 an example schedule (upper case)[2] and the resulting schedule by using this heuristic (bottom case) is illustrated.

In this example, all tasks are placed in sequence on one PU and the transfer times between the tasks do not have to be considered anymore (bottom case). Thus, no parallelism is used anymore within the original schedule, because only two PUs are available. Instead, a copy of the sequential solution is placed on the other PU. This heuristic can only be used in the fault-free case, because only one failure is considered and thus no more duplicates have to be included in case of a failure. In Lst. 5.1 a simplified code for the list scheduler is shown.

[2]For simplification, all following examples are related to this example schedule, if not declared otherwise.

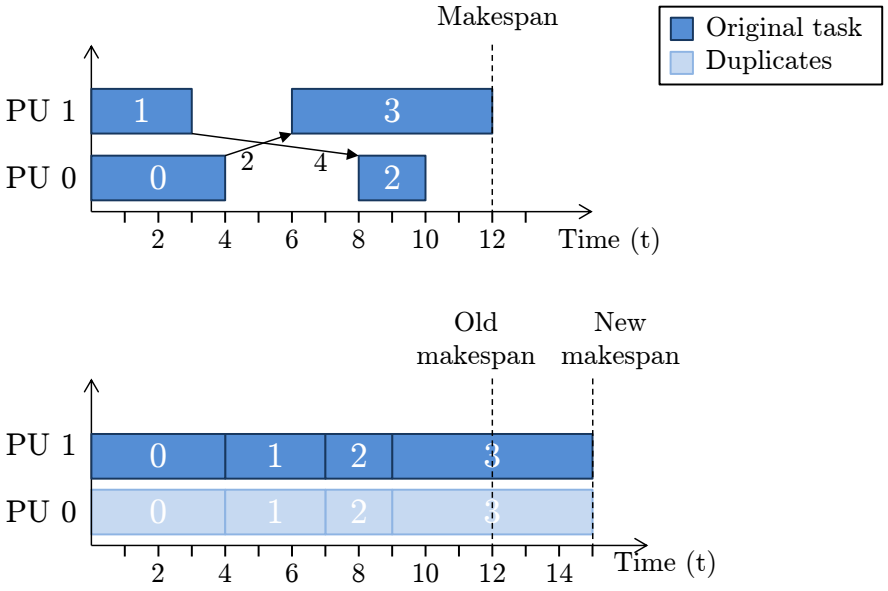

Figure 4.6: Example Schedule to Illustrate the Changes by the UHPO-option.

```
for(taskIndex = 0; taskIndex < numberOfTasks; taskIndex++) {
  earliestBeginning = -1;
  currentProcessor = beginProcessorList;
  halfProcessors = 0;
  while(currentProcessor != NULL && halfProcessors < numberOfProcessors/2)
      {
    beginning = 0;
    for(j = 0; j < task[i].predecessornumber; j++)
      if(predec[i][j].processorindex != currentprocessor->processorindex)
        if(predec[i][j].finishtime + predec[i][j].transfertime > beginning)
          beginning = predec[i][j].finishtime + predec[i][j].transfertime;
    if(currentprocessor->task != NULL) {
      currenttask = currentprocessor->task;
      while(currenttask->next != NULL)
        currenttask = currenttask->next;
      if(currenttask->finishtime > beginning)
        beginning = currenttask->finishtime;
    }
    if(earliestBeginning == -1 || earliestBeginning > beginning) {
      earliestBeginning = beginning;
      bestProcessor = currentProcessor;
    }
    halfProcessors++;
    currentProcessor = currentProcessor->next;
  }
  createtask(...);
  inserttask(bestProcessor, earliestBeginning,...);
}
```

Listing 4.1: Pseudo Code of the List Scheduler.

For each task (line 1) the earliest beginning is initially set to -1, the current processor is the first processor in the processor list and `halfProcessors` is set to zero (lines 2 - 4). Then, for each processor of the first half (line 5) the beginning of the new task is set to zero (line 6). The list scheduler checks when the starting time of the new task would be, because of the dependencies and transfer times of the predecessors (lines 7 - 10). If there exist tasks on the current processor (line 11), the finish time of the last task is compared with the beginning of the new task. Is the finish time later, the beginning has to be corrected (lines 12 - 17). Thus, also predecessors that are placed on the current processor (without any transfer times) are considered. Then, the overall earliest beginning and the corresponding processor are corrected (lines 18 - 21), `halfProcessors` is incremented and the next processor is checked (lines 22 - 24). Finally, when half of the available processors are checked, the new task is created and inserted on the best processor with the earliest start time (lines 25 - 27).

To place the duplicates, the best processor is fixed by using the processor index for the original task plus half of the processors. Thus, the duplicates are mapped in the same way like the original tasks, but on the other half of the processors.

4.2.3 Excursion: Use Duplicates for Delayed Tasks

Static task duplication is typically used either to reduce communication costs or to tolerate failures that occur during the runtime of a schedule. However, duplicates can also be used in other situations to improve the performance of a schedule. For example as demonstrated by Eitschberger and Keller [53] in case of a performance loss of a PU. Especially in grid systems, the owner of a PU might need a fraction of the computing power of a PU for other executions. In this case, the scheduled original task can only be executed with the remaining fraction of the computing power. Thus, the task is slowed down and the computation time is extended. When the execution of a slowed original task becomes longer than executing the corresponding duplicate, it can be worthwhile to abort the original task and execute the duplicate instead.

To handle such slowdowns, the completion time of the original task with the reduced performance has to be calculated during the runtime and sent via a message to the corresponding duplicate. Then, the predicted finish time of the duplicate can be compared with the completion time of the slowed original task. If the duplicate can finish earlier, an abort message is sent to the original task, otherwise the dupli-

cate is aborted. While task slowdowns are usually considered in dynamic scheduling, this approach is a combination of static and decentralized dynamic scheduling, i.e. a hybrid method. In Fig. 4.7 different situations are shown in which the execution of a duplicate can lead to a better performance of the schedule.

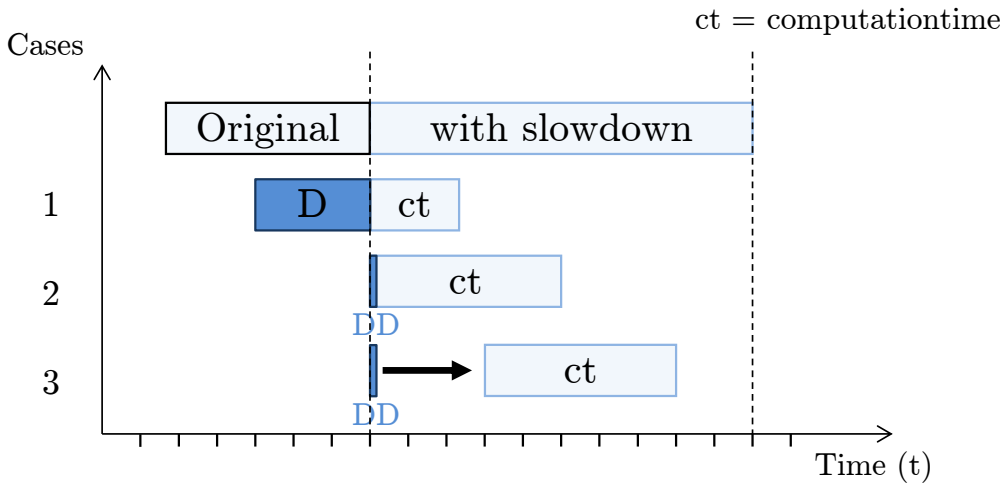

Figure 4.7: Cases of Placed Duplicates Compared with a Slowed Original Task [53].

In the first case, the duplicate starts during the execution of the original task. As the performance of the duplicate is not changed, it can finish earlier. In the second case a dummy duplicate is placed at the statically predicted end of the original task. As the slowdown of the original task leads to a higher prolongation, the dummy duplicate can be extended to a duplicate and then be executed. In the last case, the dummy duplicate is also placed at the end of the static predicted finish time of the original task. But in this case, a shift of the dummy duplicate because of some dependencies is only considered in case of a failure by the static scheduling. Thus, only if the shift and the computation time of the extended dummy duplicate leads to an improvement, the dummy duplicate is extended and executed, otherwise it is not started. This method can be considered by a corresponding runtime system because it affects the schedule dynamically during the runtime. Therefore, to evaluate this method it is implemented in the simulator. For each original task successively a slowdown is applied by a user-defined percentage rate. In every slowdown scenario the computation time of the slowed task is calculated and then two cases are simulated: On the one hand the schedule execution by using the slowed original task and all resulting shifts of the successor and succeeding tasks. On the other hand the schedule with the corresponding D or DD executed instead of the slowed original task. This is only considered if one of the situations in Fig. 4.7 is

given. Finally the resulting makespans of both cases are compared and the potential improvement of using the corresponding D or DD is saved.

Nearly 34,000 (performance optimal) schedules of the synthetic benchmark suite of Hönig [84] (see Sects. 4.2.1 and 6.1.1) are simulated for a slowdown of 50%, 6,800 schedules for a slowdown of 70%, and 10,164 schedules for a slowdown of 80%. In Tab. 4.3, the minimum, averaged and maximum results for strategies 1 and 2 are depicted.

Table 4.3: Improvement of Makespan with the Use of Duplicate in Case of 50 %, 70 % and 80 % Slowdown [53].

slowdown	**Strategy 1**			**Strategy 2**		
	Minimum	Average	Maximum	Minimum	Average	Maximum
50 %	0.00 %	0.00 %	0.00 %	0.98 %	3.03 %	31.25 %
70 %	0.13%	8.03%	39.36%	0.13%	8.55%	39.36%
80 %	0.31%	14.93%	43.18%	0.31%	15.39%	44.43%

In strategy 1 where DDs are used, an improvement can only be achieved for slowdowns that are higher than 50%, as a DD starts at the end of the corresponding original task and thus finishes at the same time like the original task with a doubled (50% slowed down) computation time. As expected, the improvements increases with an increased slowdown. However, while the averaged results seem small, a single task is slowed down in this experiments, the runtime of a task only remains a fraction of the total schedule runtime, so that a high improvement is not expected.

4.3 Energy-efficient Scheduling Heuristics and Options

In this section, two heuristics and an additional option are presented that mainly focus on the fault-free case. Then, several heuristics and options for the fault case are described that can be used to reduce the energy consumption or to invest energy for improving the performance in case of a failure.

4.3.1 Buffer for Energy Reduction (BER)

A schedule typically consists of several gaps resulting from the dependencies between the tasks. These gaps can (partly) be used to prolong tasks by scaling down the frequency of corresponding PUs to save energy. To avoid an increase of the overall

makespan, tasks can only be slowed down if they are not on the critical path. A slowed task can lead to a shift of the successor and succeeding tasks. However, as long as tasks of the critical path are not involved the makespan does not change. Therefore, the frequency to slowdown a task must be set appropriately. As an operating system does not have any information about a schedule, it can not scale the frequencies of supported PUs efficiently. Thus, the frequency scaling must be done by the scheduler or the underlying runtime system (see Sect. 2.5).

In the work of Eitschberger and Keller [54], a two-step greedy heuristic is proposed that reflects the explained behavior. They consider only tasks followed by a gap for a potential slowdown. In a first step, a so-called buffer is calculated for the tasks. The buffer represents the additional time that can be used to slowdown the task without affecting the makespan. In a second step, the lowest possible frequency for a task i is determined by considering the buffer b_i for the runtime r_i of the task; i.e. the current frequency can be scaled down by a factor $\frac{r_i}{r_i+b_i}$. If the calculated frequency is not supported, the next higher frequency is used and the runtime is set accordingly.

In Fig. 4.8, an example schedule with the calculated buffers is shown. There are gaps after tasks 0, 1 and 2 in the schedule. Thus, only these tasks are considered for a potential slowdown. The finish time of task 0 is at $t = 4$. The transfer of the results to task 3 needs $t = 2$ time units and task 3 starts at $t = 6$. Therefore, task 0 cannot be prolonged and the corresponding buffer is considered zero. Task 1 finishes at $t = 3$. The transfer time to task 2 is $t = 4$, but task 2 starts at $t = 8$. Thus, the buffer of task 1 results in $8 - 4 - 3 = 1$. Finally, task 2 ends at $t = 10$ but the makespan is at $t = 12$. As task 2 is an ending task, the distance to the makespan is considered and the resulting buffer is getting two time units. So tasks 1 and 2 can be slowed down.

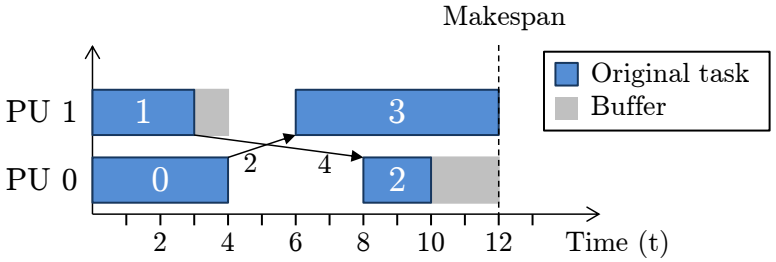

Figure 4.8: Example Schedule to Illustrate the Frequency Scaling Heuristic [54].

In case of a fault-tolerant schedule, Ds can be treated similarly. To achieve a high energy reduction, the Ds of slowed original tasks should be slowed down by the

same frequency. Thus, a buffer of an original task is reduced to the length of the corresponding duplicate buffer, if that is shorter. Buffers of successor and succeeding tasks can also be used to extend the buffer of an original task by shifting the tasks accordingly. DDs, in contrast, are not considered for a prolongation, because they are placed with runtime zero. Therefore, these are not executed in a fault-free case. They can only be shifted. For all remaining gaps the frequencies of the PUs can be scaled down for further savings to the lowest supported frequency, i.e. to the idle frequency. This heuristic is related to the fault-free case. However, it is indirectly also used in the fault case, e.g. in the EP-heuristic explained in the following. In Lst. 4.2 the pseudo code of the heuristic is presented.

```
1  Set all task->buffers to 0;
2  helpbuffer = 0;
3  gap = task->finishtime until next original task->starttime
4  for(all tasks followed by a gap) {
5    if(successor tasks exist)
6      task->buffer = min(distance to all successors - transfer times);
7    else {
8      if(task is an ending task)
9        task->buffer = makespan - task->finishtime;
10     else
11       task->buffer = gap;
12   }
13 }
14 for(all Ds, where original task->buffer > 0) {
15   if(D task->buffer < original task->buffer)
16     original task->buffer = D task->buffer;
17   helpbuffer = original task->buffer;
18
19   while(Ds exist in the gap (original PU))
20     consider all D task->buffers, reduce helpbuffer;
21   while(Ds exist in the gap (D PU))
22     consider all D task->buffers, reduce helpbuffer;
23   if(helpbuffer > 0) {
24     find the best frequency to fill the helpbuffer;
25     reduce the helpbuffer if necessary;
26   }
27
28   if(helpbuffer > 0) {
29     while(gap beginning is not reached (D PU), go backwards)
30       shift all Ds in the gap, reduce their buffers;
31     change/shift the D infront of the gap itself, reduce its buffer;
32     while(gap beginning is not reached (original PU), go backwards)
33       shift all Ds in the gap, reduce their buffers;
34     change the original task infront of the gap, reduce its buffer;
35   }
36 }
```

Listing 4.2: Pseudo Code of BER-heuristic.

The buffers of all tasks and the help buffer are initially set to zero (lines 1 - 2). In every case, where a gap is after a task, i.e. the finish time of a task is smaller than the beginning of the next original task, its buffer is firstly set either according to the minimum time, that can be used without shifting successor tasks, or to the gap size (lines 3 - 13). Thus, also duplicates[3] within a gap can have a buffer bigger than zero. Secondly, to avoid shiftings as a result of duplicates, the buffers of all original tasks are reduced to its corresponding duplicate buffers, if the duplicate buffers are smaller and the help buffer is always set to the original task buffer (lines 14 - 18). Then, the gaps between original tasks are checked. If there exist duplicates within a gap, these buffers are considered to keep the buffer size of the original task before the gap as large as possible (lines 19 - 20). Therefore, if a duplicate within a gap has a buffer and the beginning of the duplicate lies within the buffer of the original task before the gap, the duplicate could be shifted for the time of its own buffer. The beginning of the duplicate can then be used as restriction for the buffer of the original task and afterwards the duplicate buffer can be reduced for the time of the shifting. This procedure can be repeated for all duplicates within a gap until the next original task starts. The overall resulting buffer is saved in the help buffer. In the same way, the duplicates within the gap on the corresponding duplicate PU are considered (lines 21 - 22). The calculation of the overall buffer and the resulting shifting of all duplicates within a gap (and the reduction of their buffers) is separated in two while loops (lines 19 and 32 for the original task and lines 21 and 29 for the duplicate task), because only if the overall buffer, i.e. the help buffer, is higher than zero, any duplicate must be shifted. The buffer is then used to find the best frequency. A frequency is selected to fill the help buffer as much as possible by slowing down the task. If the help buffer cannot be filled completely, it is reduced to be filled by using the next higher supported frequency (lines 23 - 26). At the end, if the help buffer is still bigger than zero (line 28), all duplicates within the gaps can be corrected backwards from the end of the gap. Finally, the buffer and properties of the original task and its corresponding duplicate before the gap are corrected (lines 29 - 36).

Tasks are typically placed as early as possible in a schedule, so that only gaps after tasks must be considered. However, in general also buffers before tasks are possible, when tasks are placed later like task 2 in Fig. 4.8. The calculation of these buffers and the corresponding frequencies of tasks can then be done in a similar way, but from the end of a gap to the beginning. This leads to different frequencies and a difference in energy consumption. While the consideration of both buffers, i.e. before

[3]In this context, the term duplicates is used for Ds and DDs.

a task and after a task, can lead to better results, a much more complex algorithm and a longer scheduling time is necessary to find optimized solutions. A decision about using a buffer then also influences the buffer usage of other tasks, i.e. the starting (and ending) times of tasks, which results in several additional solutions to be checked. As in the synthetic benchmark suite used in Sect. 6.1.1 tasks are always placed as early as possible, the BER-heuristic is restricted to such cases. Therefore, buffers before tasks are not considered.

4.3.2 Option: Insert Order (SDE vs. SED)

When considering both fault tolerance by duplicates and heuristics to reduce energy consumption like BER, the insert order can have a significant influence on the optimization. While placing the duplicates first and reduce the energy afterwards improves the FT, the vice versa handling, in contrast, can result in a lower energy consumption. Therefore, the user can select the order *SED (Scheduling → Energy → Duplicates)* and *SDE (Scheduling → Duplicates → Energy)*. In Fig. 4.9 both situations are illustrated based on the initial example.

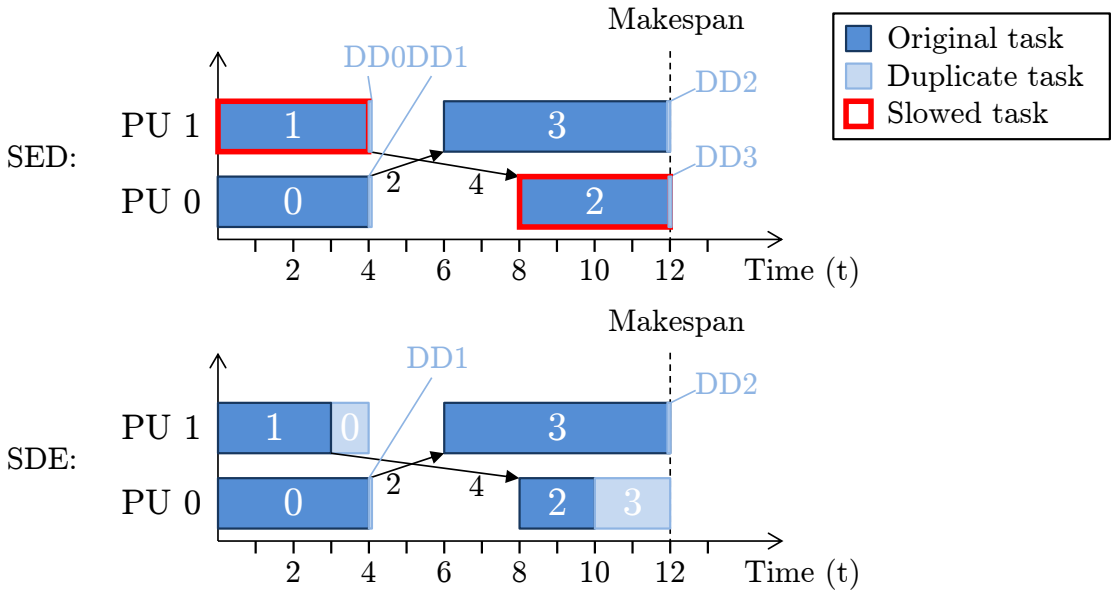

Figure 4.9: Example Schedule to Illustrate the Changes by the Insert Order [54].

In the upper case (SED), the original tasks are slowed down first by the BER-heuristic to reduce the energy. Then, the duplicates are included. Thus, tasks 1 and 2 are slowed down by using a lower frequency and only DDs can be placed for all tasks. In contrast, in the bottom case (SDE) the duplicates are placed first.

Therefore, the buffers that were used to slowdown the tasks in the upper case are now used to extend some DDs to Ds, when using the second strategy. While in this example the energy cannot be reduced anymore by the BER-heuristic, there exist other situations where only parts of the buffers are used for the Ds. Thus, the tasks can be slowed down for a shorter time. The insert order is related to the fault-free case only, because the duplicates are placed prior to execution and only one failure is considered.

4.3.3 Change Base Frequency (CBF)

In a static schedule without fault tolerance or energy efficiency aspects, typically one fixed frequency is assumed for the whole schedule, i.e. for all tasks and gaps. Therefore, with this strategy the base frequency for the schedule can be changed according to the user preferences. Scaling the frequency up or down leads to a change of the runtime for all tasks. As the mapping of the tasks is not modified in this heuristic, a change can lead to some shifts, so that the placement of the tasks must be corrected. By considering the example schedule above, the resulting schedule after a scale down of the base frequency is illustrated in Fig. 4.10.

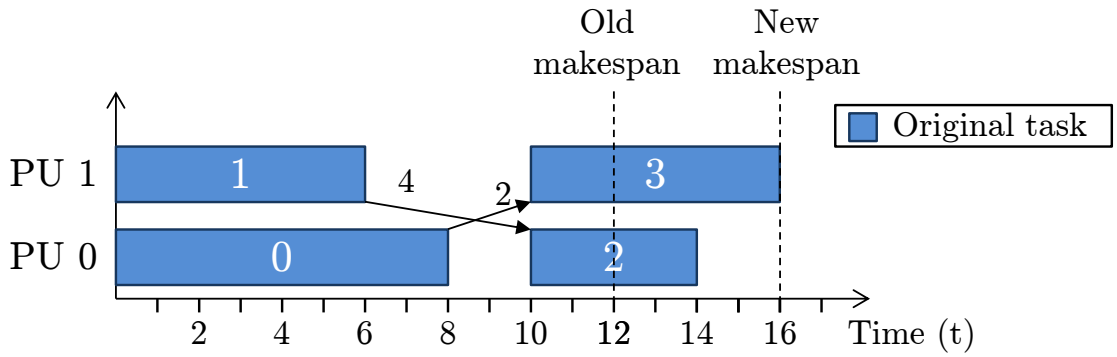

Figure 4.10: Example Schedule with a Low Base Frequency.

With a lower frequency the makespan is increased and the tasks are prolonged, but the energy might be reduced. This heuristic can also be used to improve the performance of the schedule by scaling up the frequency and thus shortening the runtime of tasks. Therefore, the CBF-heuristic can be used separately for both the fault-free case and the fault case. In Lst. 4.3 the pseudo code of the heuristic is presented.

```
1  readschedulefile(...);
2  createsprocessorlist(...);
3  for(all tasks){
4    newtask = createtask(...);
5    inserttaskinprocessorlist(...);
6  }
7  for(all tasks){
8    change properties of the task;
9    shift(...);
10 }
```

Listing 4.3: Pseudo Code of CBF-heuristic.

Initially, the already existing schedule (and task graph) file is read (line 1). A processor list with the number of PUs is created (line 2). Then, all tasks are created and inserted in the processor list on the corresponding PU (lines 3 - 6). Until now, the schedule is created as given by the schedule and task graph file.

Then, for all tasks the properties are adapted by changing the base frequency to the frequency (level) given by a user (lines 7 - 8). Firstly, the computation time and the finish time of a task are changed. Secondly, the frequency used is saved. Finally, the energy consumption is re-calculated with the new frequency. After all properties are adapted, the successor and succeeding tasks are shifted according to the prolongation or shortening of a task or according to the dependencies between the tasks (lines 10 - 11). The mapping of the tasks is, therefore, not changed. Only the beginning and ending of the tasks are modified. Gaps within a schedule are sometimes filled by the prolongation of tasks and often new gaps arise by the resulting shifts of other tasks.

4.3.4 Energy for Performance (EP)

A failure of a PU during the runtime of a schedule typically leads to a performance overhead. While in a fault-free case the focus might be on the energy, this could change in case of a failure. Then, a fast termination of a schedule can be preferable. As demonstrated by Eitschberger and Keller [54], energy can be invested to improve the performance of a schedule in case of a failure by scaling up frequencies and thus speeding up the execution of tasks. One possibility is to consider all Ds and DDs that have to be extended due to the failure and calculate for each of these tasks a higher frequency that can be used to reduce or totally undo the resulting overhead. When the calculated frequency for a task is not supported by the system, the next higher or the highest possible frequency is used instead. In this heuristic,

each task can earliest start at the time given in the static schedule, independently of dependencies and transfer times. An earlier start is not allowed. In Fig. 4.11 a fault-tolerant schedule based on the example for the option SDE, see Fig. 4.9 (bottom case), is shown for two failure cases. For simplification, a normalized frequency of 1 is assumed in general and a continuous frequency range between 0.1 and 3. Therefore, all tasks run initially with frequency 1.

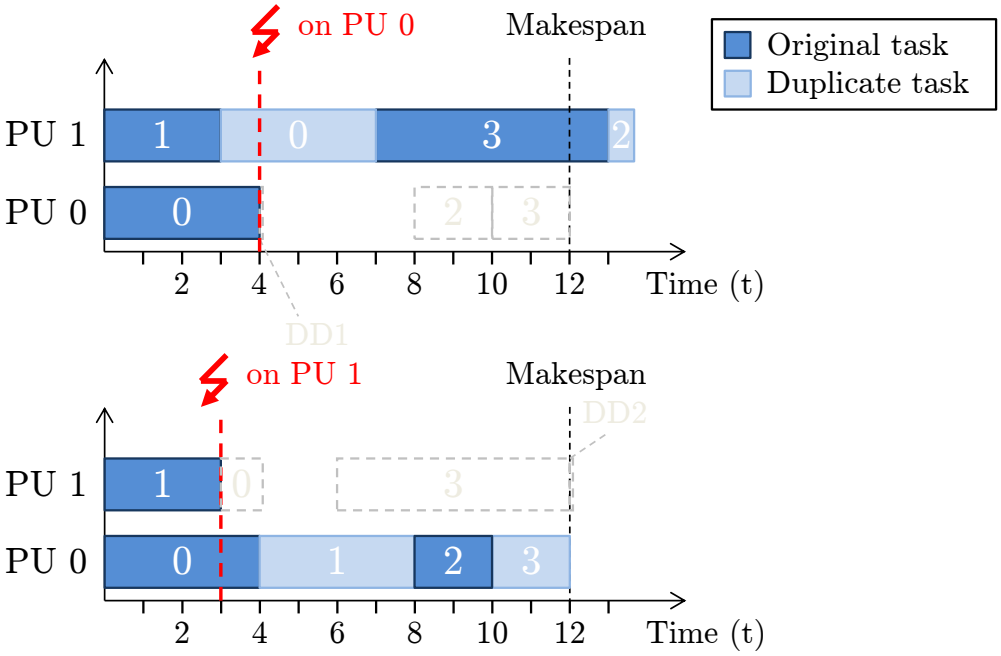

Figure 4.11: Schedule in Case of a Failure with EP-option.

In the upper part of Fig. 4.11, a failure occurs on PU 0 at the end of task 0. The duplicate 0 on PU 1 runs for one time unit at frequency 1. As the commit message from task 0 is not received, the duplicate has to be extended. As a task can only run with a constant frequency and no frequency changes are allowed during the execution of a task, the remaining part of the duplicate (3 time units at frequency 1) does not fit into the existing gap of two time units after the duplicate. Therefore, the succeeding original task 3 is shifted by one time unit. The DD2 at the end of the schedule can start at $t = 13$. The runtime with the highest frequency results in $\frac{2}{3} = 0.67$ time units. Thus, the resulting overhead cannot be undone completely. In contrast, in the bottom case the same schedule is depicted with a failure on PU 1 at the end of task 1. In this case, the extended DD1 that is placed at $t = 4$ fits into the gap and one time unit is still free to slow down the task. Thus, the new frequency can be calculated by $\frac{3}{3+1} = 0.75$ and the gap is closed. The duplicate 3

starting at $t = 10$ was initially scheduled with frequency 1. As the information about the failure of PU 1 was already known before, the frequency for duplicate 3 can directly be scaled up. Thus, the runtime of duplicate 3 still remains two time units but now with frequency 3. The makespan does not change and an overhead can be avoided. Therefore, it is highly dependent on when a failure is occurred and which frequencies are supported by the system used.

As Ds and DDs are always extended in case of a failure and only those tasks are considered in the EP-heuristic, the implementation can directly be included in the functions to check and change the properties of tasks in case of a failure. Then, the new frequency and the resulting computation time of a task can be calculated similar to the calculation in the BER-heuristic. Thus, tasks are speeded up if necessary, i.e. if no buffer exist or the buffer is smaller than the extended duplicate. And tasks are also slowed down if possible, when the buffer size of a task is bigger than the time needed to execute the extended duplicate. However, this heuristic is restricted to duplicates and only considers the resulting overhead for each duplicate separately.

4.3.5 Option: Delete Unnecessary Duplicates (DUD)

In case of a failure, PUs might be informed directly about the crashed PU, e.g. by broadcasting a message to all remaining PUs. Then, all unnecessary duplicates do not have to be executed and can be deleted, because only one failure is tolerated per schedule. In contrast, if the information about a failure is not sent directly to all remaining PUs, each PU gets the information separately when it sends a message to the crashed PU. Therefore, with this option different system behaviors in case of a failure can be set.

4.3.6 Lazy Frequency Re-scaling (LFR)

To optimize the energy for a given deadline in case of a failure, the frequencies of tasks can be re-scaled (without changing the mapping). As deadline, the makespan of the fault-free schedule is assumed. Eitschberger and Keller present in [56] an online scenario where the frequency for a task is set according to the remaining time and workload on the corresponding PU until the deadline. Therefore, for each task the difference between the deadline and the start time is calculated and divided by the sum of the workloads of all remaining tasks on the PU (originals and duplicates). Thus, the frequencies are always set to the lowest possible. However, because of gaps between tasks and shifts that result from extending DDs, this heuristic ends up with

high frequencies close to the deadline and is, therefore, called *lazy*. This heuristic is only related to the fault case.

Consider the initial example schedule and a failure on PU1 at the end of task 1, similar to the example in Fig. 4.11 bottom case. In Fig. 4.12 upper case, the schedule is depicted without any frequency scaling. An overhead of four time units is present. For simplification, a normalized frequency of 1 is assumed in general and a continuous frequency range between 0.1 and 3. Therefore, all tasks run initially with frequency 1.

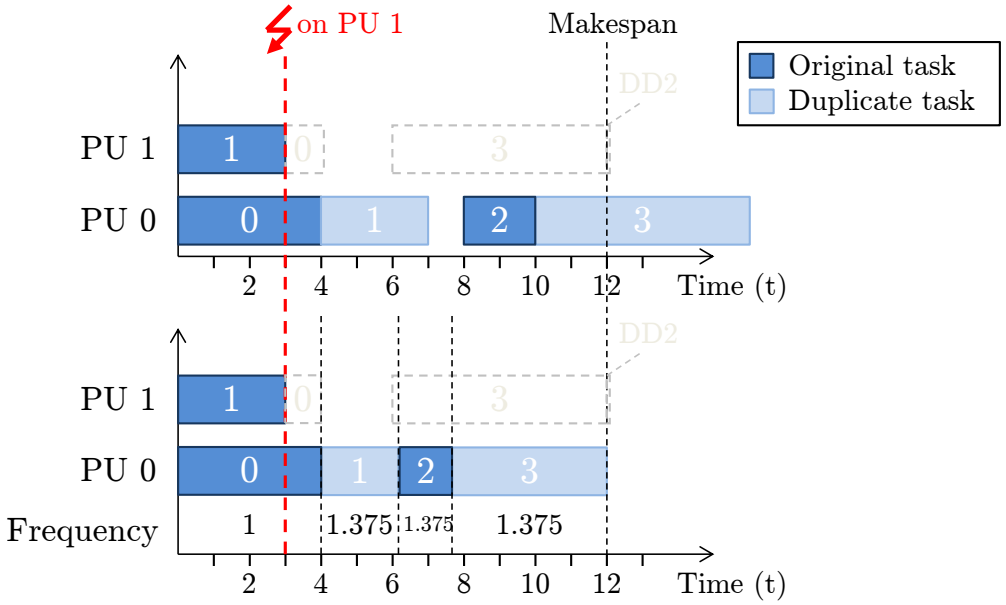

Figure 4.12: Example Schedule to Illustrate the Changes by the LFR-heuristic.

In the bottom case of Fig 4.12, the lazy frequency scaling is shown. In the fault-free case, all tasks are running with the normalized frequency 1. At time unit 4 the duplicate of task 1 on PU 0 has to be executed. Now, to calculate the new frequency, the sum of the remaining task workloads on this PU is divided by the remaining time until the deadline, i.e. $\frac{3+2+6}{12-4} = \frac{11}{8} = 1.375$. Duplicate 1 is executed with a frequency of 1.375. As in this example only two PUs exist (where one PU has failed) no transfer time is considered, because all remaining tasks must run on PU 0. In contrast to the EP-heuristic (where each task can earliest start at the time given by the static schedule), the tasks are allowed to start when all results of the predecessors are available. Therefore, tasks are not only shifted forward but also sometimes backwards if possible. Hence, task 2 can directly be started after D1 and also task 3 after task 2 with a frequency of 1.375 and without any overhead.

However, in this example the lazy frequency scaling behavior is invisible, because there are no transfer times to be considered and thus all remaining tasks are running with a constant frequency of 1.375. To demonstrate the "lazy" behavior, an extended example is given in Fig. 4.13.

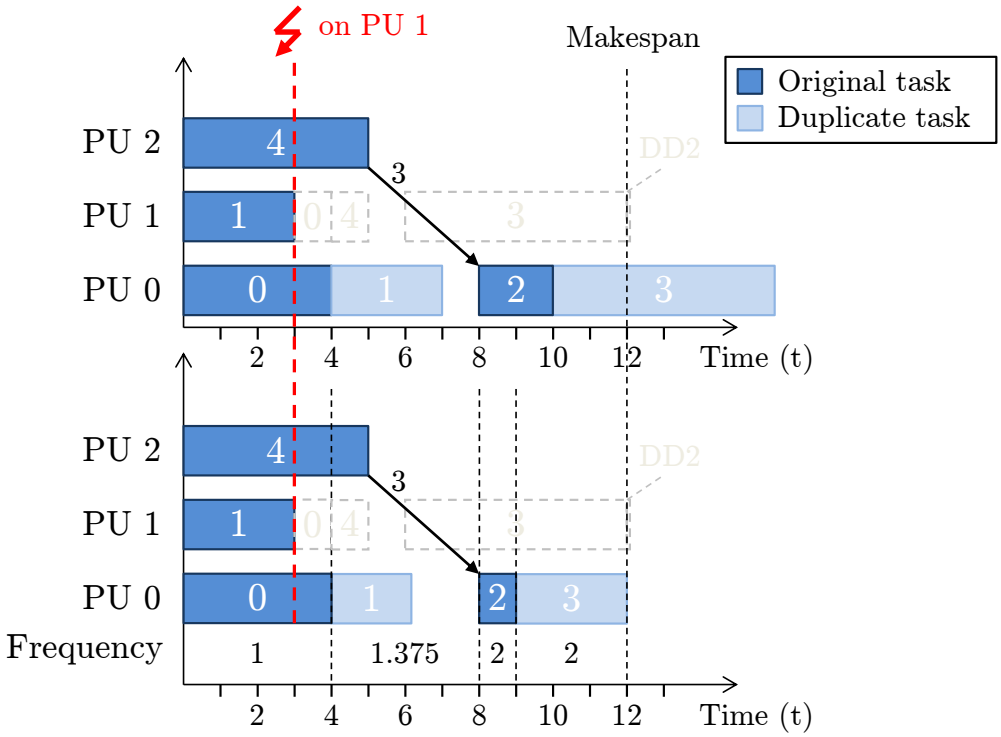

Figure 4.13: Extended Example Schedule to Illustrate the Changes by the LFR-heuristic.

In this example, yet another task (task 4) is executed on an additional PU (PU 2) and has to send its result to task 2, that takes three time units. Therefore, task 2 can not be started before $t = 8$. In the upper case, the new schedule without any frequency scaling is depicted. The overhead is the same like in the example before. In the bottom case, task 2 can only be started at $t = 8$ because it has to wait for the results of task 4. Therefore, at the beginning of task 2 a new frequency is calculated and used to execute the task, i.e. $\frac{2+6}{12-8} = \frac{8}{4} = 2$. D3 is then also be executed with frequency $\frac{6}{3} = 2$. In this example the lazy behavior can be seen better. On PU 0, firstly task 0 is running with frequency 1, then D1 with frequency 1.375 and finally task 2 and 3 with frequency 2. In Lst 4.4 the pseudo code of the heuristic is presented.

```
1  Task *currenttask = NULL;
2  currenttask = task;
3
4  while(currenttask != NULL){
5    if(currenttask is an original or a duplicate that has to be executed)
6      totalworkload = totalworkload + currenttask->comptime;
7    currenttask = currenttask->next;
8  }
9
10 newfreq = totalworkload/(makespan - task->starttime);
11 task->finishtime = task->starttime + task->comptime * task->freq/newfreq;
12 task->comptime = task->comptime * task->freq/newfreq;
13 task->freq = newfreq;
14 task->energyconsumption = setenergy(...);
15 check if adapted properties influence position of successor tasks;
16 check if adapted properties influence position of succeeding tasks;
```

Listing 4.4: Pseudo Code of LFR-heuristic.

In the beginning, a pointer `currenttask` is created and initialized with the task to be considered (lines 1 - 3). Then, the total workload of all succeeding tasks that have to be executed is summed up (lines 4 - 9). The new frequency for the task is the total workload divided by the time until the makespan (line 10). The finish and computation times of the task are calculated based on the new frequency (lines 11 - 12). The new frequency is saved as task frequency and the energy consumption for the task is calculated (lines 13 - 14). Finally, the influences on successor and succeeding tasks of the adapted task are checked and changed if necessary (lines 15 - 16).

In contrast to the EP-heuristic, that only focuses on the overhead for each duplicate, the LFR-heuristic only considers the total workload of a PU to find a new frequency. Thus, neither the overhead of a task nor the dependencies between tasks are considered.

4.3.7 Constant Power (CP)

Another approach to re-scale the frequencies of tasks in case of a failure is the so-called *CP-heuristic*, demonstrated by Eitschberger and Keller [56]. This heuristic is based on the approximation algorithm from Pruhs et al. [147] (see Sect. 3.1.2). To obtain an energy-optimal solution, the power consumptions of all PUs are kept constant. Therefore, the frequency of a PU is set according to the fraction of the power budget that is related to the PU. This fraction changes with the number of PUs that are executing tasks at the same time. While several distributions of

the power budget are possible that may lead to an improved energy consumption, a simple uniform distribution of the power budget is used in this approach for all active PUs to get a short scheduling time. Thus, neither the static power consumption nor the total workload or the overhead are considered. When a task finishes or a new task starts, the power budget is re-distributed over all PUs and the frequencies are adapted. Thus, a discrete event simulation can be used, where the events (a task starts or a task finishes) are saved in a priority queue. In this queue, the priorities can be changed in both directions. The resulting runtime of a task is then used to calculate an averaged frequency, because only one frequency per task is assumed. The minimum energy consumption for a schedule is found by nested intervals with the assumption that a lower power budget leads to a longer runtime. The nested intervals stop if the interval is small enough. This heuristic is only considered in the fault case.

In Fig. 4.14 the resulting schedule of the extended example explained for the LFR-heuristic is shown by considering the CP-heuristic. In this example, a simple quadratic power model, i.e. $P(f) = f^2$ is assumed and a power budget of 4.

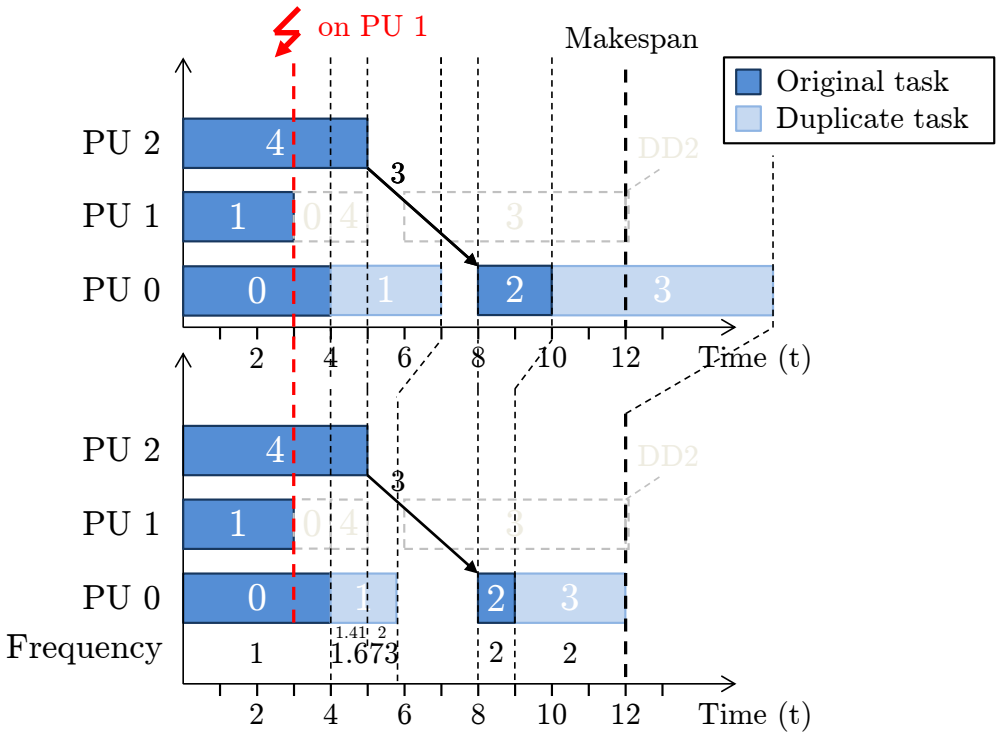

Figure 4.14: Extended Example Schedule to Illustrate the Changes by the CP-heuristic.

A failure of PU 1 occurs at the end of task 1. As task 0 and task 4 are running at this time, the frequencies of both tasks are not changed by the CP-heuristic, because only one frequency per task is allowed. At $t = 4$ the duplicate of task 1 can start on PU 0. As two PUs are active at this event, the resulting frequency f for PU 0 can be calculated by $f = \sqrt{\frac{\text{power budget}}{\text{active PUs}}} = \sqrt{\frac{4}{2}} \approx 1.414$ and the new runtime of duplicate 1 results in 2.12 time units. At t = 5, task 4 is finished so that the power budget can be completely used by PU 0 with a frequency of 2. Thus, the remaining part of duplicate 1, i.e. $(2.12 - 1) = 1.12$ time units at frequency 1.414 can be executed with the higher frequency. The overall runtime of duplicate 1 is then $1 + \frac{1.12 \cdot 1.414}{2} = 1.793$ and the average frequency for duplicate 1 is $\frac{3}{1.793} \approx 1.673$. All remaining tasks on PU 0 are running with a frequency of 2, because PU 0 is the only active PU until the end. In Lst. 4.5 the pseudo code of the heuristic is presented.

```
while(there are tasks to be executed){
  PU = nextevent(...);

  if(PU is not activated){ // A new task starts
    activate PU;
    activePUs++;
    oldfrequency = newfrequency;
    newfrequency = sqrt(totalpowerbudget/activePUs);
    for(all PUs)
      if(PU is activated and task starts after or at failure)
        adapt remaining runtime according to the new frequency;
    timestamp = task start time;
  }
  else{ // A task finishes
    deactivate PU;
    activePUs--;
    oldfrequency = newfrequency;
    newfrequency = sqrt(totalpowerbudget/activePUs);
    for(all PUs)
      if(PU is activated and task starts after or at failure)
        adapt remaining runtime according to the new frequency;
    timestamp = task finish time;
    take next task on PU;
  }
}

for(all tasks){
  calculate average frequency according to the runtime;
  set energy for task;
}

calculate makespan;
return(makespan);
```

Listing 4.5: Pseudo Code of CP-heuristic.

As long as there are tasks to be executed (line 1):

- Find the PU with the next event, i.e. where the next task starts or finishes. In the function `nextevent(..)` unconsidered duplicates are deleted if the DUD-option is set (line 2).

- If a new task starts (line 4), the corresponding PU is activated and the active number of PUs is incremented (lines 5 - 6). Then, the old and new frequencies are set (lines 7 - 8). The tasks (after or at the failure) of all active PUs, including the PU with the new started task are considered to change the remaining runtime according to the new frequency (lines 9 - 11). Finally, the time stamp is updated with the tasks' starting time (lines 12 - 13).

- If a task finishes (lines 14), the procedure is similar to the above explained. First, the corresponding PU is deactivated and the number of active PUs is decremented (lines 15 - 16). The frequencies are set and the runtimes of the active tasks excluding the finished task are adapted (lines 17 - 21). The time stamp is set to the finish time of the task. The next task is chosen. In both cases, i.e. a task starts or finishes, the old frequency and the time stamp are necessary to determine the previous time interval with the old frequency for the calculation of the remaining runtime with the new frequency.

After all tasks are completed, an average frequency for each task and the corresponding energy is calculated according to the adapted runtime of a task (lines 27 - 30). Then, the makespan of the schedule is calculated and returned (lines 31 - 33). Thus, a resulting makespan and the corresponding schedule is given.

An optimized solution can now be found by using nested intervals. Starting with a maximum power budget $PB_{max} = f_{max}^2 \cdot PUs$, i.e. all PUs are running with the highest frequency f_{max}. If the makespan of a schedule by using the CP-heuristic is higher than the makespan in a fault-free case, a solution is infeasible and the nested intervals stop. Otherwise, the mean power budget of the interval between the maximum power budget and the minimum power budget $PB_{min} = f_{min}^2 \cdot PUs$, i.e. all PUs are running with the lowest frequency f_{min}, is chosen. Then, the interval is halved in each step and the new mean power budget is calculated. If the makespan is higher than the makespan in a fault-free case, the minimum power budget is set to the mean power budget, otherwise the maximum power budget is set to this value. The nested intervals end, when either the makespan of the schedule by using the

CP-heuristic m_{cp} is close to the makespan m in a fault-free case, i.e. $|m_{cp} - m| \leq \epsilon$, where $\epsilon = 0.001$, or when m_{cp} does not change anymore between two steps.

4.3.8 Option: Maximum Makspan Increase (MMI)

As the static task scheduling is done prior to execution (see Sect. 2.3.1), the makespan is already known when the schedule starts. A crash of a PU during the runtime of a schedule usually leads to an overhead, i.e. a makespan increase. To undo the overhead or at least to keep it small, the tasks can be accelerated by scaling up the frequencies as much as possible. However, a small prolongation in case of a failure can basically be acceptable for a user. Therefore, a maximum makespan increase to be tolerable can be set as a percentage value to restrict the overhead and thus to scale up frequencies only in a moderate fashion. As demonstrated by Eitschberger and Keller [56], a given maximum makespan increase in case of a failure often results in a lower energy consumption. In some cases, the energy consumption for the faulty schedule even becomes lower than for its corresponding fault-free variant when considering an increased makespan. This option can only be used in combination with the two heuristics LFR and CP explained above. Otherwise, the maximum makespan increase is set to zero percent and thus switched off.

4.4 User Preferences and Corresponding Strategies

As seen in Chap. 3, an overall solution for the trade-off between PE, FT and E does not exist. A solution is very situational and highly depends on the user preferences. The focus can change for different systems and between the fault-free and fault case. Often more than one criterion are important for a "good" solution. Next to this fact, it is hardly possible to rate the preferences of a user in a general way like using a percentage value for the different criteria. Instead, several strategies should be provided by the scheduler to reflect the preferences of a user in detail. Therefore, in this section firstly valid combinations of the above explained heuristics and options are discussed. Secondly, combined strategies for the fault-free and fault case are proposed. The strategies have been developed and integrated in the existing scheduler.

4.4.1 Valid Combinations

The heuristics and options explained above are either related to the fault-free, to the fault case, or to both. An overview of the relationship is depicted in Fig. 4.15. While the option SDE/SED and the strategy UHPO are only related to the fault-free case, BER and CBF can be used in both cases. EP, DUD, MMI, LFR and CP are only related to the fault case, where LFR and CP can be influenced by MMI.

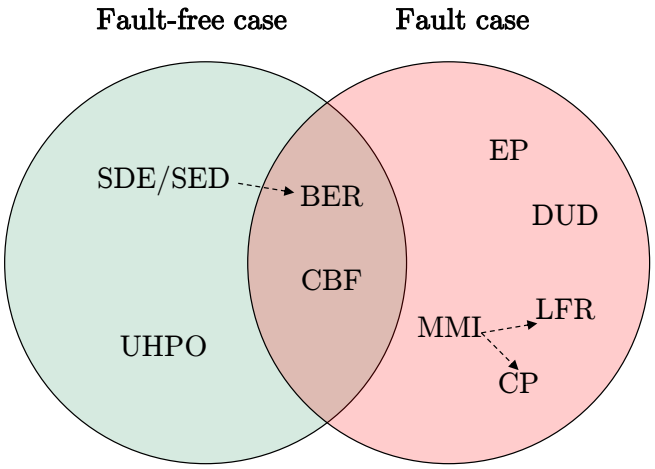

Figure 4.15: Heuristics/Options and their Relationship to the Fault-free and Fault Case.

As the user preferences might change between the fault and fault-free case, all options and strategies for the fault-free case can be combined with the options and strategies for the fault case. Therefore, only the remaining combinations are described in the following.

SDE/SED & UHPO The UHPO-heuristic changes the initial schedule before including duplicates or energy efficiency aspects. Therefore, the combinations SDE $\leftrightarrow$ UHPO and SED $\leftrightarrow$ UHPO are possible in general, but both lead to the same schedule. When using only half of the PUs for the original tasks and the other half for the duplicates, the placement of the duplicates is not influenced by any energy option or heuristic and vice versa. Thus, the order of including duplicates first or saving energy first is irrelevant for the resulting schedule.

SDE/SED & BER Whenever duplicates are placed onto PUs containing original tasks, the order of placing duplicates and scaling down frequencies is important. Hence, this combination is mandatory.

SDE/SED & CBF The insert position of a duplicate depends on the structure of a schedule, i.e. on the distribution of gaps and on the dependencies between tasks. As original tasks are prolonged or shortened and often shifted when changing the base frequency, some gaps would be closed, others might arise. Thus, an initial placement of duplicates before using the CBF-heuristic would lead to further overhead that could be avoided when using the CBF-heuristic first. Therefore, the CBF-heuristic has to be considered initially. Afterwards the SDE/SED-option can be used for further savings. Hence, a combination of both is possible, but the SDE/SED-option exclude the change of the base frequency.

UHPO & BER This combination is possible, too. First, the UHPO-heuristic is used to re-schedule the original tasks. After that, the BER-heuristic takes place. As explained above, the BER-heuristic has to be considered together with the SED/SDE-option.

UHPO and CBF In general, using half of the PUs for originals and changing the base frequency can be used together. In this case, initially the original tasks are re-scheduled onto half of the PUs. Afterwards the base frequency is changed before including the duplicates.

BER- & All Heuristics for the Fault Case The BER-heuristic can be combined with all fault strategies. However, the frequencies for duplicates resulting from the BER-heuristic are changed whenever the user focus on another direction in case of a failure. This means: By using any heuristic, the frequencies of duplicates are changed according to the heuristic in use.

CBF & All Heuristics for the Fault Case The CBF-heuristic can be combined with the heuristics for the fault case analog to the BER-heuristic.

DUD & All Heuristics for the Fault Case Deleting duplicates before using other heuristics or options in a fault case is always possible, because the DUD-option does not influence the frequency scaling negatively.

EP & (LFR or CP) These combinations are impossible, because the heuristics focus concurrently on the performance or the energy consumption and therefore change the frequencies in different ways.

EP & MMI As the EP-heuristic scales up the frequencies of duplicates according to the arising overhead, the makespan is neglected. Therefore, setting the MMI-option to a value higher than zero does not change the resulting schedule.

4.4.2 Strategies Fault-free Case

The following strategies can now be obtained by using the duplicate placement strategies, i.e. use only DDs or use Ds and DDs together with a valid combination of the heuristics and options as described above. To provide a variety of different preferences, several corner cases are considered. The alignment of the combined strategies and therefore the desired user preferences are illustrated in Fig. 4.16. Next to the new strategies also older ones (framed with dots) from the previous work without any energy efficiency aspects are shown for comparison.

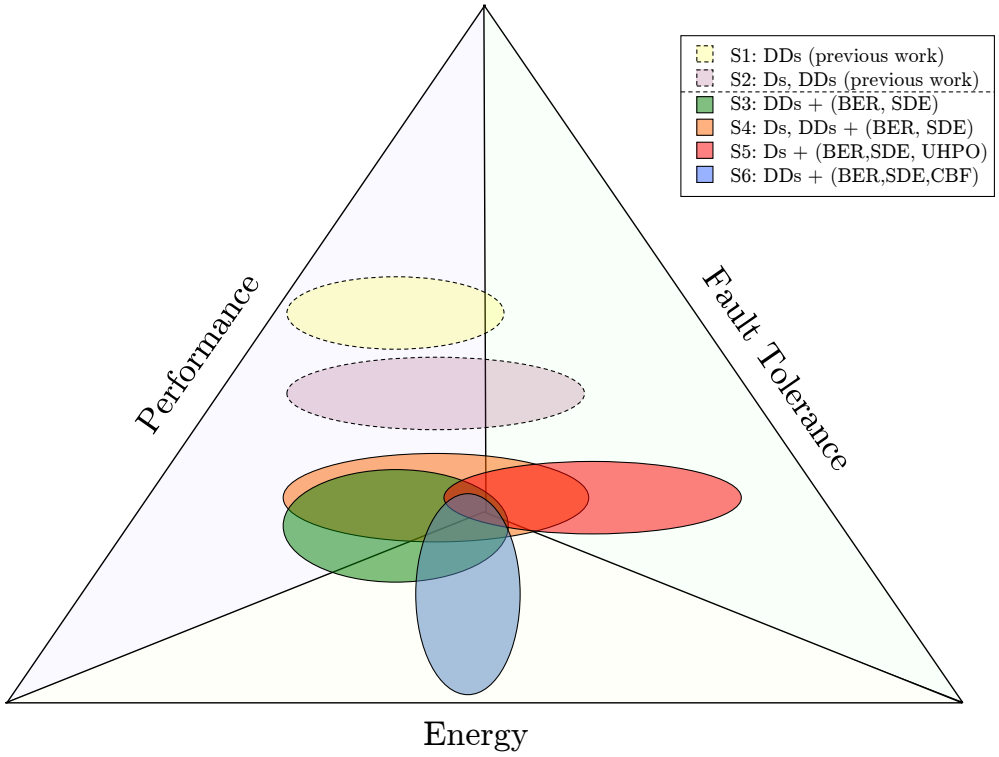

Figure 4.16: Alignment of Different Strategies for the Fault-free Case.

Strategy 1 + 2: DDs, DDs + Ds (Previous Work) In these strategies, the makespan of an initial schedule is not changed by including DDs and Ds. Thus, the performance in both cases is high. In strategy 2 the fault tolerance is improved by converting DDs into Ds where possible. In these strategies, the energy consumption

is not considered. Both are only related to the performance and the fault tolerance. Therefore, with these strategies the main focus lies on the performance and secondly on the fault tolerance.

Strategy 3: DDs + (BER, SDE) This strategy is a combination of the first one, i.e. use only DDs with the BER-heuristic and the SDE-Option. Therefore, the DDs are placed before using any buffers to slowdown tasks by scaling down the frequencies. The performance and the fault tolerance are equal to strategy 1. But the energy consumption is decreased by using the BER-heuristic. Thus, with this strategy the user mainly focuses on the performance and secondly on the energy consumption. The fault tolerance is considered, but should not influence the other criteria.

Strategy 4: Ds, DDs + (BER, SDE) Instead of using only DDs to improve the fault tolerance, here Ds are used, too. Thus, the performance is still the most important criterion, but energy is invested to get a better fault tolerance. Therefore, with this strategy the user would like to compensate (partly) the energy spent for the fault tolerance by slowing down tasks and reducing the energy consumption with the BER-heuristic.

Strategy 5: Ds + (BER, SDE, UHPO) In strategy 5, the second strategy is combined with the BER-, UHPO-heuristic and the SDE-Option. The main focus lies on the fault tolerance, i.e. half of the PUs are used for duplicates, the other half for original tasks. However, the performance is significantly decreased because tasks have to be re-mapped onto fewer PUs and the energy consumption is significantly increased, because the whole schedule is executed twice. Only a small benefit can be gained by using the BER-Option.

Strategy 6: DDs + (BER, SDE, CBF) To achieve a low energy consumption, the base frequency of tasks is changed and only DDs are used for the fault tolerance. The performance is significantly decreased. Additionally, the BER-heuristic is considered, to further reduce the energy consumption. Hence, the main focus of the user is put on the energy consumption and only a minor part on the other criteria.

4.4.3 Strategies Fault Case

In the fault case, only energy consumption and performance have to be considered, because only one failure per schedule is assumed. To compare all strategies in the fault case, all heuristics are combined with the first strategy, i.e. using only DDs. In Fig. 4.17 the alignment of the combined strategies in the fault case are illustrated.

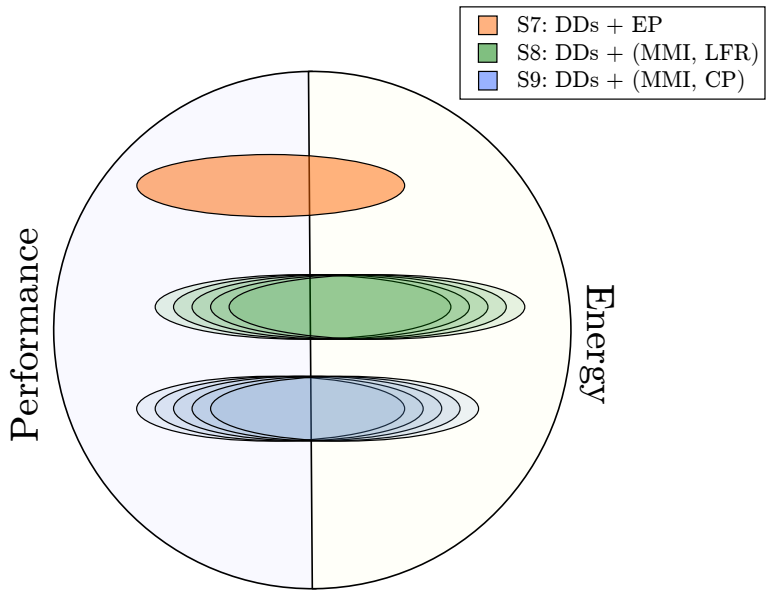

Figure 4.17: Alignment of Different Strategies for the Fault Case.

Strategy 7: DDs + EP In this strategy, the main focus is put on the performance. Duplicates are speeded up whenever an overhead arises. Additionally, the energy consumption is considered as duplicates are slowed down where possible. The strategy is advantageous for schedules with strong dependencies between the tasks.

Strategy 8: DDs + (MMI, LFR) Strategy 8 focuses on the energy consumption. The workload of each PU is considered separately. But in several cases the performance is decreased, because the frequencies are often chosen too low in the beginning, so that an overhead cannot be compensated in the end by using higher frequencies. A maximum makespan increase can be set to allow a performance loss.

Strategy 9: DDs + (MMI, CP) The focus of this strategy is also put on the energy consumption, but in contrast to strategy 8, the performance is kept constant. Therefore, the energy consumption is usually higher than using strategy 8.

This strategy is advantageous for schedules with a uniform distribution of the total workload. A maximum makespan increase can be set to improve the energy consumption.

4.5 Energy-optimal Solutions and Approximations

When the focus is put on the energy, an energy-optimal solution would be preferable. As the scheduling time for optimal solutions is disproportionally high compared to the energy savings, heuristics like the above explained are often used in practice. However, energy-optimal solutions (calculated at least for small schedules) can be used to rate the quality of the heuristics. Therefore, as demonstrated by Eitschberger and Keller [56], a mixed *integer linear program (ILP)* formulation is described as follows to compute energy optimal solutions in both the fault-free and the fault case. For simplicity, a quadratic power model is assumed, i.e. $P(f) = f^2$ for a frequency f, but also other power profiles are possible. Additionally, in case of a failure another alternative is to re-schedule the remaining part of the schedule [56] instead of using the pre-calculated mapping and ordering. Hence, an approximation and an optimal solution for the approaches is described for the fault case, i.e. using the existing mapping and using the re-scheduling approach for comparison.

4.5.1 Fault-free Case

To schedule n tasks onto a target system with p PUs and K frequency levels f_k, $n \cdot p \cdot K$ binary variables $x_{i,j,k}$ are needed with $x_{i,j,k} = 1$ if and only if task i is scheduled onto PU j with frequency level k. The energy consumption of a task i running at frequency level f_k is then the product of task runtime r_i and the power consumption $P(f_k)$:

$$E_i = \sum_j \sum_k x_{i,j,k} \cdot r_i \cdot P(f_k). \tag{4.1}$$

The runtime of a task at frequency level f_k depends on the task workload w_i and can be expressed by

$$r_i = \sum_j \sum_k x_{i,j,k} \cdot \frac{w_i}{f_k}. \tag{4.2}$$

The total energy consumption of a schedule for a given deadline Dl can then be obtained by summing up the energy consumption of all tasks (assuming that the

idle power and therefore the energy for all idle times is neglected for simplicity). Thus, the target function to be minimized is

$$E = \sum_i E_i. \tag{4.3}$$

The following constraints are used to obtain feasible solutions. Each task must be scheduled within the deadline. Hence, the start time s_i of a task i must be at least zero and its end time $e_i = s_i + r_i$ is not larger than the deadline:

$$\forall i : s_i \geq 0 \text{ and } e_i \leq Dl. \tag{4.4}$$

Each task must be mapped exactly once in the schedule:

$$\forall i : \sum_j \sum_k x_{i,j,k} = 1. \tag{4.5}$$

To indicate whether task i precedes task i' or not, n^2 binary variables $y_{i,i'}$ are used with $y_{i,i'} = 1$ if and only if task i precedes task i' on a common PU. Then task i' can only start after task i, i.e.

$$\forall i, i' : s_{i'} \geq e_i - (1 - y_{i,i'}) \cdot C, \tag{4.6}$$

where C is a constant larger than Dl, e.g. $C = 2 \cdot Dl$. When both tasks are mapped onto the same PU and i precedes i', $y_{i,i'} = 1$ and the right-hand side of this constraint results in e_i, otherwise the right-hand side is smaller than zero.

A task i can not precede itself, thus

$$\forall i : y_{i,i} = 0. \tag{4.7}$$

Either two tasks i and i' are succeeding tasks on a common PU so that one of them precedes the other one, or they are mapped onto different PUs. However, both can not precede each other:

$$\forall i, i' : y_{i,i'} + y_{i',i} \leq 1. \tag{4.8}$$

Two tasks i and i' can only be succeeding tasks, when they are mapped onto the same PU, i.e.

$$\forall i, i' \neq i, j : y_{i,i'} + y_{i',i} \geq -1 + \sum_{k} (x_{i,j,k} + x_{i',j,k}). \tag{4.9}$$

The right-hand side either results in 1, when both tasks are on PU j and thus the right-hand sum is 2, or it results in a value less or equal to zero, i.e. one or both tasks are mapped onto another PU.

When two tasks i and i' are mapped onto different PUs, they can not be succeeding tasks, i.e. the variable $y_{i,i'}$ is then forced to zero:

$$\forall i, i' \neq i : y_{i,i'} \leq 2 - \sum_{j,j' \neq j} \sum_{k} (x_{i,j,k} + x_{i',j',k}). \tag{4.10}$$

To consider dependencies and communication times between tasks, the following constraint is necessary: If task i' is a successor task of task i, i.e. an edge exists between both tasks in the task graph, and both are mapped onto different PUs, then the communication time $c_{i,i'}$ between them must be considered and task i' can start after the communication. When the tasks are mapped onto the same PU, task i' can only start after task i:

$$\forall i, i' : \text{ edge } (i, i') : s_{i'} \geq e_i + (1 - y_{i,i'}) \cdot c_{i,i'}. \tag{4.11}$$

In this constraint $(1 - y_{i,i'})$ ensures that the communication time is only considered, when both tasks are not mapped onto the same PU, i.e. $yi, i' = 0$.

4.5.2 Fault Case

A failure of a PU can occur at any time during the execution of a schedule. To have a small finite number of failure points, only one failure per task is considered. The longest delay for broadcasting a failure is when it occurs at the beginning of a task, because the remaining PUs are informed about the failure by the missing commit message. In the fault case, optimal solutions for the already existing mapping can be calculated for each failure point. Then, the optimal schedules can be used to evaluate the fault case heuristics, as they use the already existing tasks (originals and duplicates) and only change the frequencies and the start times of tasks dynamically, but not the ordering or the mapping. As a general placement of duplicates cannot

be optimal for each fault case, they have to be placed in different ways depending on the failure position to get an optimal solution. Therefore, another approach is to re-schedule the tasks in case of a failure onto the remaining $p-1$ PUs, instead of running on p PUs. This can be done statically prior to execution for each failure position, i.e. for each original task that fails. Then, the execution time of a schedule is not influenced by the scheduling time and a corresponding runtime system can choose the optimal schedule according to the failure point.

Existing Mapping To find an optimal solution for the existing mapping, only the dependencies and communication times must be considered next to end times of tasks. Therefore, an explicit declaration of PUs is unnecessary in this case. To formulate an ILP the energy consumption for a task i can be expressed by:

$$E_i = P(f) \cdot \frac{w_i}{f} \tag{4.12}$$

where $P(f)$ is the power consumption at frequency f and w_i is the workload of task i. Now, an approximation close to an optimal solution can be obtained by using $n \cdot K$ binary variables $x_{i,k}$ for n tasks and K frequency levels f_k each with $x_{i,k} = 1$ if and only if task i is executed with a frequency level f_k. A frequency range between 0.1 and 2 with a step size of 0.05 is used. The energy consumption for a schedule can then be expressed by:

$$E = \sum_i \sum_k x_{i,k} \cdot w_i \cdot \frac{P(f)}{f_k}. \tag{4.13}$$

The end time e_i of an ending task i must be less than or equal to the deadline:

$$\forall i = \text{ending task} : e_i \leq Dl. \tag{4.14}$$

The end time e_i of an entry task i is at least the runtime of the task, i.e.

$$\forall i = \text{entry task} : e_i \geq \sum_k x_{i,k} \cdot \frac{w_i}{f_k}. \tag{4.15}$$

Only one frequency level can be used for each task, thus

$$\forall i : \sum_k x_{i,k} = 1. \tag{4.16}$$

A task i can only be started after all predecessor tasks i'. If a predecessor i' is mapped onto another PU, the communication time $ct_{i',i}$ can be directly included as a constant, otherwise it is set to zero. Then, the end time e_i of a task can be expressed by

$$\forall i : \forall i' \in pred(i) : e_i \geq \sum_k x_{i,k} \cdot \frac{w_i}{f_k} + ct_{i',i} + e_{i'}. \tag{4.17}$$

Note, that this constraint is set for every predecessor task explicitly to fix the mapping and ordering of the tasks, e.g. if a task has three predecessors, this constraint is included three times.

As only the time between the failure point and the deadline is considered in the ILP, all tasks that start before and finish after the failure are cut at the failure time and the remaining part after the failure is included as workload. For these tasks, the frequency level f_k is fixed by setting $x_{i,k} = 1$, because only one frequency per task is allowed.

Re-scheduling An optimal solution for the re-scheduling in case of a failure can be achieved by modifying the above explained mixed ILP for the fault-free case. For all tasks that start before the end of the failing task, the mapping, frequencies and start times and thus the x- and s-variables in the mixed ILP are fixed. As the duplicate of the failing task can only be started after its corresponding original task, the start time of the duplicate is bound by the end time of the failed original task. The faulty PU cannot be used after the failure and all remaining tasks cannot be mapped onto the faulty PU. This is considered by forcing the corresponding x-variables to zero. The MMI-Option, i.e. when a prolongation of a schedule is allowed, can be included by changing the deadline according to the given percentage value.

5 A Fault-tolerant and Energy-efficient Runtime System

RUPS (Runtime system for User Preferences-defined Schedules) is a scheduling tool for grids, clusters, multicore and manycore systems with features allowing the user to input various preferences in the schedule. Such preferences are for example: Performance, energy consumption or fault tolerance. Schedules are then created with the RUPS tool – optimized for the user-defined preference in question. RUPS consists of four main parts illustrated in Fig. 5.1.

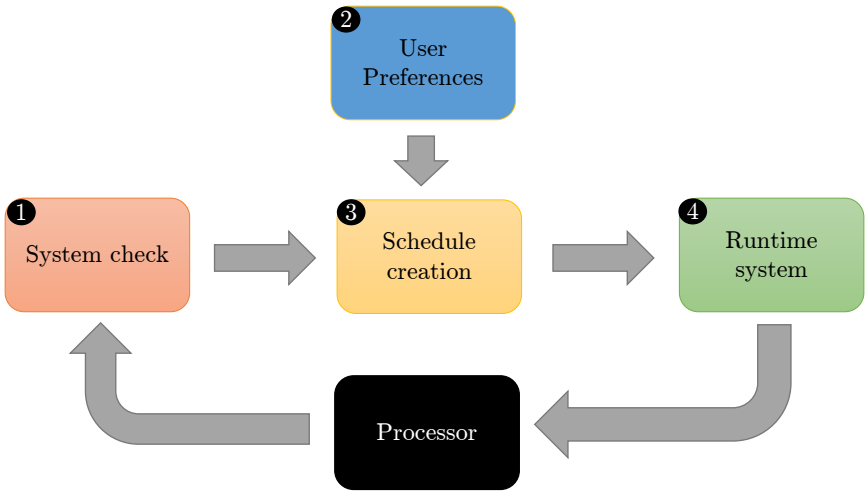

Figure 5.1: Overview of RUPS.

The processor details are extracted in Part 1 and passed to the scheduler (Part 3), which in turn optimizes the schedule based on the processor parameters and the user preferences (Part 2). Finally, the complete schedule is passed to the runtime system (Part 4), and scheduled on the parallel system. As the schedule creation and the

user preferences are already described in detail (see Chap. 4), in the following only the system check tool (Sect. 5.1), the runtime system (Sect. 5.2) and an appropriate power model for a real system (Sect. 5.3) are explained.

5.1 System Check Tool

At the first use of RUPS on a particular system, it has to be initialized once with the system check tool to adjust the power model for the processor used. This tool detects the number of processor cores. If a Turbo-Boost mode is activated, it calculates the corresponding turbo frequencies for a different number of cores. CPUFreq only indicates the activated Turbo-Boost mode but does not show the exact frequencies. Then, the tool measures the power consumption of the processor for different frequency/core settings. The number of settings depends on the number of cores C and the number of supported frequencies F. Each core is either considered to be in idle mode (inactive) or under full load (active) at a given frequency.

A binary representation is used to differ between active and inactive cores where digit d_i represents core c_i. If $d_i = 1$ the core is active otherwise it is inactive. Thus, there are in total $F \cdot 2^{|C|}$ cases. Tab. 5.1 shows the organization of the settings.

Table 5.1: Settings for a Processor with Four Cores and F Frequency Levels.

Freq-lvl	**cores** $c_3c_2c_1c_0$	**status**
1	0 0 0 0	All cores are inactive at frequency level 1
1	0 0 0 1	Only $core_0$ is active at frequency level 1
1	0 0 1 0	Only $core_1$ is active at frequency level 1
⋮	⋮	⋮
1	1 1 1 1	All cores are active at frequency level 1
⋮	⋮	⋮
F	0 0 0 0	All cores are inactive at frequency level F
F	0 0 0 1	Only $core_0$ is active at frequency level F
F	0 0 1 0	Only $core_1$ is active at frequency level F
⋮	⋮	⋮
F	1 1 1 1	All cores are active at frequency level F

The tool creates C pthreads, one for each core, and binds them to the different cores. To simulate an active core under full load, in total six micro benchmarks are used in a time-controlled while-loop, where each benchmark represents a different

class of instruction mix: ALU- (Arithmetic Logic Unit), FPU- (Floating Point Unit), SSE- (Streaming SIMD Extension), BP- (Branch Prediction), RAM-intensive and a mixed variant. In Tab. 5.2 the different benchmarks are described.

Table 5.2: Different Benchmark Settings.

Benchmark	**Description**
1. ALU-intensive	An integer variable a is initially set to 0. In each loop step the variable is incremented by 1.
2. FPU-intensive	Three double variables b, c, d are initially set to: $b = 1.5$, $c = 3.5$ and $d = 0$. In each loop step $d = b/c$, $b = b + 0.01$ and $c = c + 0.01$.
3. SSE-intensive	Realized with *intrinsic* functions that are built into and handled by the compiler to get processor-specific functionality. In each loop step the function $y = sin(x)/x$ is calculated with 4 float values in parallel.
4. BP-intensive	Two int variables k, sum are initially set to 0. In each loop step sum is incremented by 1 $if((k\&2) == 0)$, then k is incremented by 1.
5. RAM-intensive	Array[size 10^6], initially set to index. Array elements added up in the following order: $(7 \cdot index + rank)\ mod\ array\_size$.
6. Mixed	In each loop step, benchmarks 1 - 5 in sequence.

A barrier is used before and after the while-loop to synchronize the threads. Threads of inactive cores directly wait in the barrier after the while-loop. The main thread itself is used to measure the power consumption. The power consumption is measured for 10 *s (seconds)* with a sampling rate of 10 *ms (milliseconds)*. All cases are repeated five times to compensate high power values that could occur due to unexpected background processes. Between each case, all cores are set to the lowest frequency in idle mode for five seconds to reduce the rise in temperature of the processor and the influence on the power consumption. Then, the averaged results of the measure points for each case are used as values for the power model.

5.2 Runtime System

The runtime system is based on ULFM-MPI, a fault-tolerant extension of Open MPI (see Sect. 2.4.4). ULFM-MPI is used to support the error handling of MPI-processes in general. The execution of a schedule and the detection and handling of a failure during the execution is given by the runtime system. For each core, an MPI-process

is created that binds the process to a specific core and sets the userspace governor and the frequency of the core to the lowest possible[1]. Then, it reads the schedule and task graph information from files and creates a task queue that is sorted according to the starting times of the tasks to be executed. After a barrier to synchronize the work, a while-loop is executed as long as there are tasks in the queue. The loop is used for a polling mechanism that reacts and handles the communication, starts a task if possible, and aborts a task if necessary. Fig. 5.2 illustrates a short overview of the runtime system.

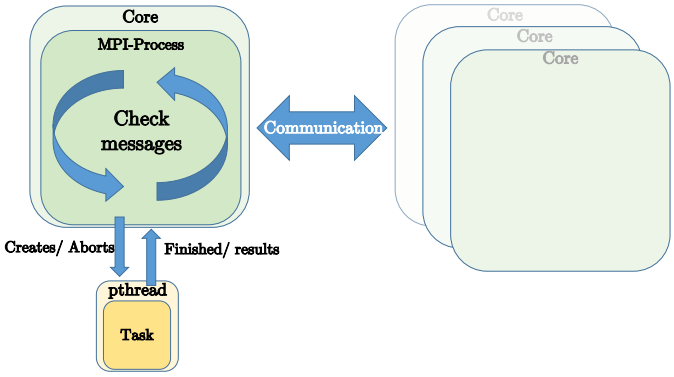

Figure 5.2: Overview of the Runtime System.

Within one loop round the following steps are done (see Lst. 5.1):

```
1  while(there is still a task in the queue){
2    sleep for 10 ms;
3    leftouttasks(...);
4    trytoreceivemessage(...);
5    if(there is no thread spawned){
6      checkmessages(...);
7      spawnthreadifpossible(..); //and scale the frequency up
8    }
9    trytoreceivemessage(...);
10   if(thread exists)
11     abortthreadifnecessary; // and scale the frequency down
12   if(thread is finished){
13     scale frequency down;
14     sendmessagesifnecessary;
15     take the next task in the queue;
16   }
17 }
```

Listing 5.1: Pseudo Code of the Runtime System.

[1]Hyper-threading is not considered in this runtime system and must, therefore, be switched off in the BIOS.

- The core sleeps for 10 ms to minimize the polling overhead (line 2).
- It is checked whether a task can be left out, e.g. a duplicate that could start next but that already received a commit message from the original task. In this case, it does not have to be executed (line 3).
- A message is tried to receive, that can mainly include either some results of another task or a commit message for a duplicate. In this version of the runtime system, just the message header is sent as results. The message header includes, next to others, the information about the task for which the message is destined, what kind of data will be sent, how much memory is needed for the data, the start time of the sending operation and the time that is needed for the transfer. The latter two are used to simulate the transfer. The communication is done on the sender side with a blocking `MPI_send` operation, on the receiver side with an unblocking `MPI_iProbe` to check if there is a message for the core. If so, a blocking `MPI_recv` operation is used to receive the message. Thus, the sender is blocked as long as the message is not sent, the receiver is blocked only if there is a message for it. Otherwise, it can continue with the remaining part of the while-loop (line 4).
- If no other task is running and if all results of the predecessor tasks are already received, the frequency is scaled up and the task is started by spawning a thread. To simulate an expensive calculation one of the six micro benchmarks from Sect. 5.1 is used for the computation time of the task. The task execution is separated from the communication process by a thread (see Fig. 5.2) so that the structure of a task is independent of the communication. Especially the runtime system lacks information about the task itself and about the functions that are done within the task. Thus, it could be impossible to interrupt the task execution at some points for trying to receive some messages. Furthermore, one thread is used for each task and not one thread for all tasks. The tasks on one core are executed one by one so that at each time just one thread or task is running. Another possibility would be to spawn only one thread for all task executions, but then a task cannot be aborted if necessary because some polling mechanism would be needed again that cannot be included in the task because of the missing knowledge of the task structure. Two flags are used to indicate whether a task is running or not and if a task has just finished (lines 5 - 8).

- After this the process once again tries to receive a message. This is necessary to avoid deadlocks that could appear if two cores try to send a message to each other (line 9).

- If a task is already running and if it is a duplicate, check if the duplicate has to be aborted because of a receiving commit message from the original task. If a duplicate should be aborted, the corresponding thread is canceled and the frequency is scaled down to the lowest possible (lines 10 - 11).

- Otherwise, it is checked if the thread has finished, the frequency is scaled down to the lowest possible and the results are sent to the successor tasks if necessary (lines 12 - 17).

After the while loop a second barrier is used to indicate that the schedule execution is finished. To implement the runtime system in an energy-efficient way, it is important that some MPI-operations like a blocking receive or a barrier use a busy waiting mechanism. If an MPI-process is running in those operations for a longer time, the corresponding core wastes energy because it is under full load for this time. Thus, those operations should be used carefully. Therefore, in this runtime system an `MPI_iProbe` is used before every blocking receive. Also the second barrier after the while loop is realized in this way.

The runtime system supports a testing mode, where one additional MPI-process is started next to the above sequence to measure the energy consumption with the help of the Intel RAPL feature (see Sect. 2.5.4). The measurement process measures the energy for the time of the whole while loop, because this is the time needed for the execution of the total schedule. The sample rate of the measurement is 10 ms. This process is not bound to a specific core like the other processes, so it can run on any core. There are some performance and energy overheads that occur by running the testing mode with the further process. But the influence can be neglected for two reasons: The energy overhead is included in the measurements of the system check that are done by a master thread. Thus, the energy values for the power model consider this overhead. And the performance overhead is less than one percent of a processor core. Therefore, the influence on the task execution is low.

Until now, the runtime system is described in general but not, how a failure of a PU (here a PU represents a processor core) can be handled. A failure is simulated by exiting an MPI-process just before the corresponding task is started, i.e. in the function `spawnthreadifpossible()`. The other processes are informed about the

failure by an error handler that is connected to the communicator and is involved if a message cannot be received. The `MPI_iProbe` operation checks messages from any source and with any tag. This means, a failure is nearly directly identified from all remaining MPI-processes. The time delay is less than 10 ms because messages are tried to be received two times per loop round.

5.3 Power Model

To predict the energy consumption for a schedule, an appropriate power model for the processor is necessary. In general, the power consumption can be subdivided into a static part, that is frequency-independent and a dynamic part, that depends both on the frequency and on the supply voltage.

$$P_{processor} = P_{static} + P_{dynamic} \tag{5.1}$$

The static power consumption consists of the idle power P_{idle} and a device specific constant s, that is only needed when the processor is under load.

$$P_{processor} = \begin{cases} P_{idle} + s + P_{dynamic} & \text{if under load,} \\ P_{idle} & \text{else.} \end{cases} \tag{5.2}$$

The dynamic power consumption is typically modeled as a cubic frequency function [7]. Additionally, the supply voltage and thus the dynamic power consumption depends on the load level of a core. As only fully loaded cores are considered or cores that are in idle mode (at the lowest frequency) the influence of a load level can be given by a parameter $w \in \{0, 1\}$. If a homogeneous multicore processor with n cores is assumed, a simple power model for the dynamic part can be given by the following equation, where a, b and β are device specific constants, i is the core index and $f_{cur,i}$ is the current frequency of core i:

$$P_{dynamic} = \sum_{i=0}^{n-1} w_i \cdot \beta \left({f_{cur,i}}^3 + a \cdot {f_{cur,i}}^2 + b \cdot f_{cur,i} \right) \tag{5.3}$$

Only if a core runs at a higher frequency under full load, the dynamic part of the power consumption for the processor is considered.

6 Experiments

6.1 Test Environment

6.1.1 Test Sets

To evaluate the heuristics and strategies presented in Chap. 4, a benchmark suite of 36,000 synthetic task graphs and schedules for small cases with up to 24 tasks and 36,000 task graphs for large cases with up to 250 tasks is used [84]. The task graphs are subdivided into several groups according to their properties. As the organization of the benchmark suite for small and large cases differs in the intervals of task count only, the structure is described for both together. One property is the size of the target system that is considered by the number of PUs. In total 5 groups exist with 2, 4, 8, 16 and 32 PUs and with 7,200 task graphs each. Every group is categorized by the number of tasks into 3 subgroups: 7 - 12, 13 - 18 and 19 - 24 tasks for the small cases and 25 - 99, 100 - 174 and 175 - 250 tasks for the large cases. In each interval 2,400 task graphs exist that are further subdivided by the edge density in low, average, high and random. Each of these groups contains 600 task graphs that are further differentiated by the edge length into short, average, long and random. The remaining 150 task graphs per group are finally subdivided by the edge and node weights into 5 groups with high node/high edge, high node/low edge, low node/high edge, low node/low edge and random node/random edge weights. In 4 of these groups are 25 task graphs each. The group of random node and edge weights consists of 50 task graphs. In general, the benchmark suite is organized as a flat file database, where the structure is given by the directory tree [84]. The principle organization is shown in Fig. 6.1.

Based on the task graphs for small cases, two schedule sets with performance optimized schedules are used that were generated by Hönig [84]. Both schedule sets comprise nearly 34,000 schedules each, because for some task graphs no optimized solution was found. The first schedule set, called *TB-Optimal (Test Benchmark)*, comprises performance optimal schedules. These schedules were generated with a

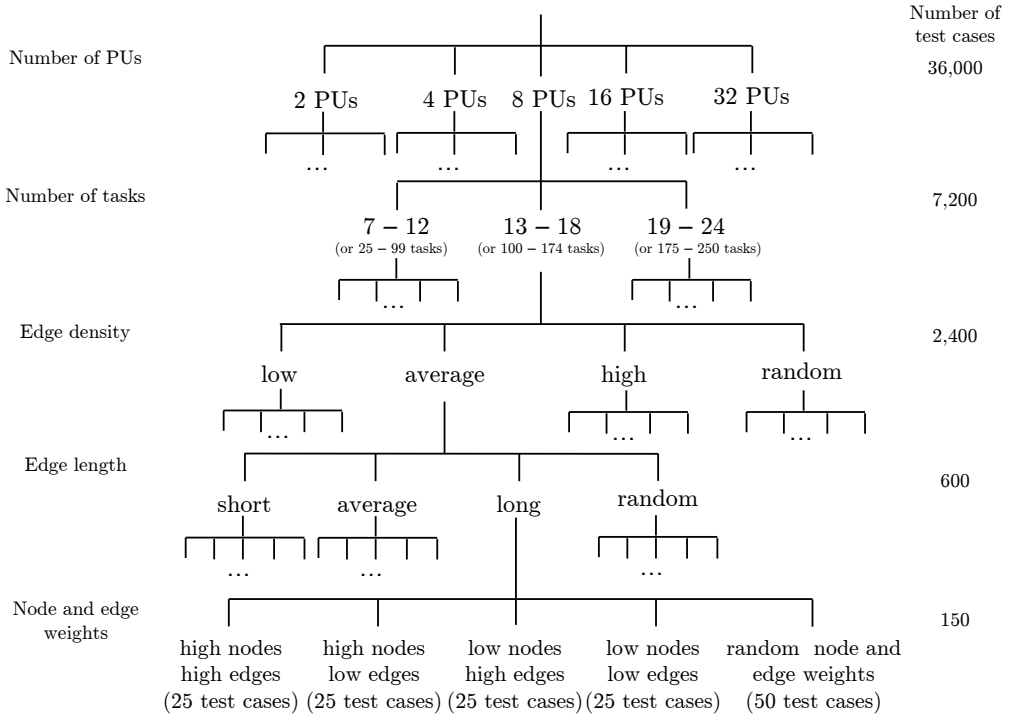

Figure 6.1: General Organization of the Benchmark Suite [44][84].

PDS-algorithm (*Pruned Depth-first Search*). To find optimal solutions in an acceptable time, the search space is reduced by pruning selected paths in the search tree. The second schedule set with performance optimized schedules, called *TB-ACO*, is based on the ACO-algorithm described in Sect. 2.3.1.

Next to the performance optimized schedule sets, non-optimal schedules were generated for both the small and large cases by using the simple list scheduler described in the UHPO-heuristic without the restriction to half of the PUs (see Sect. 4.2). The scheduler assigns tasks to PUs, where they can start their execution first. The schedule set for the small cases is called *TB-SLS (Simple List Scheduler)*, the set for the large cases is called *LTB-SLS (Large Test Benchmark)*.

All task graphs and schedules are given and generated in the *STG*-Format *(Standard Task Graph)* and *SSF (Standard Schedule Format)* described by Kasahara[1] [100] and by Hönig [84]. In this work, the resulting SSF-files are extended by two columns for the information about the used frequency levels or computation rates and for the task types (original, duplicate or dummy duplicate). Additionally, two example schedules from real world applications are used. The first one is a task

[1]Hironori Kashara is a Professor of the Advanced Computing Systems Laboratory from the Waseda University in Tokyo.

graph for robot control, i.e. for the calculation of Newton-Euler dynamic control for a 6-degrees-of-freedom Stanford manipulator [100]. The second one is a task graph for a sparse matrix solver of an electronic circuit simulation [100]. Both task graphs are depicted in Fig. 6.2.

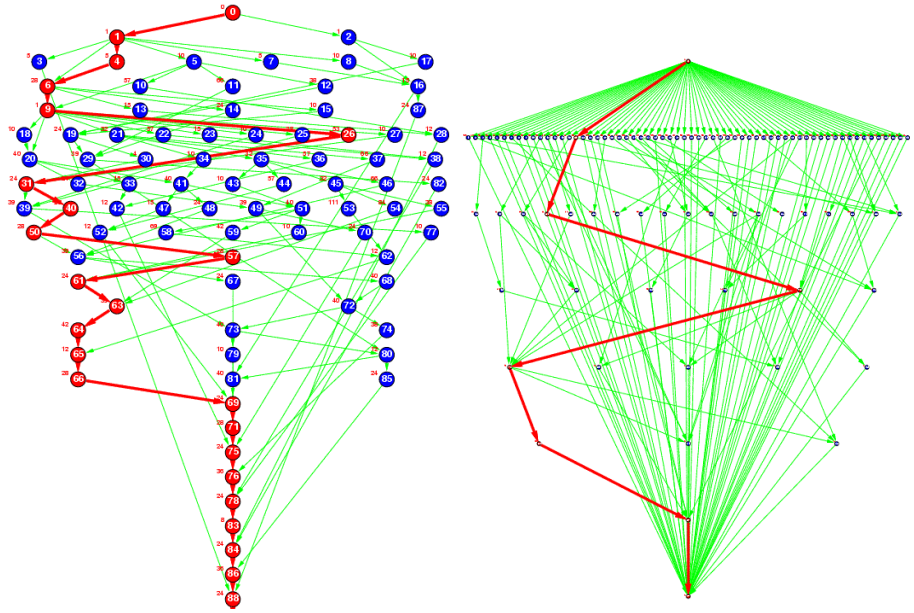

Figure 6.2: Taskgraphs of Real Applications, Robot Control (left) and Sparse Matrix Solver (right) [100].

6.1.2 Test Systems

The simulations are done on a Core-i7 Sandy-Bridge processor system with the operating system Windows 7. As compiler MinGW and for optimal solutions the solver CPLEX 12.7.1 are used. The system consists of 8 *GB (GigaByte)* RAM and 2 x 500 GB hard disk drives set up as *RAID* 1 *(Redundant Array of Independent Disks)* system.

The prototype runtime system presented in Chap. 5 has been tested on three different platforms:

1. Intel i7 3630qm Ivy-Bridge based laptop
2. Intel i5 4570 Haswell based desktop machine
3. Intel i5 E1620 Haswell based server machine

On all three platforms the operating system Linux Ubuntu 14.04 LTS is installed. Additionally, OpenMPI/ULFM-MPI, CPUFreq, and PAPI are used to execute the schedules, scale the frequencies and measure the power consumption. Further experiments are done based on the power model for these systems described in Chap. 5.

Next to the three systems that can be used for grid computing, also experiments are done based on a power model for Intel's SCC (*Single-chip Cloud Computer*) as an example manycore system. The SCC consists of 48 cores that are organized in 24 tiles with two cores each. The tiles are interconnected by an on-chip 6×4 mesh network that also connects the cores to four *memory controllers (MCs)*. For DVFS, the cores are grouped into 24 frequency islands, one for each tile and 6 voltage islands that include 4 tiles each. The memory controllers and the network are separate frequency islands [92]. The general structure is shown in Fig. 6.3.

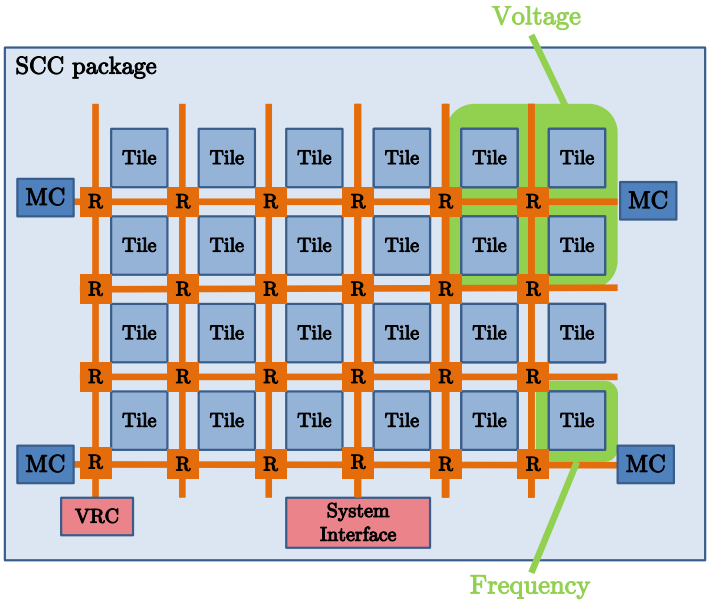

Figure 6.3: Structure of the Intel SCC [92].

6.2 Experiments with a Generalized Power Model

All strategies were evaluated with a generalized power model $P(f) = c + f^3$ (see Sect. 2.5.2) by simulating the execution of the schedules in the fault-free and fault cases with the integrated simulator. For simplicity, the static power consumption is left out and thus $c = 0$. As no frequencies are considered in the classical scheduling, the initial schedules in the test sets are assumed to be executed with a normalized frequency $f_{normal} = 1$. In these schedules, the power consumption of idle times

and tasks is assumed to be equal. This is a simplification as in a real system the power consumption of a fully loaded PU is higher than the power consumption of an idle PU at the same frequency. For the strategies S3 to S9 a continuous frequency scaling is assumed with a frequency range between $f_{lowest} = 0.1$ and $f_{highest} = 2$. The idle frequency in these strategies is set to f_{lowest}. In the strategies S1 and S2 the idle frequency is set to 1, as the strategies do not use any frequency scaling. As all frequencies are relative values, no units are given for the frequencies and for the energy consumption.

In the remainder of this section, firstly the strategies for the fault-free case are evaluated and analyzed separately in Sects. 6.2.1 to 6.2.4. After a comparison of all strategies for the fault-free case (S1 to S6) in Sect. 6.2.5, investigations of the strategies for the fault case (S7 to S9) are done in Sects. 6.2.6 and 6.2.7.

6.2.1 Strategies S1 & S2

The strategies S1 and S2 from the previous work (see Sect. 4.2.1) are analyzed together, as in both strategies no frequency scaling is considered and all tasks and idle times are executed with the same frequency 1. The strategies were evaluated with the test set TB-Optimal. Fig 6.4 depicts the averaged results over all schedules. For all schedules the energy consumption for the fault-free case and for the fault case (averaged over all failure points) are calculated. Also the performance in the fault-free case and the performance in case of a failure (rated as fault tolerance) is given. To be independent from the makespan of the schedules, the overhead results are given as relative values to the corresponding values of the original schedules without any changes. Therefore, in all figures of this type a negative overhead represents an improvement, a positive overhead represents an impairment.

We start with the energy consumption in the fault-free case. In both strategies, the energy overhead is expected to be 0, because DDs are placed with runtime 0 and thus with energy consumption 0 and the energy consumption of Ds is equal to the energy consumption of corresponding idle times. In case of a failure, in contrast, the energy consumption for both strategies should be lower than for the fault-free case, because the efficiency of the used PUs is lower than 100% as dependencies and communication times must be considered. Thus, the energy consumption is expected to be improved in the fault case (i.e. a negative energy overhead is expected), because then less idle times exist as the remaining tasks after a failure of a schedule are executed on $p - 1$ PUs. In Fig. 6.4, the energy consumption in strategies S1 and

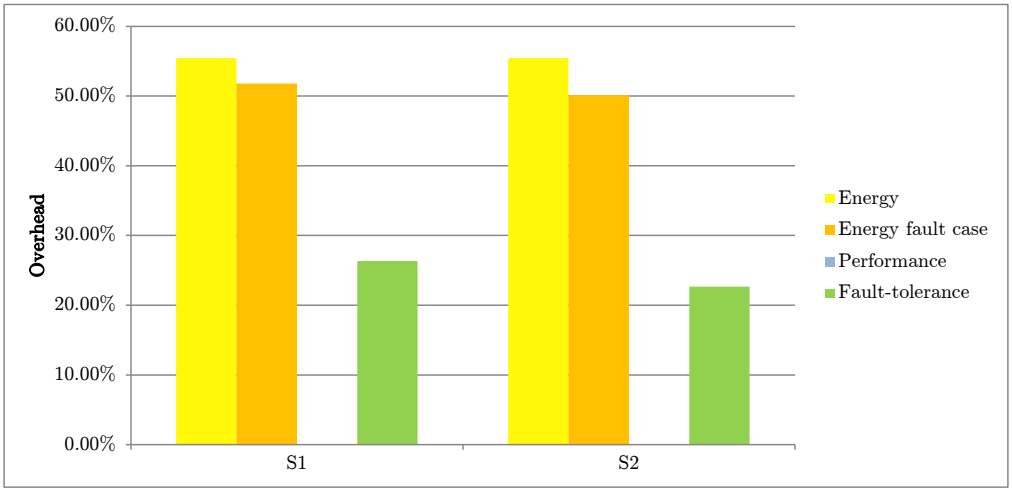

Figure 6.4: Overheads of Strategies S1 & S2.

S2 is significantly increased by 55.43% in the fault-free case (yellow bars). In the fault case (orange bars) the energy increase is 51.79% for strategy S1, and 50.12% for strategy S2.

This fact results from the mapping of the Ds and DDs in both strategies. As in the previous work the focus is put on the performance for both the fault-free and the fault case and the energy is neglected, the best solution is to place duplicates not only onto PUs with original tasks but also onto additional PUs that are already available in the initial schedules (but not in use). Especially for schedules with a high number of PUs, i.e. for 8, 16 and 32 PUs usually several PUs are left free in the TB-Optimal test set (with only up to 25 tasks per schedule). As unused PUs are considered to be in a sleep mode with a very low neglectable energy consumption or to be switched off totally, the energy is significantly increased by placing duplicates onto these PUs.

In Fig 6.5 the results are shown, when unused PUs are not considered for placing duplicates. As expected, no energy overhead in a fault-free case exists and an energy improvement in case of a failure.

The performance overhead in the fault-free case is as expected 0% (and therefore no blue bars exist). The performance overhead for the fault case in Fig. 6.4 is 26.36% in strategy S1 and 22.65% in S2. Thus, the use of Ds already in the fault-free case (S2) leads to a nearly 4% better performance in case of a failure. Compared to the results in Fig. 6.5 where the performance overhead in the fault case is at 27.83% (S1) and 25.88% (S2), the performance is improved by up to 3% when using free PUs for the mapping of duplicates (but with the cost of additional 50% energy).

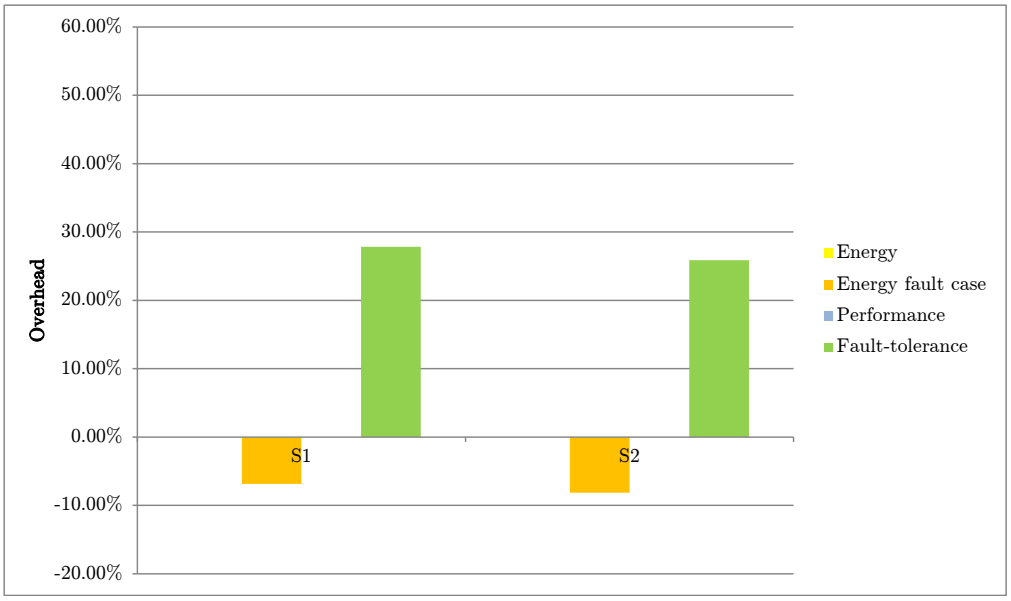

Figure 6.5: Overheads of Strategies S1 & S2 without Using Free PUs.

Therefore, with the assumptions made in the beginning, duplicates should better be placed onto PUs that were already in use in the initial schedules when no frequency scaling is possible.

6.2.2 Strategies S3 & S4

The strategies S3 and S4 are extended versions of the strategies S1 and S2 where energy efficiency is included by the BER-heuristic (see Sect. 4.4). Additionally, the idle frequency that is used for the gaps within the schedules is set to the lowest possible frequency f_{lowest}. In Fig. 6.6 the results of strategies S3 and S4 for the TB-Optimal test set are shown.

In strategy S3, the energy consumption can be improved by around 40% (40.74% in the fault-free case and 39.52% in the fault case) compared to the original schedules. Since the energy consumption for idle times is very low compared to the energy consumption of tasks[2], the idle times have only a small influence on the total energy consumption of a schedule. Therefore, the effects on the energy are also negligible by the use of free processors for the placement of duplicates in the fault-free case. In contrast, the slightly higher energy consumption in case of a failure compared to the fault-free case results from a non-optimal placement of duplicates that leads to more and longer gaps after the failure.

[2]Now the idle frequency is set to 0.1.

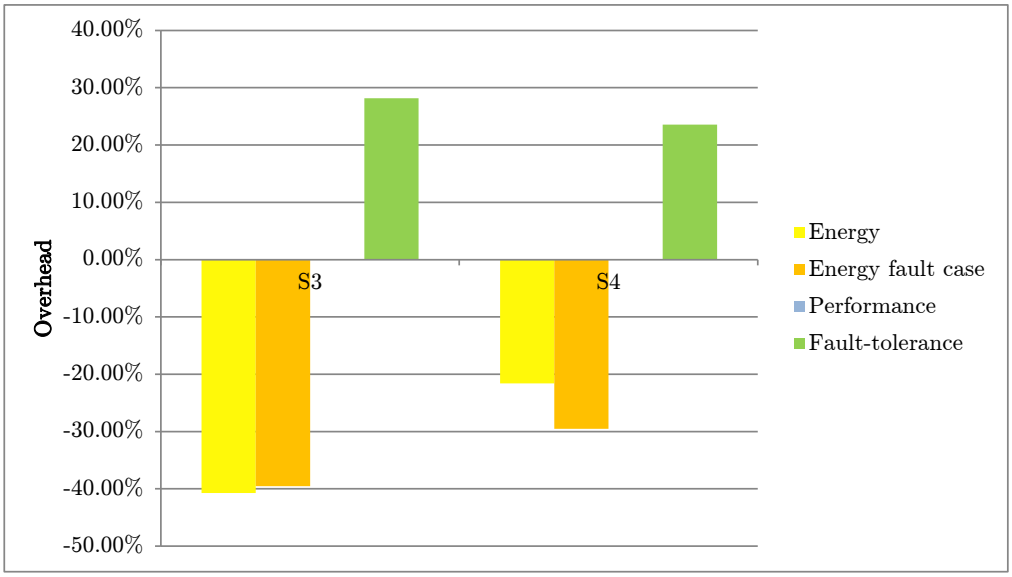

Figure 6.6: Overheads of Strategies S3 & S4.

The performance overhead in case of a failure is at 28.16% and therefore slightly increased compared to S1 from the previous work (26.36%). This performance overhead increase of further 2% results from using lower frequencies for original tasks and thus from delayed starting duplicates that influence the makespan only in case of a failure.

The energy improvements of 21.16% for the fault-free case and 29.52% for the fault case in strategy S4 are lower than in strategy S3 because in S4 several Ds are partly executed but then not longer needed and aborted (in both the fault-free and fault case). In contrast, as Ds start earlier than DDs, the performance overhead of 23.56% in S4 is improved compared to S3 (28.16%) in the fault case.

Energy distribution and Scaling In the experiments above, no differentiation between energy improvements resulting from the low idle frequency and from the BER-heuristic in S3 and S4 were made. Also the influence of different task mappings and a different number of tasks was not considered. In order to obtain the energy distribution and the scaling behavior, both strategies were also evaluated with the test sets TB-ACO, TB-SLS and LTB-SLS. In Fig. 6.7, the results of the energy improvements for the different test sets are depicted, where the light parts of the bars represent the improvements from the low idle frequency and the dark parts the improvements from the BER-heuristic.

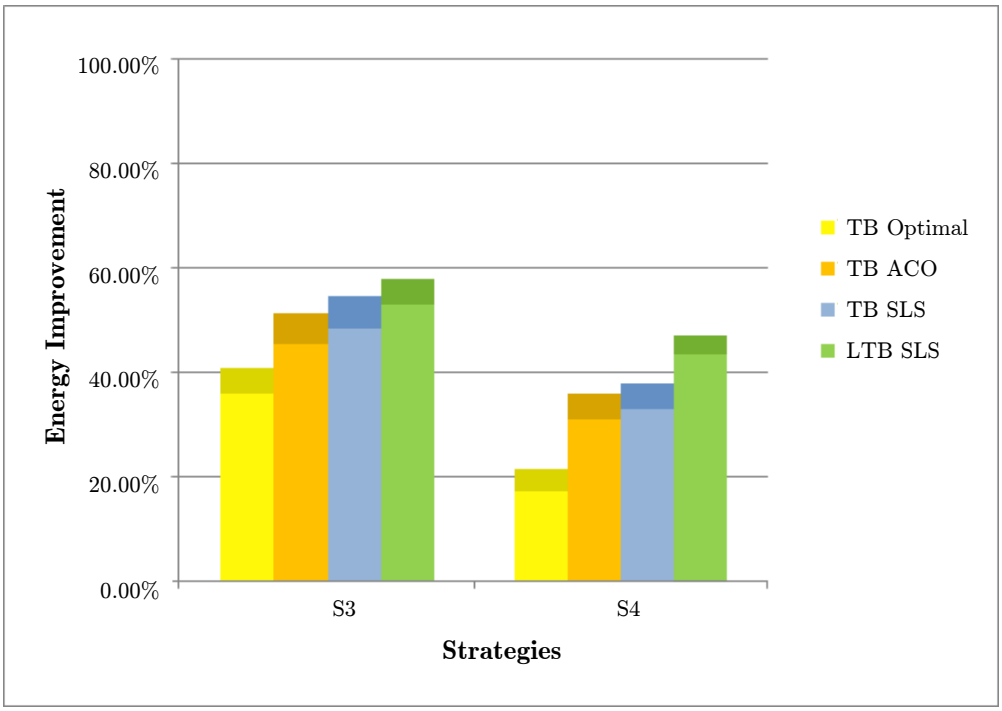

Figure 6.7: Energy Improvements for Different Test Sets.

As expected, the energy improvements for S3 with 40% to 58% are significantly higher than for S4 with 21% to 46%, because in S4 Ds are placed next to DDs that lead to less idle times and possibilities to scale down the frequencies for tasks. Most of the energy improvements with up to 53% (LTB-SLS) result from using a low idle frequency. By using the BER-heuristic, additional 4% to 6% can be saved.

Optimal vs. Non-optimal Solutions As in strategy S3 the focus is mainly put on the energy, a comparison to energy optimal solutions is important to analyze the quality of this strategy. Therefore, 200 random task graphs (and schedules from the TB-Optimal test set) were used to calculate the energy improvements for different optimal and non-optimal solutions. In Tab. 6.1 all approaches are shown.

Table 6.1: Different Approaches to Integrate Frequency Scaling into Scheduling.

	Classical Scheduling	Frequency Scaling
1.	combined optimal	
2.	optimal	optimal
3.	optimal	non-optimal (S3)
4.	non-optimal (SLS)	non-optimal (S3)

Firstly, energy optimal schedules are generated by combining the scheduling of the tasks with frequency scaling in one step. The schedules are calculated by the mixed

ILP for the re-scheduling variant described in Sect. 4.5. As deadline the makespan of each performance optimal schedule is assumed. Therefore, this solution is the most energy-efficient one without increasing the makespan. Secondly, energy optimal schedules are generated by separating the classical scheduling from the frequency scaling. In this variant, the mapping of the already existing performance optimal schedules is used and in a second step the frequencies for all tasks are optimized by the mixed ILP for the use existing mapping variant described in Sect. 4.5. As in strategy S3 the scheduling and frequency scaling is also separated, this solution directly demonstrates the potential in an optimal case. Thirdly, strategy S3 is used with the performance optimal schedules. Thus, in this variant a heuristic is used to scale the frequencies of tasks instead of an optimal scaling. Finally, also a solution is calculated, where the classical scheduling is done by the simple list scheduler described in the beginning of this chapter. To make this solution comparable to the others, the resulting makespan of a schedule must be adapted by increasing the base frequency for the whole schedule. Therefore, after the scheduling is done, an optimal base frequency for all tasks (and idle times) is calculated so that the tasks are speeded up and the makespan is equal to the performance optimal makespan from the TB-Optimal test set. In some cases, a solution can be infeasible when a base frequency is required that is higher than the highest supported frequency of 2. Then, strategy S3 is used again to optimize the frequency for each task. Therefore, this solution shows the results when using a non-performance-optimal scheduling and a heuristic for the frequency scaling. The generic approach to use a non-optimal classical scheduling and an optimal frequency scaling is left out, because it is an untypical and unrealistic scenario. But the energy improvement for this case is expected to be higher than for a non-optimal scheduling with S3 and lower than for an optimal scheduling with S3. In Fig. 6.8 the energy improvements relative to the corresponding original schedules for all 4 approaches are depicted.

As expected, the highest energy improvement with 48.95% can be reached by using the combined optimal approach. The separated optimal approach leads to 45.92% energy improvement. Strategy S3 with an optimal classical scheduling is only slightly lower (42.61%) than the separated optimal solution. The improvement drops down to 30.10% for strategy S3 with a non-optimal classical scheduling. In this approach, the higher base frequency leads to a significant higher energy consumption and thus to a significant lower energy improvement. For the 200 tested task graphs and schedules, one schedule was infeasible in this approach and therefore not used. As in this experiment the quality of strategy S3 with all its components is rated, the

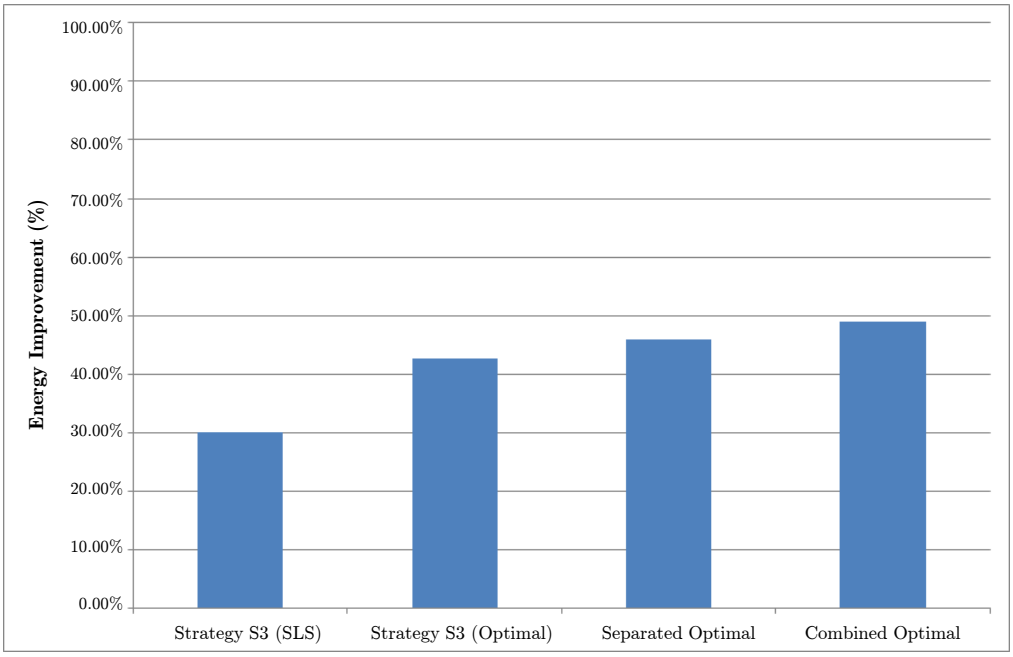

Figure 6.8: Energy Improvement in a Fault-free Case (Optimal vs. Non-optimal).

overall result is good. Compared to the optimal approaches, strategy S3 leads with a performance-optimal scheduling to a small improvement decrease of less than 7%.

When only focusing on the BER-heuristic part in S3 that leads only to a very small improvement of up to additional 6% compared to the improvement of a low idle frequency (see Fig. 6.7), the quality of the BER-heuristic itself is worse. We assume that 6% from the total improvement of 42.61% in S3 result from the BER-heuristic, i.e. from scaling down the frequencies for tasks, the improvement from using a low idle frequency results in 36.61%. When we consider this 36.61% improvement from a low idle frequency also for the optimal approaches as a rough estimation[3], the separated optimal variant leads to an improvement of $45.92\% - 36.61\% = 9.31\%$ by scaling the frequencies for tasks. The combined approach, in contrast, results in $48.95\% - 36.61\% = 12.34\%$ energy improvement for the task frequency scaling. Therefore, the improvement in the combined approach is twice as good as in the BER-heuristic and also the separated optimal approach leads to 55.16% better results when only considering the frequency scaling for tasks.

[3]The higher the total energy improvement is, the lower are the frequencies for the tasks and thus the less idle times exist. Therefore, the improvement of using a low idle frequency is getting smaller.

6.2.3 Strategy S5

In strategy S5, the focus is put on the fault tolerance. In Fig. 6.9 the overhead results for the TB-Optimal test set are shown.

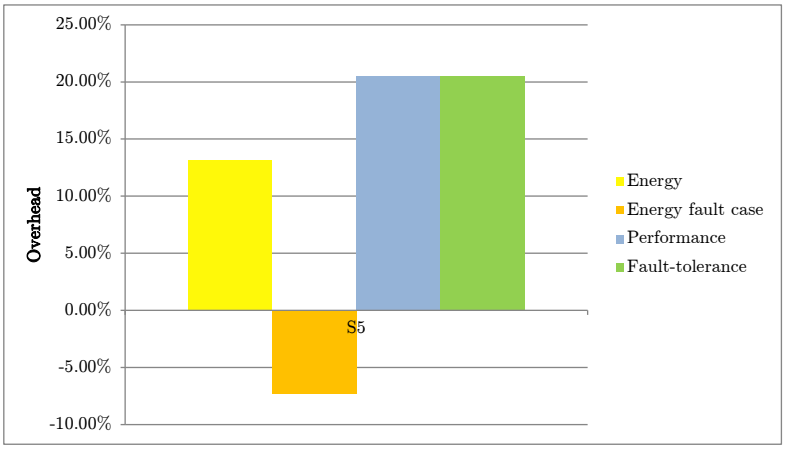

Figure 6.9: Overheads of Strategy S5.

The energy in the fault-free case is increased by 13.14%, because in this strategy all original tasks are scheduled to half of the available PUs. Then, the whole schedule is copied to the other half, i.e. all tasks are executed twice. Therefore, the additional energy invest for the duplicates cannot be compensated by using the BER-heuristic or a low idle frequency. In case of a failure, the improvements of using the BER-heuristic are visible, because then all unnecessary duplicates are deleted and only the original number of tasks is executed. As the duplicates are placed onto unused PUs, gaps between original tasks are not interrupted by duplicates so that the tasks (originals and duplicates) can be slowed down more. Therefore, the energy improvement results in 7.29%. The performance overhead of 20.50% results from using the simple list scheduler to map the tasks onto the PUs, i.e. from a non-optimal mapping of the tasks and from using fewer PUs for the original tasks where necessary. Thus, the performance is expected to get better, the higher the number of PUs is.

To see the influence of a different number of available PUs, the results in Fig. 6.10 are separated by the number of PUs. For two PUs, the energy overhead in the fault-free case is significantly increased by 61.91% because all original tasks are executed in sequence on one PU and all duplicates on the other PU. This leads also to an energy overhead in case of a failure of 17.80%. The execution of the schedule cannot be speeded up by using two PUs and running original tasks in parallel. Thus,

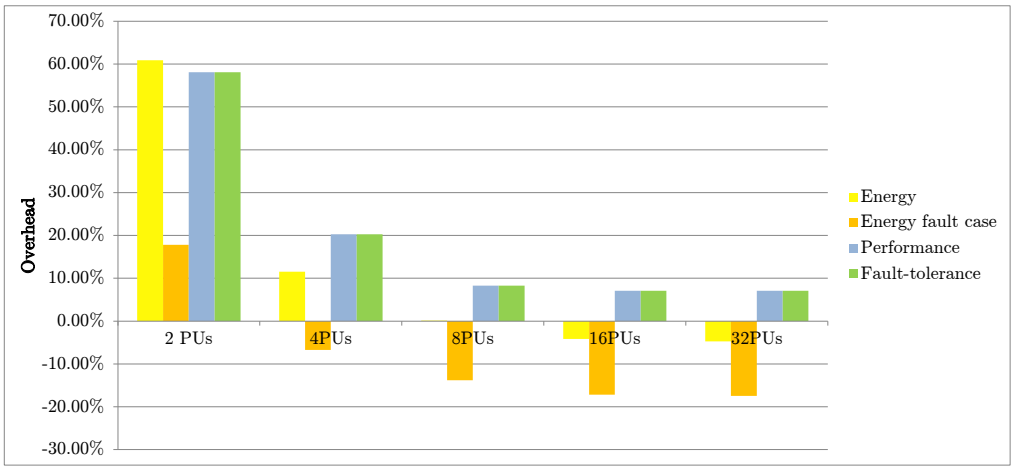

Figure 6.10: Overheads of Strategy S5 for a Different Number of PUs.

the makespan of the schedule, i.e. the performance of the schedule is significantly increased by 58.12% in both the fault-free and fault case.

When more than two PUs are available, the energy consumption for both the fault-free and fault case is getting smaller. Finally, for a high number of PUs an energy improvement of 4.73% (fault-free case) and 17.47% (fault case) is achieved compared to the original schedules. Especially for 16 and 32 PUs the results differ only slightly. This indicates that already for 16 PUs most of the PUs are not needed for the execution of the original schedules. Then, the performance overhead is only 7.08%.

6.2.4 Strategy S6

In strategy S6, the base frequency for the whole schedule is changed to improve the energy consumption. With the generalized power model $P(f) = f^3$ the lowest frequency leads to the lowest energy consumption. Therefore, as base frequency f_{lowest} is used. In this case, no additional improvements can be achieved by using the BER-heuristic. Thus, no differentiation for the insert order SED or SDE exists, as no further slowdown of tasks is possible. In Fig. 6.11, the overhead results for the TB-Optimal test set are depicted.

The energy is improved by 98.50% in the fault-free case and by 98.62% in case of a failure. In contrast, the performance overhead in both cases is significantly increased (853.46% in the fault-free case and 1,035.80% in the fault case) and therefore disproportional high in comparison to the energy improvements. This strategy represents a corner case for the energy improvement.

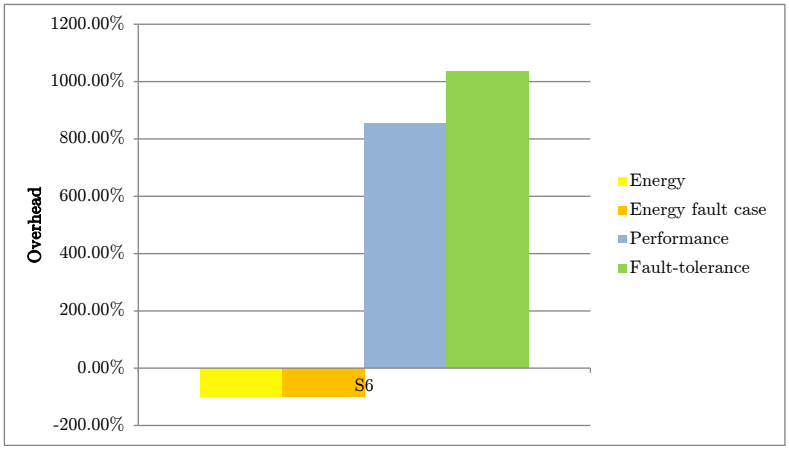

Figure 6.11: Overheads of Strategy S6.

6.2.5 Comparison of Strategies for the Fault-free Case

In Fig 6.12, the results for all strategies for the fault-free case (S1 to S6) are shown.

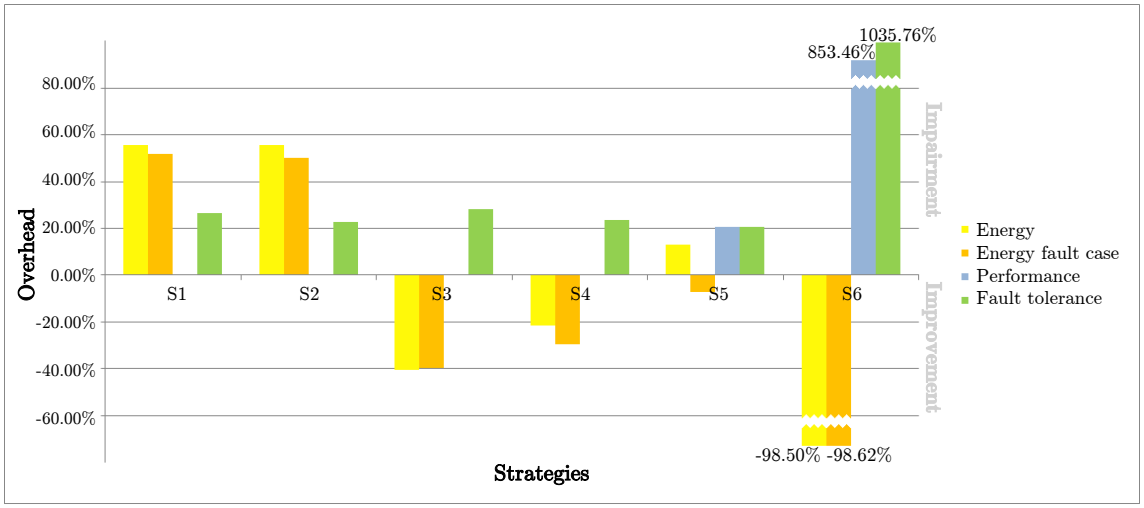

Figure 6.12: Overheads of Different Strategies.

In all strategies, where frequency scaling is used (S3 to S6), the energy consumption can be improved. As expected, strategy S6 leads to the highest energy improvements but also to the highest performance overheads in both the fault-free and fault case. Note that the performance trends in this strategy can be totally different for another power model as discussed in the next section with a power model of a real system. Strategy S3 represents a good solution to improve the energy without decreasing the performance in a fault-free case, as the energy improvements are high, no performance loss in the fault-free case exists and the performance overhead

in case of a failure is only slightly higher compared to the other strategies S1 to S5. The best performance in case of a failure (fault-tolerance) is achieved in strategy S5 where the duplicates are executed completely in the fault-free case. In contrast, the duplicates in all other strategies are executed only partly. But in this strategy, a performance loss already exists in the fault-free case, as all tasks are scheduled with the simple list scheduler and also only half of the PUs are used for originals. Strategy S4 can be used as an alternative for S5 as well as for S3, because the energy and performance results are between the results of S3 and S5. Thus, in strategy S4 a moderate energy improvement exists but also a low performance overhead in case of a failure.

6.2.6 Strategy S7

In strategy S7, the focus is put on the performance in the fault case. The strategy is evaluated with the test sets TB-Optimal, TB-ACO and TB-SLS. As in this strategy only DDs are used, the performance overhead in case of a failure is compared to strategy S3. Therefore, the focus changes form the energy in the fault-free case to the performance in case of a failure. In Fig. 6.13 the averaged results are depicted.

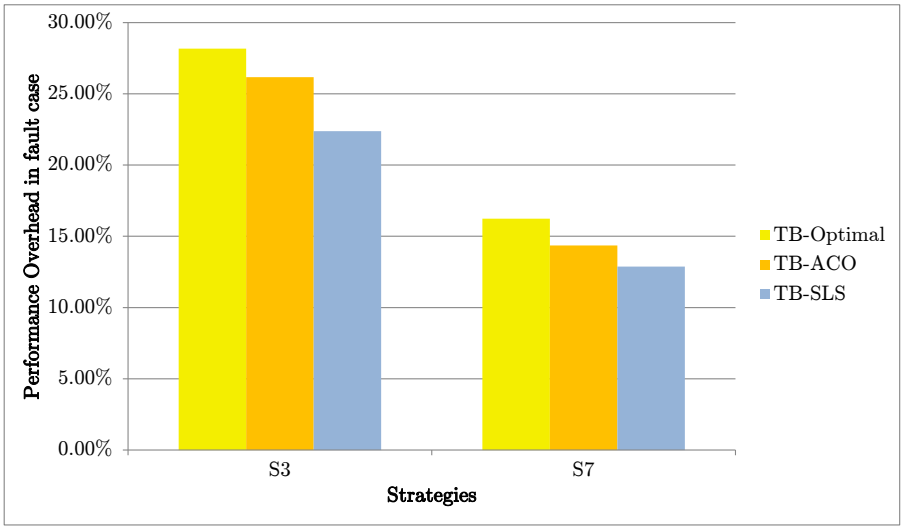

Figure 6.13: Performance Overheads in the Fault Case.

The performance overhead in case of a failure for strategy S3 is in a range of 22.38% (for TB-SLS) to 28.16% (for TB-Optimal). When using the EP-heuristic in the fault case, i.e. strategy S7, the performance overhead is decreased to 12.88% (for TB-SLS) and 16.24% (for TB-Optimal). This is an improvement of around 42%. Thus, with strategy S7 the performance is significantly increased in the fault case.

6.2.7 Strategies S8 & S9

The results for strategies S8 and S9 are discussed together, as both focus on improving the energy in the fault case. We assume to have a fixed deadline Dl until the schedule execution must finish. When Dl is set to the makespan of the schedule in a fault-free case, a solution for the fault case is often infeasible. For example, if a failure occurs at the latest task in the schedule, the corresponding duplicate cannot be started before the end of this task, i.e. the duplicate starts at Dl and can thus only be finished after Dl. For other cases, a feasible solution is impossible, where the frequency for a task must be set higher than the highest supported frequency to meet the deadline. To analyze the quality of strategies S8 and S9, i.e. the energy investment spent for a specific deadline, 187 random task graphs (and schedules) with 7 to 12 tasks for 2, 4, 8, and 16 PUs from the TB-Optimal test set are used. The relative energy increase, i.e. relative to the fault-free case for different deadline increases (5%, 10%, 15%, 20% and 25%) is calculated. In a first step, an energy optimal schedule is calculated with the ILP described in Sect. 4.5 for the fault-free case, where the makespan of a performance optimal schedule is used as deadline. In this way, the initial assumptions for both strategies are identical and the energy overhead in case of a failure is separated from effects due to scheduling and frequency scaling heuristics [56]. In a second step, an energy optimal schedule for each failure point is calculated with the help of the ILP for the re-scheduling variant presented in Sect. 4.5. Finally, strategies S8 and S9 are used and compared to the results of the optimal solution.

In Fig. 6.14 the results for the optimal variant and both strategies are depicted, where for each deadline increase the number of cases are shown that lead to different energy increases. We start with a low value of deadline increase (5%). The energy increase of the optimal variant (upper case) is mostly between 0% and 110%. Some cases result in a 10% lower energy consumption compared to the corresponding original schedule, as in these cases the frequencies for more tasks can be scaled down in the fault case, because of the additional time of 5%. When increasing the deadline further by 10% up to 25% the energy increase becomes lower and for more schedules the energy can be improved compared to the fault-free case (up to 40%). In strategy S8 (middle case), where the LFR-heuristic is used, the number of cases is significantly decreased and the distribution is shifted slightly to the right of the figure. As already described in Sect. 4.3, the LFR-heuristic starts with very low frequencies for tasks that result in high frequencies in the end (often higher than the

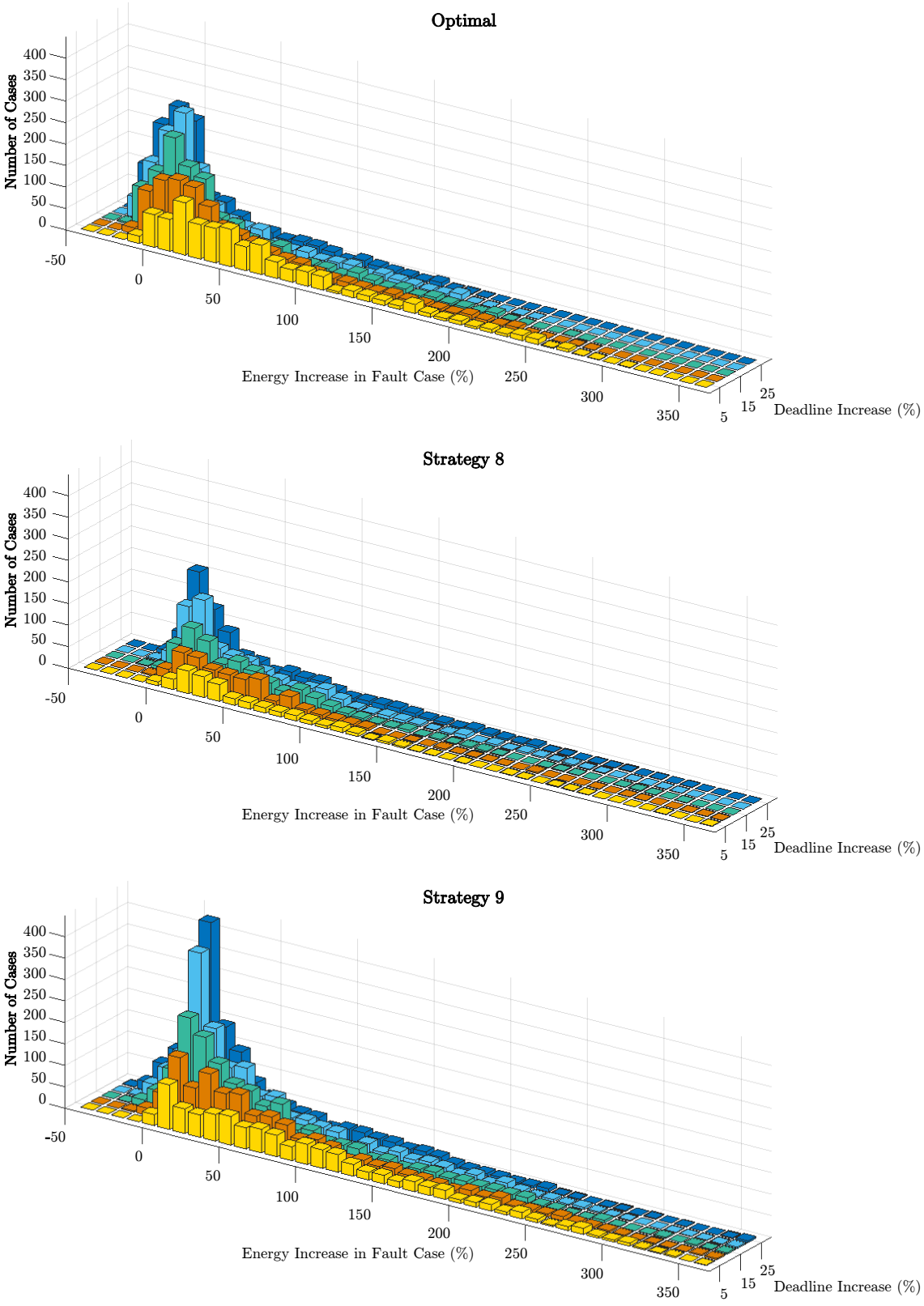

Figure 6.14: Distribution of Relative Energy Increase for Different Values of Deadline Increase.

highest supported frequency), so that in many cases the deadline cannot be met. Additionally, the time for the communication between dependent tasks is ignored when calculating a new frequency. Therefore, the remaining time to execute all tasks is much shorter than the time interval used by the LFR-heuristic. In strategy S9 (bottom case) that consists of the CP-heuristic, the distribution of the solutions is concentrated at 10% energy increase in all cases of deadline increase. Only a smaller number of cases leads to a higher or lower energy increase. In total the cases are more spread than in the optimal variant, mainly between 0% and 200% for the lowest deadline increase. Also for the other values of deadline increase strategy S9 leads to a higher energy increase than the optimal variant or strategy S8.

The distribution of the relative energy increase for the different values of deadline increase for all strategies and the optimal variant is a *GEV* (*Generalized Extreme Value*) distribution. The GEV distribution is a composition of three sub families, the Gumbel, Fréchet and Weibull distribution. Each GEV distribution is associated with one of these sub families depending on the shape parameter ξ [152].

In Figs. 6.15, 6.16 and 6.17 the density and probability of the data for the optimal variant and for strategies S8 and S9 are shown for different values of deadline increase, i.e. 5%, 10%, 15%, 20% and 25% from top to bottom. Also the fitted GEV curves are depicted, where the used parameters $GEV(\xi, \sigma, \mu)$ for shape ξ, scale σ and location μ are shown within the figures. As in all figures the shape $\xi > 0$, the GEV distributions for these experiments correspond to Fréchet distributions.

For all values of deadline increase, the fitted GEV distribution curve differs only slightly from the results in both the density and probability of the data. Only for a high energy increase the probability results are higher than expected.

Another important property to rate the quality of the strategies is the number of feasible solutions that are found by the strategies. As already seen, strategy S8 (LFR) leads to a significantly lower number of feasible solutions compared to S9 and the optimal variant.

In Fig. 6.18, the number of feasible solutions is shown for the energy optimal schedules and for the schedules that are generated by strategies S8 and S9. In total 26,295 schedules were calculated, i.e. 8,765 for each strategy and for the optimal variant. For every value of deadline increase 1,753 schedules are possible when all solutions are feasible. As expected, the optimal variant (yellow bars) finds the highest number of feasible solutions. Starting from a low value of deadline increase (5%), 1,259 schedules were found, i.e. 494 infeasible solutions. For a higher value of deadline increase, the number of feasible schedules is increased up to 1,741 schedules

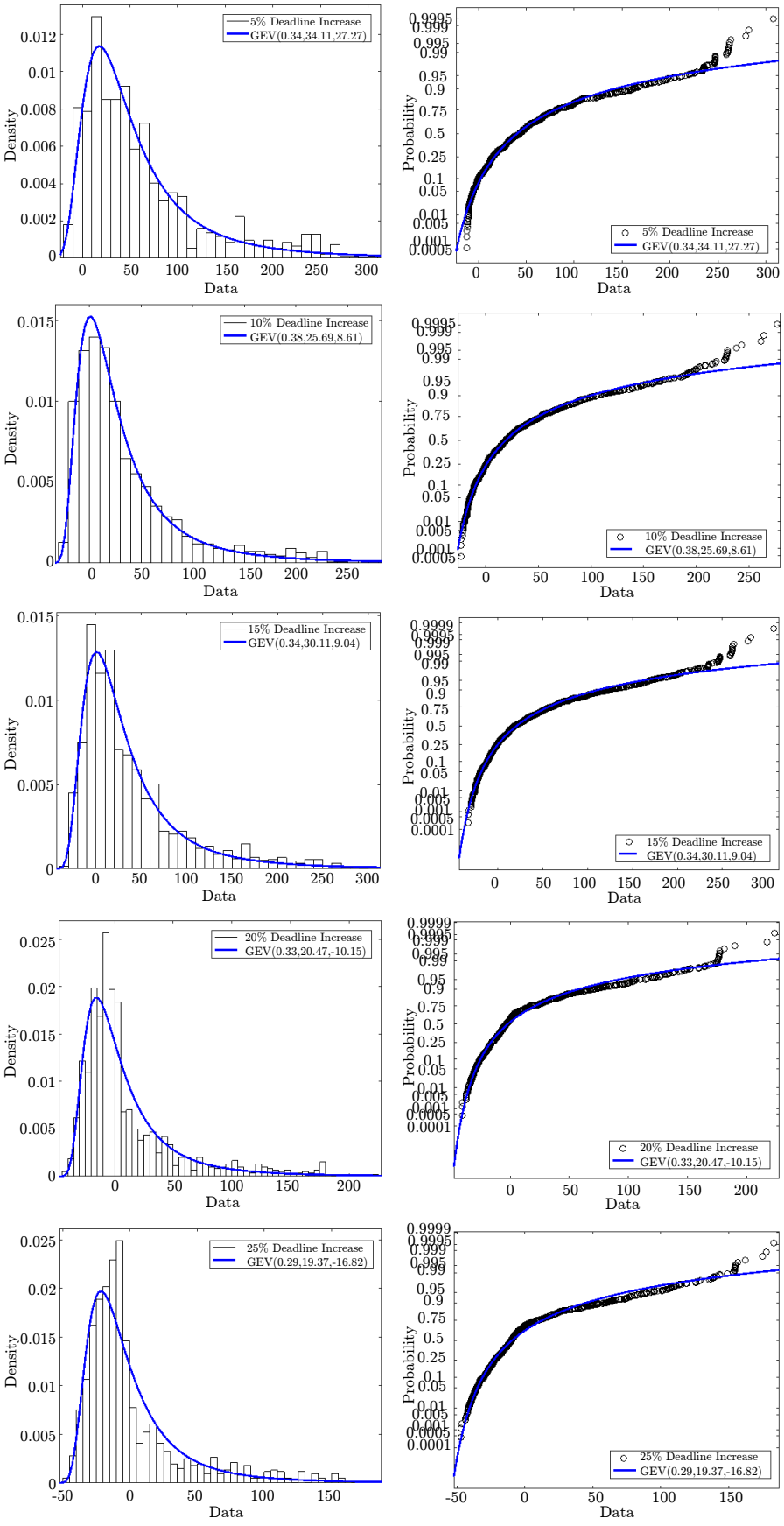

Figure 6.15: Density and Probability of GEV Distribution for Optimal Solutions.

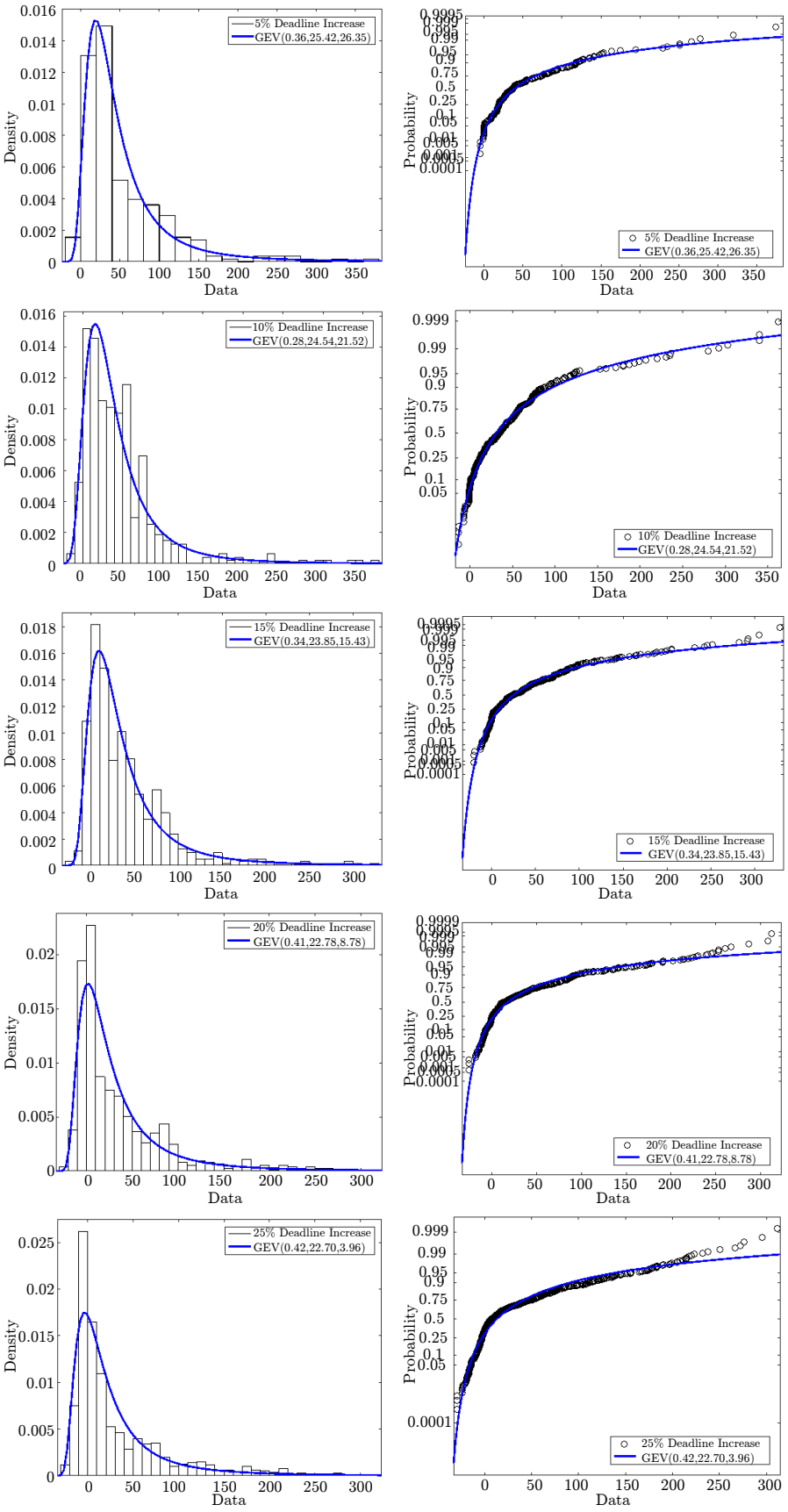

Figure 6.16: Density and Probability of GEV Distribution for Strategy S8.

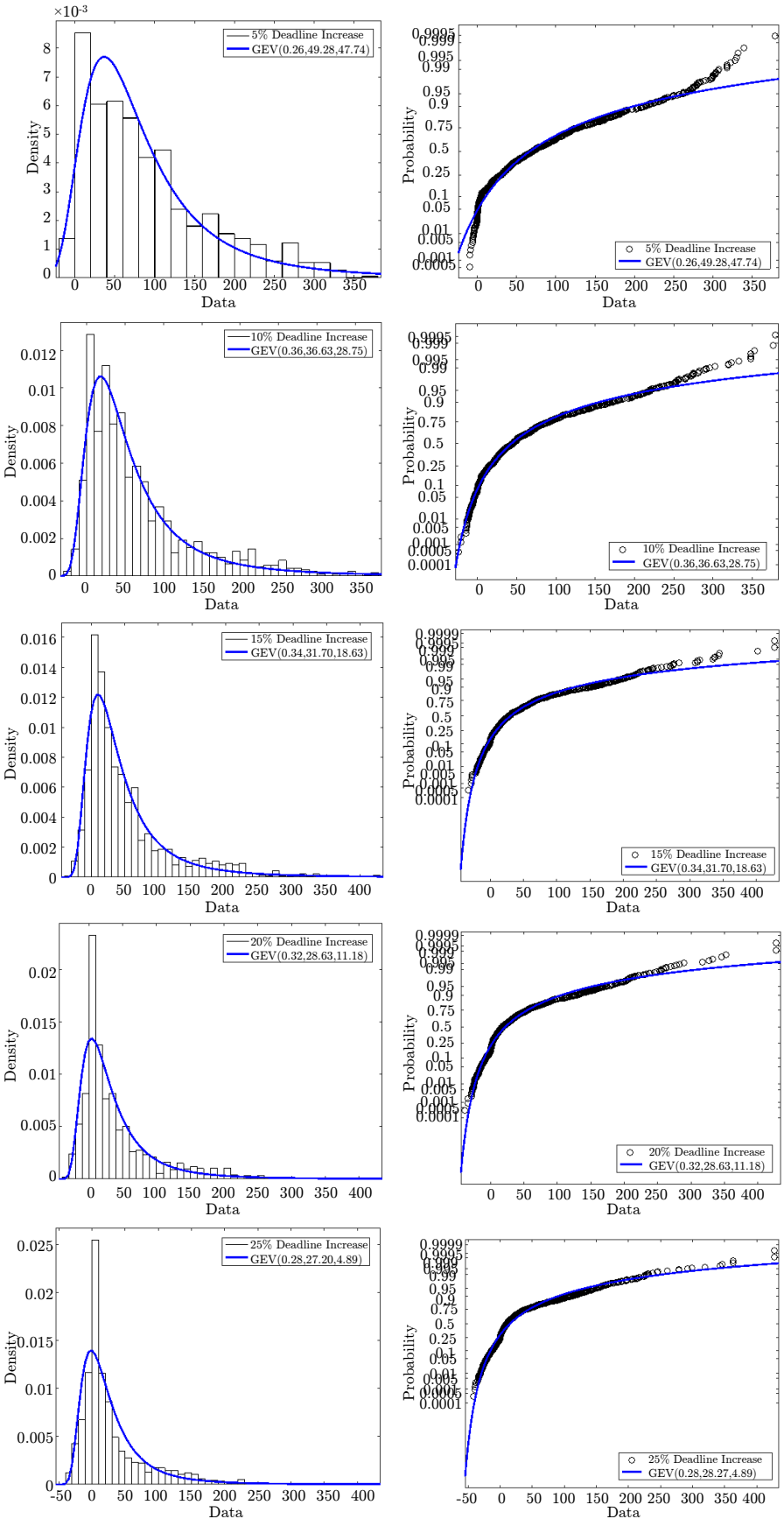

Figure 6.17: Density and Probability of GEV Distribution for Strategy S9.

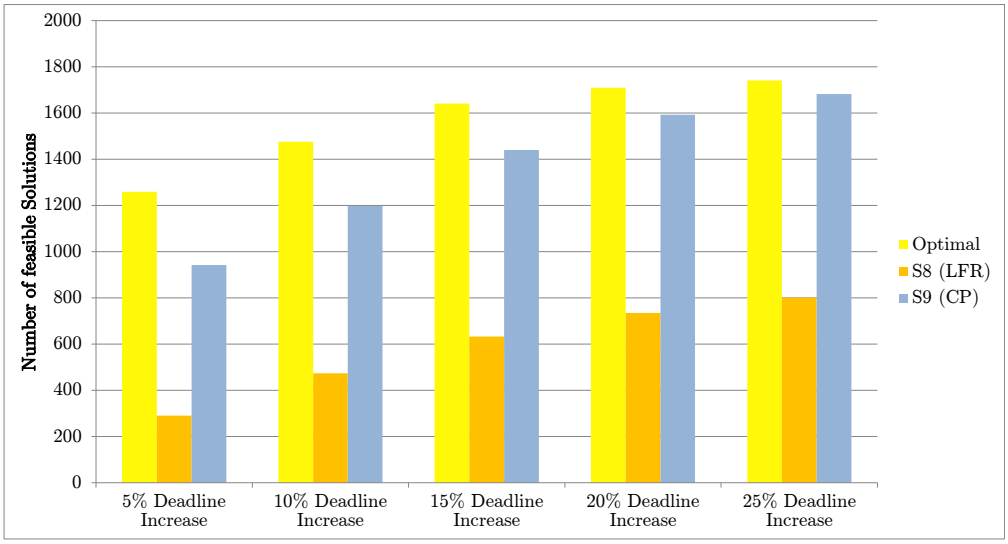

Figure 6.18: Number of Feasible Solutions for Different Values of Deadline Increase.

for a deadline increase of 25%, i.e. only 12 solutions were infeasible. In total 7,828 out of 8,765 schedules were found. In contrast, the LFR-heuristic used in strategy S8 (orange bars) leads to 2,935 out of 8,765 schedules in total. Thus, only for 33,49% of all cases a feasible solution was found. The CP-heuristic used in strategy S9 results in 6,857 feasible schedules. For a low value of deadline increase (5%), 943 schedules were found, i.e. 316 less than in the optimal vairant. For a higher value of deadline increase the number of feasible solutions is increased up to 1,683 schedules at 25% deadline increase. The CP-heuristic then finds only 58 less schedules than in the optimal variant.

In summary, the energy increase in both strategies is only moderately higher compared to the optimal solutions. Additionally, strategy S9 results in a high number of feasible solutions, but strategy S8 finds no suitable solution for most of the cases. Therefore, strategy S9 is a good alternative to the optimal variant, that can be further improved by using a more complex distribution of the power budget than a simple uniform distribution (see Sect. 4.3).

6.3 Experiments on Real World Platforms

6.3.1 Intel i7 3630qm, Intel i5 4570 and Intel i5 E1620

Model Validation To prove the accuracy of the power model explained in Sect. 5.3, three different computer systems with Intel processors are used as test platforms:

1. Intel i7 3630qm Ivy-Bridge based laptop
2. Intel i5 4570 Haswell-Bridge based desktop machine
3. Intel i5 E1620 Haswell-Bridge based server machine

To construct the power model, the power values are extracted by physical experiments using the Intel RAPL tool (see Sect. 2.5.5). As described in Sect. 5.1 the power consumption is measured for each frequency combination for 10 seconds with a sampling rate of 10 ms and all measurements are repeated five times. The power model is tested on all platforms for a mixed workload (micro benchmark 6) and on the server system also for the other benchmarks (1 to 5) described in Sect. 5.1.

The measured power values were used to construct the power model for the platforms and scenarios. The architecture specific tuning parameters (s, β, a and b) in Eq. 5.2 and 5.3 were then determined by using a least squares analysis.

Tab. 6.2 shows the individual parameters for the server platform for the ALU-, FPU-, SSE-, BP- and RAM-intensive workloads after fitting the physical measurements to Eq. 5.2 and 5.3 and optimizing the tuning parameters.

Table 6.2: Values of the Architecture Specific Tuning Parameters for the Benchmarks (i5 E1620).

	1. ALU	**2. FPU**	**3. SSE**	**4. BP**	**5. RAM**
P_{idle}	8.895 W	9.184 W	8.793 W	9.374 W	8.752 W
s	3.83 W	3.83 W	3.83 W	3.83 W	3.83 W
β	0.344 W/Hz^3	0.344 W/Hz^3	0.344 W/Hz^3	0.344 W/Hz^3	0.344 W/Hz^3
a	-3.01 Hz	-2.87 Hz	-2.82 Hz	-2.88 Hz	-2.92 Hz
b	5.61 Hz^2	5.14 Hz^2	5.62 Hz^2	5.14 Hz^2	5.61 Hz^2

The results of the least squares analysis for a mixed workload scenario on each platform is shown in Tab. 6.3.

Table 6.3: Values of the Architecture Specific Tuning Parameters for a Mixed Workload.

	i7 3630	**i5 4570**	**i5 E1620**
P_{idle}	3.781 W	5.976 W	8.728 W
s	1.29 W	0.42 W	3.83 W
β	0.340 W/Hz^3	0.091 W/Hz^3	0.344 W/Hz^3
a	-3.42 Hz	1.02 Hz	-2.87 Hz
b	5.88 Hz^2	12.08 Hz^2	6.13 Hz^2

Tab. 6.4 shows exemplary the difference between the data values and the model as the maximum and average deviation for the mixed workload. As seen in the table,

the maximum deviation was lowest using the desktop CPU (i5 4570) and higher using the laptop CPU (i7 3630) and the server CPU (i5 E1620). The reason for having a less exact fit using the server and laptop CPU is because of the significantly higher power output using the turbo boost on the i7 3630qm and on the i5 E1620 CPU, which is more difficult to fit to the curve than the more smooth power curve of the i5 4570 CPU. However, with a low average error value this model is considered feasible for the experiments.

Table 6.4: Difference Between the Data and the Model as Error Values Squared from Figure 6.19.

	i7 3630	**i5 4570**	**i5 E1620**
Avg. deviation	1.09%	0.84%	1.13%
Max. deviation	15.56%	7.28%	17.07%

In Fig. 6.19 the resulting power curves for the three test platforms are presented for the real data and for the model.

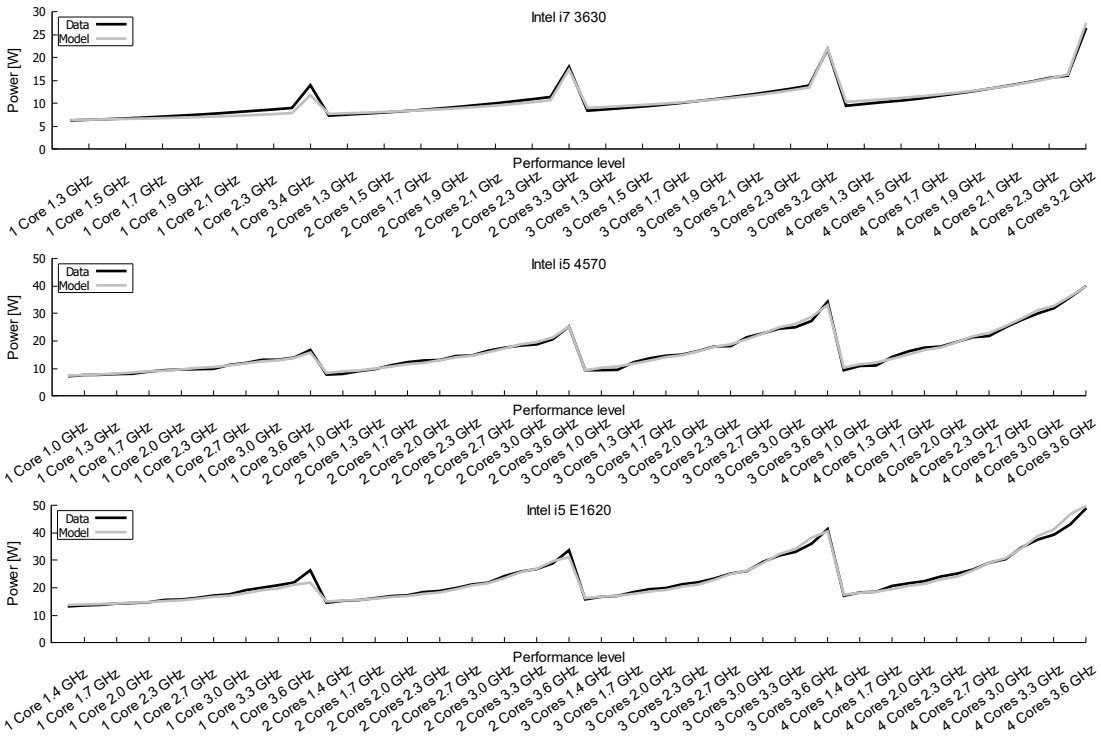

Figure 6.19: Power Consumption and Power Model for Different Platforms.

Real-world Evaluation First tests on the i5 4570 desktop machine and on the i7 3630qm laptop machine indicate that the power model from Sect. 5.3 cannot be used for these systems, although the model seems to fit the real data well as shown above. In Fig. 6.20, exemplary the power consumption of the desktop machine and the modeled power consumption for an example schedule with three tasks mapped onto two cores with different frequencies is shown.

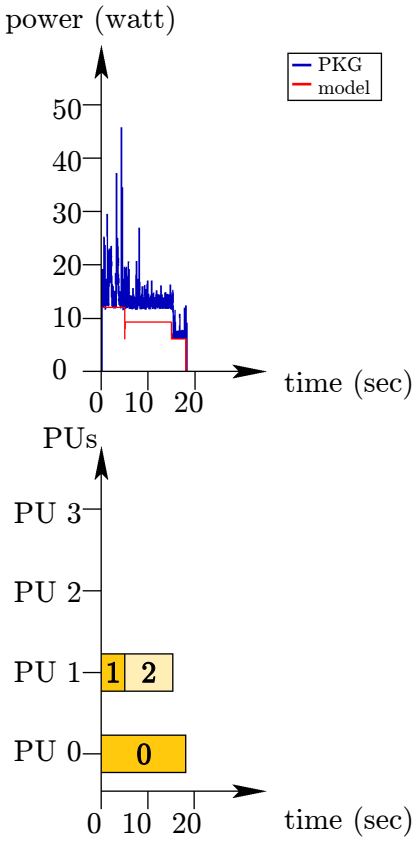

Figure 6.20: Power Consumption for an Example Schedule Using Different Frequencies (Desktop Machine).

The schedule is depicted in the bottom part of the figure, where task 0 and 1 are running with frequency 3.2 GHz (dark orange) and task 2 with frequency 2.5 GHz (light orange). In the top part, corresponding power curves are shown. We consider first the modeled power consumption (red line): As expected, the power is decreased after the frequency on PU 1 is scaled down from 3.2 GHz to 2.5 GHz (task 2). If we now consider the real power curve (blue line), this decrease of power is not seen. The reason for the unchanged power consumption, when scaling the frequency of a core is caused by the processor type and the ACPI standard (see Sect. 2.5.4). For both systems, voltage is scaled for all cores together and not separately. Although the

total voltage might be decreased depending on the frequencies, the ACPI standard forces the system to use a voltage level high enough to let all active cores running on the highest frequency in use. Hence, as long as not all cores are scaled down, the voltage level is kept high. After task 0 and task 2 have finished, the power consumption is, as expected, decreased to the idle power of around 6 W. As the system check tool (see Sect. 5.1) does not measure power consumption for different frequencies at the same time, i.e. only one frequency for a different number of active cores, the above explained behavior was not identified by this tool.

The behavior of the power consumption on the server system differs from the other systems, as now for each core a separate voltage regulator is used independently. Fig. 6.21 shows an example test case, where the expected behavior, i.e. the modeled power consumption fits the real power consumption well[4].

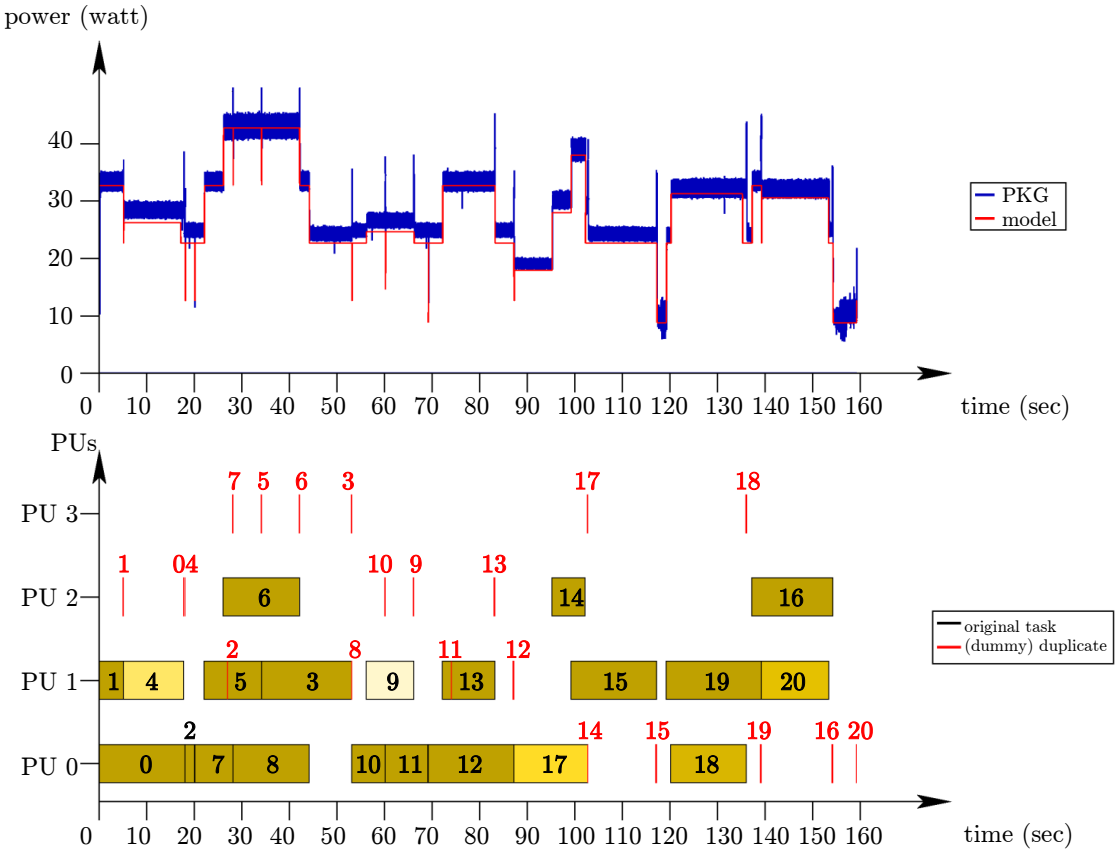

Figure 6.21: Power Consumption for an Example Schedule Using Different Frequencies (Server Machine).

[4]This figure was generated automatically by a developed figure generator that is included into the runtime system RUPS, to check (in an easy way) the correct behavior of the real system in use.

In this example, a fault-tolerant schedule is depicted that was executed in a fault-free case. The red tasks represent the duplicates (in this case only DDs are used) and different frequencies are indicated by different shades of orange. A darker orange represents a higher frequency, a lighter orange a lower frequency.

The server system as a common platform for clusters and grids is therefore used for the real-world evaluation. For each workload scenario, 922 schedules were tested that are related to 40 task graphs from the TB-Optimal test set with random properties and between 19 and 24 tasks. For each task graph the already existing schedule runs firstly without any changes and thus without any failures. Then, the fault-tolerant schedules that result from strategy S3, where only DDs and the BER-heuristic are used (see Sect. 4.4.2) were calculated and executed by the runtime system. Then, all fault-tolerant schedules run with a simulated failure at each task by exiting the corresponding MPI-Process directly before the task execution started.

The accuracy of the prediction is validated by comparing the predicted energy values that result from the scheduler with the real measurements of the runtime system.

In Fig. 6.22, exemplary the predicted and real energy consumption for a mixed workload for all schedules is presented.

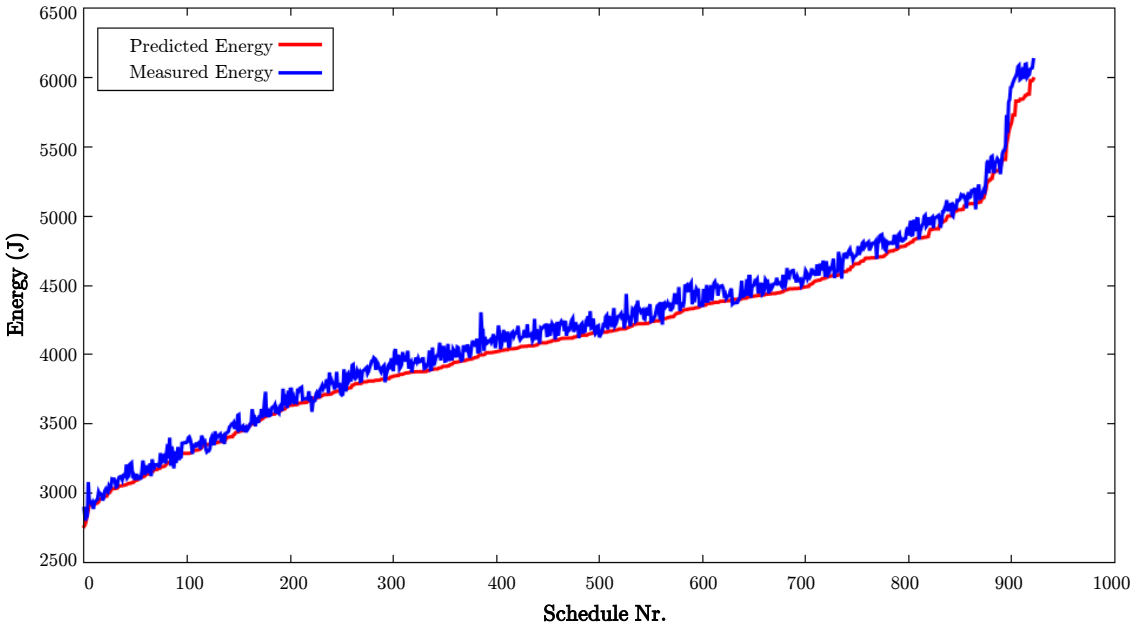

Figure 6.22: Comparison Between Predicted and Measured Energy Consumption (for Mixed Workloads).

Tab. 6.5 presents the averaged and maximum deviation for the prediction for all benchmarks. With a maximum deviation of around 7.73% and nearly 2.24% on average, the prediction fits the reality quite well.

Table 6.5: Differences Between Measured Data and Predictions for all Schedules.

	Deviation Prediction	
	Average	**Maximum**
1. ALU	2.42%	7.63%
2. FPU	2.00%	7.33%
3. SSE	1.81%	7.75%
4. BP	3.45	8.61%
5. RAM	2.09%	7.90%
6. Mixed (all tests)	1.64%	7.14%
In total (averaged)	2.24%	7.73%

Upper/Lower Bounds for the Energy Consumption The estimation of upper and lower bounds for PE and FT in Sect. 3.4 are independent of a certain system. They are correct in general for every system. But the estimation of the upper and lower bounds for E depends highly on the system in use. In Sect. 3.4, these bounds are estimated for a generalized power model. Therefore, the bounds have to be adapted to the server system by considering the measured values resulting from the system check tool for the estimation. Based on these values, the energy consumption of the i5 E1620 processor is calculated for a workload at different frequencies and number of cores. We assume a perfectly divisible workload can be executed in one second when using 4 cores and the highest supported frequency (3.6 GHz). Based on this assumption, the runtime of the workload for all other frequency/core combinations can be calculated. Then, the energy consumption for the workload can be achieved by multiplying the calculated runtime with the measured power consumption for the corresponding frequency and number of active cores. In Fig. 6.23, the energy behavior for the server system is presented.

The lowest energy consumption is at 2.3 GHz using four cores, the highest energy consumption is at the lowest frequency (1.2 GHz) using one core. Thus, a lower bound for E can be given by:

$$E_{best} = \frac{m_{seq}}{p_{max} \cdot 2.3\ GHz}. \tag{6.1}$$

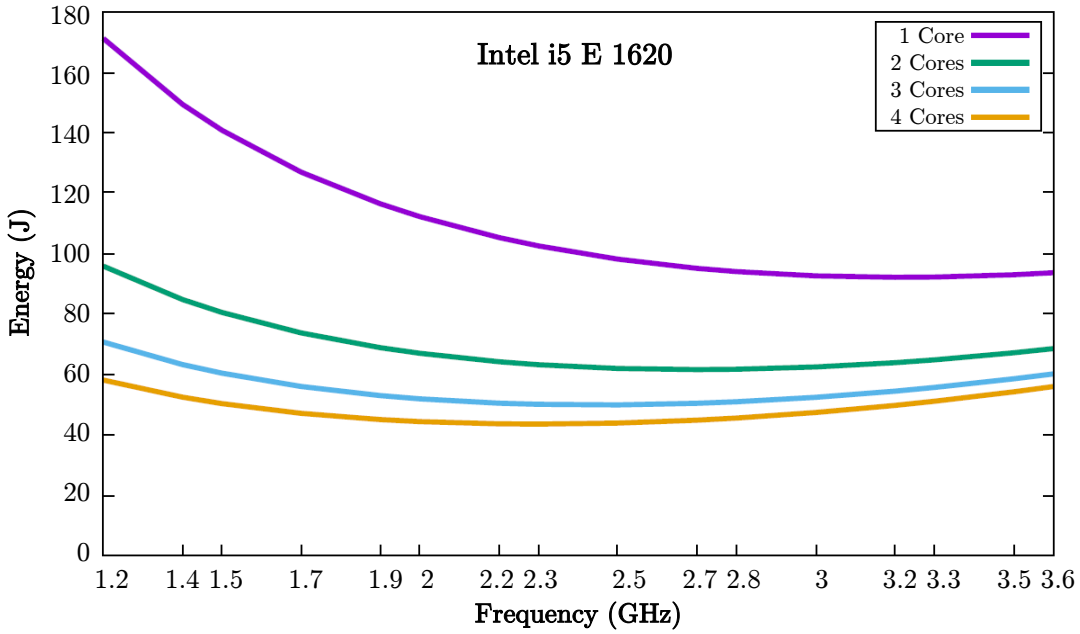

Figure 6.23: Energy Behavior for a Workload at Different Frequencies and Number of Cores.

An upper bound for E can be achieved by running all tasks in sequence on one PU with the lowest possible frequency f_{lowest}:

$$E_{worst} = \frac{m_{seq}}{1 \cdot f_{lowest}}. \tag{6.2}$$

Results The experiments are done with the TB-Optimal test set. As seen above, the system model closely reflects the real system in terms of energy consumption. This fact is used to simulate nearly 34,500 of the given schedules using the RUPS system. The trade-off between PE, FT, and E is evaluated with the four strategies S3 to S6 from Sect. 4.4.2. These strategies reflect system setups with one of the three parameters as inherently dominating. This choice will give a wide range of experiments with the extreme corner cases covered and everything between the corner cases. In all strategies, the BER-heuristic is used to reduce the energy consumption. The following strategies were used for the simulation, where the turbo frequency is not considered to avoid throttling effects:

S3: Use only DDs and start with the second highest supported frequency (3.5 GHz). In this scenario, the focus is firstly put on PE and secondly on E.

S4: Use Ds and DDs and start with the second highest supported frequency (3.5 GHz). This scenario mainly targets on PE, but also on FT.

S5: Create the schedules with a simple list scheduler that uses half of the PUs for original tasks, the other for Ds and start with the second highest supported frequency (3.5 GHz). Here the focus is mainly on FT.

S6: Select a lower frequency for original tasks and start with frequency level 7 (2.3 GHz). With this scenario we try to focus on E.

In Fig. 6.24 the results for all strategies are presented. The equations from Sect. 5.3 are used to find good upper and lower bounds for PE, E, and FT. For a better illustration only the results for systems with 4 PUs (in total 6500 schedules) are shown. But the results for the other number of PUs are similar with respect to the overall trends. They differ only slightly by small shifts.

The left column of the figure presents the trade-off between E and PE for all strategies (S3, S4, S5 and S6), the middle column presents the trade-off between E and FT and the right column presents the trade-off between PE and FT.

We start with strategy S3. In this strategy, a better performance also leads to a better energy consumption. With a performance of nearly 100% the energy consumption goes down to around 5% (related to the best and worst cases from the boundaries). This behavior seems to be related to the high idle power of the system compared to the dynamic power. The higher the idle power is, the better it is to run on a high frequency, e.g. at the second highest like here. When we consider the trade-off between E and FT, a lower energy consumption in the fault-free case, and thus a higher performance of the schedule, leads to a higher performance overhead in case of a failure. This behavior results from the decreasing number and size of gaps within a schedule, when improving the performance. Because then each DD leads directly to a shift of its successor and succeeding tasks. The trade-off between PE and FT shows directly the same behavior. The higher the performance the higher is also the performance overhead.

In strategy S4, Ds and DDs were used for the fault tolerance. Here the left part ($E \leftrightarrow PE$) of the figure is more spread compared to S3. This indicates that especially for a lower performance more gaps can be filled with Ds. This leads to an increased energy. The middle part of the figure ($E \leftrightarrow FT$) shows the resulting improvement of the performance overhead in case of a failure. And also on the right part ($PE \leftrightarrow FT$) a slightly shift of all results to the left can be seen.

In strategy S5, the focus is put on a good FT result. On the left side, the performance is much lower and the energy consumption is much higher than for strategies S3 and S4. But in this strategy no performance overhead in case of a

failure exists. Therefore, the middle and right boxes of the figure are empty and the performance in the fault case is equal to the performance in the fault-free case.

Strategy S6 is used to generate schedules that run with a lower frequency (frequency level 7, 2.3 GHz). The energy consumption can be improved by running on a lower frequency, bot only if the performance is increased. Then, nearly the best energy consumption of 0%, i.e. the lower bound of E can be reached. The other both trade-offs are the same like for strategy S3. They are just a little bit stretched.

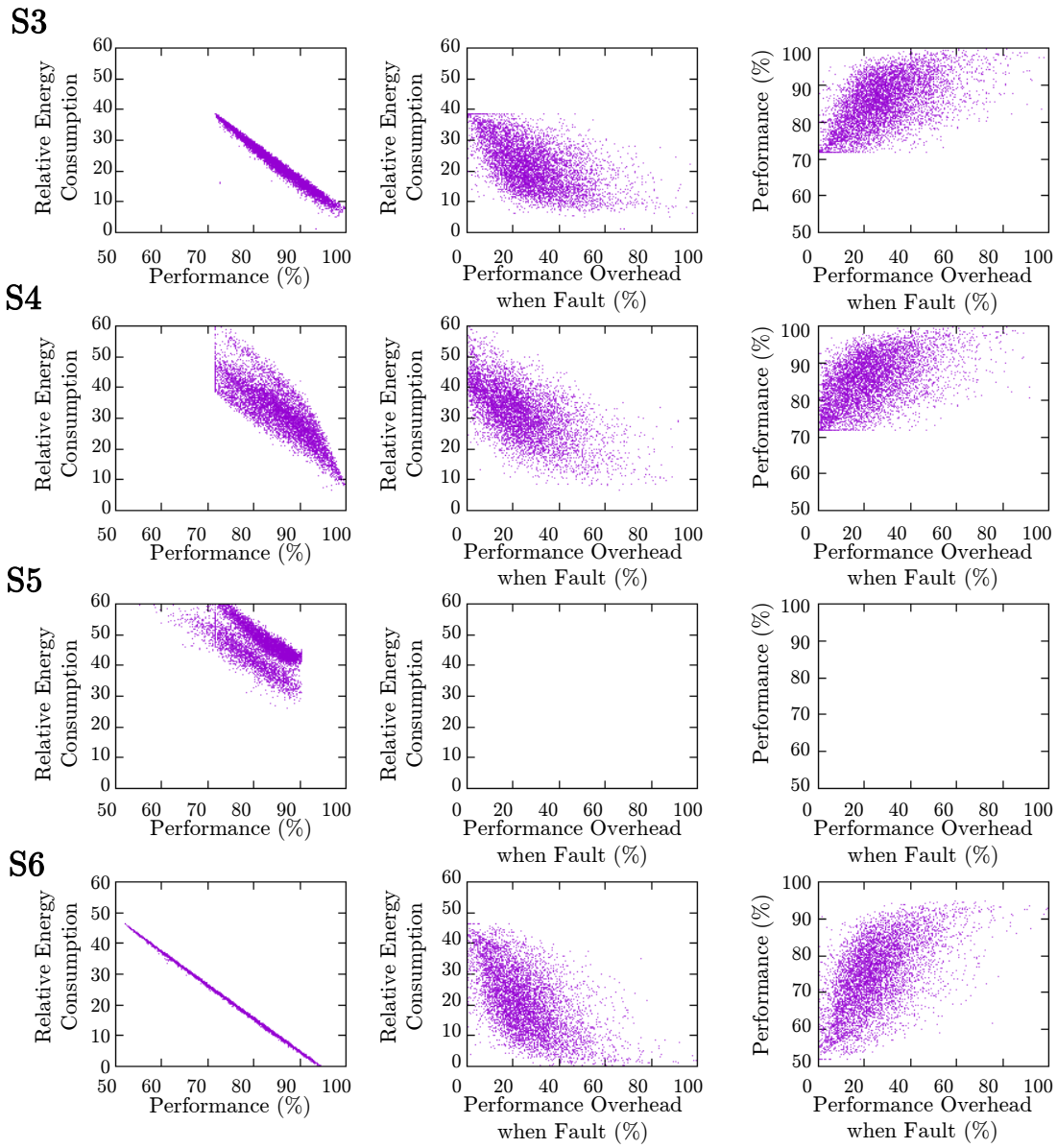

Figure 6.24: Results when Scheduling According to Scenarios A,B,C,D Showing: Relative Energy Consumption (Lower is Better), Performance (Higher is Better), Performance Overhead when Fault (Lower is Better).

With these experiments we could show that there does not exist any overall solution for that three-variable problem without giving up at least one of the three parameters. Thus, the decision on which parameter the main focus lies must be made by the user.

6.3.2 Intel SCC

The experiments for the Intel SCC are done based on the power model presented by Eitschberger and Keller [45]. They use a cubic power function for the whole processor and separate the power consumption of components like the cores, network or memory controller. The power function can be described by the following equation:

$$P_{SCC}(f_1, ..., f_6) = 8 \cdot \sum_{i=1}^{6} (b_c \cdot {f_i}^3 + s_c \cdot f_i) + \tilde{s}, \tag{6.3}$$

where s̃ represents a static value for the power consumption of the network and memory controller. The power consumption of a core is given by a dynamic part $b_c \cdot f_i^3$ and a static part $s_c \cdot f_i$, where b_c and s_c are device specific constants and f_i is the frequency for island i. As 6 voltage islands exists with 8 cores each, the sum of all voltage islands must be multiplied by the number of cores in every island. They obtain the values $b_c \approx 2.015 \cdot 10^{-9}$ Watt/MHz^3 *(MegaHertz)*, $s_c \approx 10^{-6}$ Watt/MHz and $\tilde{s} \approx 23$ Watt and an averaged error value of 5.58%. In their power model, they consider voltage islands with fully loaded cores at a given frequency and do not consider idle power.

To use the power model for the experiments in this thesis, we assume that each schedule is executed 8 times in parallel, e.g. with different input values, so that all cores of a voltage island are either fully loaded at the same frequency or in idle mode. The idle power is assumed to be zero for simplicity.

The task graphs of the real applications (robot control and sparse matrix) and the test set TB-SLS are used with strategy S3 and the power model of the SCC for the experiments. The results of the test set TB-SLS are then compared to the results of the real applications, because the schedules for the real applications are also only generated by the simple list scheduler described in the beginning of this chapter. Additionally, the communication times in the TB-SLS test set are ignored as the communication is directly done within the SCC. The task graphs of the real applications do also not consider any communication times. In Fig. 6.25 the energy improvements in the fault-free case are depicted.

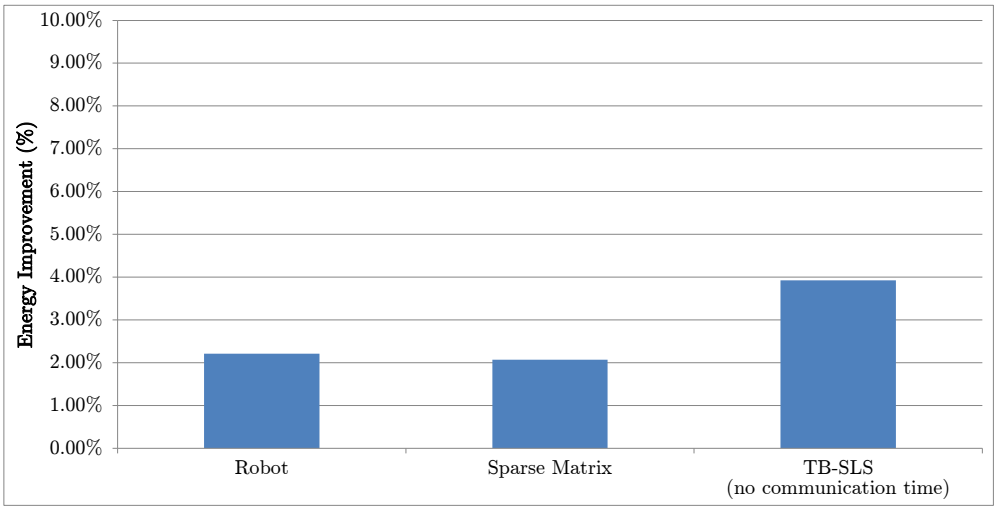

Figure 6.25: Energy Improvement in a Fault-free Case (Strategy S3).

For the SCC as an example of a manycore system and task graphs without communication costs, the energy can be improved by around 2% for the real applications and by 3.93% for the TB-SLS test set without communication costs.

6.4 Analysis of the Scheduling Time

The averaged and maximum scheduling time to generate a schedule is shown in Tab. 6.6.

Table 6.6: Scheduling Time.

	Schedules with 7-24 Tasks (heuristics)	Schedules with 25 - 250 Tasks (heuristics)	Schedules with 7 - 12 Tasks (optimal)
Average	0.038 s	11.27 s	21.63 s
Maximum	8.78 s	194.31 s	1,723.14 s

For the test sets TB-Optimal, TB-ACO and TB-SLS with a small number of tasks (7 - 24 tasks), the averaged scheduling time is very low (0.038 s). Even the maximum scheduling time of 8.78 seconds is moderate compared to the runtime of a schedule (of around 120 s on average). For all strategies the scheduling time for the small cases varies in a small range of 0.029 s and 0.053 s, where the lower values were achieved in strategies S1 to S4.

The scheduling time for the large cases with up to 250 tasks is on average at 11.27 s. The maximum is at 194.31 s. Also here the scheduling time is small compared to the runtime of a schedule (of around 1000 s on average).

In contrast, the scheduling time for the optimal solutions for 7 - 12 tasks is very high with 21.63 s on average and with a maximum of 1,723.14 s. Especially for schedules with a high number of PUs the solution space is significantly increased. The time for the generation of optimal solutions is therefore disproportional high compared to the improvements resulting from the optimization.

6.5 Summary & Discussion

In this chapter, the proposed strategies from Sect. 4.4 were evaluated and analyzed for various power models and platforms. As a result, the accuracy of the power model for parallel platforms, presented in Sect. 5.3, is on average quite well with a standard deviation of around 1% (see Sect. 6.3.1). For processors with separate voltage regulators for each core like the Intel i5 E1620, the prediction of the scheduler (and simulator) is close to the real behavior. Only small standard deviations of 2.24% (on average) and 7.73% (on maximum) exist. Different types of workloads can be considered by using various workload classes, as proposed in Sect. 5.1, and corresponding parameters for the power model, as presented in Sect. 6.3.1.

If the frequency of PUs is constant over the whole schedule execution, the use of available and free PUs for the placement of duplicates leads to a better performance but to a significant higher energy consumption. However, if frequency scaling is supported by the parallel system, most of the energy improvements result from idle times where the frequency is scaled down to the lowest possible. In addition, up to 6% can be saved, when the frequency for tasks is scaled, too (see Sect. 6.2.2). A combined optimization of performance and energy consumption is slightly better than a separated optimization. Compared to solutions with an optimal classical scheduling and strategy S3 for reducing the energy consumption, the difference is small. Hence, using S3 is a good alternative to the optimal solutions.

If the focus of a user is put on the fault tolerance, strategy S5 is advantageous for high numbers of PUs. For small numbers of PUs, strategy S4 leads to better results.

Strategy S6 with the highest energy savings is worthwhile for parallel platforms, where the most energy efficient frequency is in the upper range of the supported frequencies. Otherwise, the performance decrease is disproportional high so that then strategy S3 is the better choice.

For the fault-case, strategy S7 leads to high performance improvements whereas strategy S9 is a good solution for improving the energy consumption, as the number and quality of feasible solutions is close to the optimal. Finally, the scheduling time for all strategies is small.

Exemplary user preferences and favored strategies are summarized in Tab. 6.7.

Table 6.7: User Preferences and Favored Strategies.

User Preferences		Favored
Fault-free Case	Fault Case	Strategies
PE (FT)	PE	S2 + S7
PE (E)	E	S3 + S9
PE (FT) (E)	E	S4 + S9
E (PE)	PE	S3 + S7
E	E	S6 + S9
FT	PE	S5 + S7
FT (PE)	PE	S4 + S7
FT	E	S5 + S9
FT (PE)	E	S4 + S9

As seen in Tab. 6.7, various user preferences are represented by combinations of the proposed strategies. Next to major objectives, also minor criteria can be considered, resulting in a variety of possible solutions with reasonable results. The worsening of criteria that are not focused is moderate. Thus, the investment for improving favored objectives is low. In addition, the strategies can be hidden from users that do not have any background knowledge about scheduling, so that they only have to give their preferences by selecting a combination of objectives. Then, the corresponding strategies can be chosen automatically by the scheduler.

7 Conclusions

In this thesis, the interplay between performance, fault tolerance and energy consumption was explored. In general, an overall optimal solution for this three-dimensional optimization problem does not exist. All two-dimensional combinations of the above mentioned objectives already result in a trade-off, where the improvement of one criterion leads to a worsening of the other one. In the literature, the focus is therefore often put on one criterion that is improved, while the other one is fixed. For example, the energy consumption for a schedule is optimized without changing the performance. A combination of all three above mentioned objectives is even more complex, since a change of an objective no longer affects only another but also can affect both other objectives. In the literature, the three-dimensional optimization is rarely considered. Often, the focus on the primary and secondary criterion is already fixed in advance. However, as a result, only a very small part of the total view of the trade-off is addressed on the one hand, and also no other orientations are considered on the other hand.

To discuss the trade-off between the three objectives in detail, in this thesis several fault-tolerant and energy-efficient strategies were presented that focus on different criteria in the fault-free case and ultimately represent several preferences of a user. Therefore, the fault-tolerant task duplication-based scheduling approach of Fechner et al. [59] is used, extended and combined with energy-efficient options and heuristics. Additionally, another trade-off between the fault-free and fault case is not addressed in previous work. But the preferences of a user can change between both cases. For example in a fault-free case, the energy consumption is the most important criterion, while in the fault case the performance is dominating. Therefore, in this work also strategies for the fault case were presented that either focus on the performance or on the energy consumption. In addition, energy-optimal solutions were presented and used for comparison.

The strategies were evaluated with various test sets from the benchmark suite of Hönig [84]. In a first step, the energy consumption for the resulting schedules was predicted based on a generalized power model, independently of a certain platform.

In a second step, power models for real parallel platforms with an acceptable accuracy were constructed, tested and used in the evaluation. Therefore, in this work a fault-tolerant and energy-efficient prototype runtime system called RUPS (Runtime system for User Preferences-defined Schedules) was presented, that is realized with an extended fault-tolerant version of MPI and supports frequency scaling by integrating DVFS.

To analyze and visualize the trade-off between all objectives, the performance and energy consumption for all strategies were given for both the fault-free and fault case. The results show, that various factors have a significant influence on the trade-off like different energy curves of processors, the number of used PUs for the placement of duplicates or a tolerable increase of the makespan in case of a failure. In terms of energy, the most improvements result from idle times, where a lowest supported frequency can be used and only a smaller part can be achieved by also scaling frequencies for tasks.

8 Outlook

As in this thesis several subjects like performance, fault tolerance, energy efficiency, heuristics, optimization, power modeling, or runtime system are combined under a broader topic, a wide spectrum of future research can be done. Therefore, only some possible directions are described in the following:

Performance In this thesis only homogeneous systems are considered. An extension to heterogeneous systems can be included to get more possibilities to improve the performance of a schedule. Until now, the CBF-heuristic is only implemented for the fault-free case. Therefore, the implementation for the fault case can be done to also speedup the whole schedule, i.e. the performance of the schedule in case of a failure. The EP-heuristic can be improved by also allowing tasks to start earlier than the time given in the static schedule.

Fault tolerance The probability of a fault or failure depending on the runtime of a task, schedule, or PU is not considered. Tasks for example can be weighted according to their runtime, so that longer running tasks get a higher probability to fail. Another aspect focuses on the number of failures, that can be tolerated by the runtime system. Mainly two possibilities exist: Including more copies of each original task, i.e. duplicates directly in the fault-free case into the schedule or delete all unnecessary duplicates in case of a failure and include a new duplicate for each task. The first approach might lead to a lower performance in case of more than one failure because of an inefficient placement of the duplicates, the second approach in contrast is a hybrid approach where the duplicates after a failure are included dynamically during the runtime of a schedule and therefore lead to a lower performance, but also to a better placement of duplicates. Additionally, the second approach is in contrast to the first approach restricted to only one failure at a time. In this work only permanent faults (failures) are considered. Another extension could be to include also the toleration of temporary faults, where a PU is available again after some time.

Energy Efficiency The energy consumption in case of a failure can be improved for strategy S9 by using a more complex distribution for the power budget. The number of feasible solutions for strategy S8 and also the energy consumption in case of a failure can be improved by considering a certain percentage value for the communication time when calculating a new frequency. As the scheduling time of strategies S7 - S9 is small compared to the schedule runtime, another approach is to calculate the energy and makespan for all strategies in the fault case and choose the best one according to the users' preferences. Another approach to save energy can be done by considering also the frequency scaling for the network in use. For example, the Intel SCC offers 2 frequency levels for the network: 800 MHz and 1600 MHz.

Optimization One question that arose during the creation of the ILPs in this work, but that is not considered in this thesis is, if it is possible to find the maximum number of PUs necessary for an energy optimal solution with the help of the information from a performance optimal solution and the task graph when the deadline is set to the makespan of the performance optimal schedule. For example, the number of PUs for an energy optimal solution must be at least as high as the number of the performance optimal solution, because otherwise the schedule would be prolonged and thus the deadline cannot be met. On the other side, an energy optimal solution might need more PUs than the performance optimal solution to get more possibilities to scale down tasks.

Power Modeling In this thesis a very simple power model is used for the intel processors. This model might be improved by including more details into this model. Also other components of a computer system might be included into the overall energy rating, like the energy consumption for the memory controller or for the hard disks. Another possibility is to find a power model for processors, that have no separate voltage regulators like the discussed i5 desktop processor or the i7 laptop processor. Then, new strategies and heuristics can be implemented, that consider frequency scaling only for all cores together and not independently.

Runtime System The prototype runtime system can be extended by including complete messages for the communication and not only the message header. Additionally, the runtime system can be adapted to support real applications based on a task graph instead of simulated workloads.

Bibliography

[1] J. H. Abawajy, "Fault-tolerant Scheduling Policy for Grid Computing Systems," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*. IEEE, 2004, pp. 238–245.

[2] T. L. Adam, K. M. Chandy, and J. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[3] N. Agarwal, P. Chauhan, and D. Nitin, "Fault-tolerant Heterogeneous Limited Duplication Scheduling Algorithm for Decentralized Grid," *International Journal of Computers & Technology (IJCT)*, vol. 4, no. 3, pp. 765–775, 2013.

[4] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," in *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '96)*, Jun 1996, pp. 207–213.

[5] K. Ahn, J. Kim, and S. Hong, "Fault-tolerant Real-time Scheduling Using Passive Replicas," in *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*. IEEE, 1997, pp. 98–103.

[6] B. Alam and A. Kumar, "Fault Tolerance Issues in Real Time Systems with Energy Minimization," *International Journal of Information and Computation Technology (IJICT)*, vol. 3, no. 10, pp. 1001–1008, 2013.

[7] S. Albers, "Energy-efficient Algorithms," *Communications of the ACM*, vol. 53, no. 5, pp. 86–96, May 2010. [Online]. Available: http://doi.acm.org/10.1145/1735223.1735245

[8] D. G. Amalarethinam and G. J. Mary, "A New DAG-based Dynamic Task Scheduling Algorithm (DYTAS) for Multiprocessor Systems," *International Journal of Computer Applications*, vol. 19, no. 8, pp. 24–28, April 2011.

[9] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid Scheduling for the Parallel Solution of Linear Systems," *Parallel Computing*, vol. 32, no. 2, pp. 136–156, 2006.

[10] C. Anglano and M. Canonico, "Fault-Tolerant Scheduling for Bag-of-Tasks Grid Applications," in *Proceedings of the European Grid Conference (EuroGrid '05). Lecture Notes in Computer Science*. Springer, 2005, p. 630.

[11] ARM Ltd., "White Paper: big.LITTLE Technology: The Future of Mobile," 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[12] G. Aupy, A. Benoit, P. Renaud-Goud, and Y. Robert, "Energy-aware Algorithms for Task Graph Scheduling, Replica Placement and Checkpoint Strategies," in *Handbook on Data Centers*. Springer New York, 2015, pp. 37–80.

[13] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Checkpointing Strategies for Scheduling Computational Workflows," *International Journal of Networking and Computing (IJNC)*, vol. 6, no. 1, pp. 2–26, 2016.

[14] C. R. Babu and C. S. Rao, "Automatic Checkpointing-based Fault Tolerance in Computational Grid," in *Proceedings of the International Conference on Computing, Management and Telecommunications (ComManTel '14)*. IEEE, 2014, pp. 41–45.

[15] S. D. Babu, C. R. Babu, and C. S. Rao, "An Efficient Fault Tolerance Technique Using Checkpointing and Replication in Grids Using Data Logs," *International Journal Publications of Problems and Applications in Engineering Research (IJPAPER)*, 2013.

[16] J. Balasangameshwara, "Survey on Job Scheduling, Load Balancing and Fault Tolerance Techniques for Computational Grids," *Global Journal of Technology and Optimization*, vol. 6:1, 2014. [Online]. Available: http://dx.doi.org/10.4172/2229-8711.1000169

[17] M. Balpande and U. Shrawankar, "Checkpointing-based Fault-tolerant Job Scheduling System for Computational Grid," *International Journal of Advancements in Technology (IJACT)*, vol. 5, no. 2, 2014.

[18] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware Scheduling for Real-time Systems: A Survey," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, 2016.

[19] S. Bansal, P. Kumar, and K. Singh, "An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 6, pp. 533–544, Jun. 2003. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2003.1206502

[20] P. Baptiste, "Scheduling Unit Tasks to Minimize the Number of Idle Periods: A Polynomial Time Algorithm for Offline Dynamic Power Management," in *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA '06)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006, pp. 364–367. [Online]. Available: http://dl.acm.org/citation.cfm?id=1109557.1109598

[21] B. Barney. (2010) POSIX Threads Programming. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[22] ——. (2016) Introduction to Parallel Computing. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/

[23] R. Basmadjian and H. de Meer, "Evaluating and Modeling Power Consumption of Multi-core Processors," in *Proceedings of the 3rd International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy '12)*, May 2012, pp. 1–10.

[24] A. Benoit, M. Hakem, and Y. Robert, "Fault-tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms," in *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing (IPDPS '08)*. IEEE, April 2008, pp. 1–8.

[25] ——, "Fault-tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms," in *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing (IPDPS '08)*. IEEE, 2008, pp. 1–8.

[26] K. Berry, F. Navarro, and C. Liu, "Application-level Voltage and Frequency Tuning of Multi-phase Program on the SCC," in *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing*

Systems (ADAPT '13). ACM, 2013, pp. 1:1–1:7. [Online]. Available: http://doi.acm.org/10.1145/2484904.2484905

[27] P. Bjorn-Jorgensen and J. Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architectures," in *Proceedings of the 5th International Workshop on Hardware/Software Co-Design*. IEEE Computer Society, 1997.

[28] W. Bland, "User Level Failure Mitigation in MPI," in *Proceedings of the European Conference on Parallel Processing (Euro-Par 2012)*. Springer, 2012, pp. 499–504.

[29] C. Blum, "Ant Colony Optimization: Introduction and Recent Trends," *Physics of Life Reviews*, vol. 2, no. 4, pp. 353–373, 2005.

[30] A. Bode, K. Kran, U. Brüning, M. Cin, W. Händler, F. Hertweck, U. Herzog, F. Hofmann, R. Klar, C. Linster *et al.*, *Parallelrechner: Architekturen - Systeme - Werkzeuge*, ser. XLeitfäden der Informatik. Vieweg+Teubner Verlag, 1995. [Online]. Available: https://books.google.de/books?id=0pMePQAACAAJ

[31] C. Bolchini, M. Carminati, and A. Miele, "Self-adaptive Fault Tolerance in Multi-/Many-core Systems," *Journal of Electronic Testing*, vol. 29, no. 2, pp. 159–175, 2013.

[32] D. Bozdağ, U. Catalyurek, and F. Özgüner, "A Task Duplication Based Bottom-up Scheduling Algorithm for Heterogeneous Environments," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS '06)*. IEEE Computer Society, 2006, pp. 160–160. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1899086

[33] D. Bozdağ, F. Özgüner, E. Ekici, and U. Catalyurek, "A Task Duplication Based Scheduling Algorithm Using Partial Schedules," in *Proceedings of the 34th International Conference on Parallel Processing (ICPP '05)*. IEEE, 2005, pp. 630–637.

[34] 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

[35] D. J. Brown and C. Reams, "Toward Energy-efficient Computing," *Communications of the ACM*, vol. 53, no. 3, pp. 50–58, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1666420.1666438

[36] A. Burkimsher, I. Bate, and L. S. Indrusiak, "A Survey of Scheduling Metrics and an Improved Ordering Policy for List Schedulers Operating on Workloads with Dependencies and a Wide Variation in Execution Times," *International Journal of Future Generation Computer Systems*, vol. 29, no. 8, pp. 2009 – 2025, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12002257

[37] Y. Cai, S. M. Reddy, and B. M. Al-Hashimi, "Reducing the Energy Consumption in Fault-Tolerant Distributed Embedded Systems with Time-Constraint," in *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED'07)*, March 2007, pp. 368–373.

[38] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[39] A. Chandak, B. Sahoo, and A. K. Turuk, "An Overview of Task Scheduling and Performance Metrics in Grid Computing," in *Proceedings of the 2nd National Conference-Computing, Communication and Sensor Network*. Foundation of Computer Science, New York, USA., 2011, pp. 30–33.

[40] C. Y. Chen and C. P. Chu, "A 3.42-Approximation Algorithm for Scheduling Malleable Tasks under Precedence Constraints," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 8, pp. 1479–1488, Aug 2013.

[41] G. Chen, K. Huang, and A. Knoll, "Energy Optimization for Real-time Multiprocessor System-on-chip with Optimal DVFS and DPM Combination," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 111, 2014.

[42] H. Cho, B. Ravindran, and E. D. Jensen, "An Optimal Real-Time Scheduling Algorithm for Multiprocessors," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, Dec 2006, pp. 101–110.

[43] M. Chtepen, F. H. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-tolerant Grids," *IEEE Transactions on Parallel and Distributed systems (TPDS)*, vol. 20, no. 2, pp. 180–190, 2009.

[44] P. Cichowski, "Implementierung eines fehlertoleranten statischen Schedulers für Grid-Anwendungen," Diplomarbeit, FernUniversität in Hagen, 2009.

[45] P. Cichowski, J. Keller, and C. Kessler, "Modelling Power Consumption of the Intel SCC," in *Proceedings of the 6th Many-core Applications Research Community Symposium (MARC '12)*, E. Noulard and S. Vernhes, Eds. Toulouse, France: ONERA, The French Aerospace Lab, Jul. 2012, pp. 46–51. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00719033

[46] B. Cirou and E. Jeannot, "Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems," in *Proceedings of the 30th International Conference on Parallel Processing Workshops (ICPP'01)*. IEEE, 2001, pp. 231–236.

[47] J. Cong and B. Yuan, "Energy-efficient Scheduling on Heterogeneous Multi-core Architectures," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '12*. ACM, 2012, pp. 345–350.

[48] M. I. Daoud and N. Kharma, "An Efficient Genetic Algorithm for Task Scheduling in Heterogeneous Distributed Computing Systems," in *Proceedings of the IEEE International Conference on Evolutionary Computation (CEC '06)*, 2006, pp. 3258–3265.

[49] ——, "Efficient Compile-time Task Scheduling for Heterogeneous Distributed Computing Systems," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS '06)*, vol. 1. IEEE, 2006, pp. 9–18.

[50] F. Dong, "A Taxonomy of Task Scheduling Algorithms in the Grid," *Parallel Processing Letters*, vol. 17, no. 04, pp. 439–454, 2007.

[51] E. Dubrova, *Fault-Tolerant Design*. Springer New York, 2013. [Online]. Available: https://books.google.de/books?id=FRs_AAAAQBAJ

[52] K. Echtle, *Fehlertoleranzverfahren*. Springer, 1990.

[53] P. Eitschberger and J. Keller, "Efficient and Fault-tolerant Static Scheduling for Grids," in *Proceedings of the 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC '13)*, 2013, pp. 1439–1448.

[54] ——, "Energy-Efficient and Fault-tolerant Taskgraph Scheduling for Manycores and Grids," in *Proceedings of the 1st Workshop on Runtime and Operating Systems for the Many-core Era (ROME '13)*, 2013, pp. 769–778.

[55] ——, "Energy-Efficient Task Scheduling in Manycore Processors with Frequency Scaling Overhead," in *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '15)*, March 2015, pp. 541–548.

[56] ——, "Fault-Tolerant Parallel Execution of Workflows with Deadlines," in *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '17)*, March 2017.

[57] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[58] G. E. Fagg and J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2000, pp. 346–353. [Online]. Available: http://dl.acm.org/citation.cfm?id=648137.746632

[59] B. Fechner, U. Hönig, J. Keller, and W. Schiffmann, "Fault-Tolerant Static Scheduling for Grids," in *Proceedings of the 13th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '08)*, 2008, pp. 1–6.

[60] D. G. Feitelson and L. Rudolph, "Metrics and Benchmarking for Parallel Job Scheduling," in *Proceedings of the Workshop on Job Scheduling*

Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 1998, pp. 1–24. [Online]. Available: http://dx.doi.org/10.1007/BFb0053978

[61] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers (TC)*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

[62] S. Forrest, "Genetic Algorithms," *ACM Computing Surveys*, vol. 28, no. 1, pp. 77–80, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/234313.234350

[63] I. Foster, *Designing and Building Parallel Programs*. Addison Wesley Publishing Company Reading, 1995, vol. 191.

[64] Y. Gao, H. Rong, and J. Z. Huang, "Adaptive Grid Job Scheduling with Genetic Algorithms," *Future Generation Computer Systems*, vol. 21, no. 1, pp. 151–161, 2005.

[65] R. Garg and A. K. Singh, "Fault Tolerant Task Scheduling on Computational Grid Using Checkpointing Under Transient Faults," *Arabian Journal for Science and Engineering*, vol. 39, no. 12, pp. 8775–8791, 2014.

[66] F. C. Gärtner, "Formale Grundlagen der Fehlertoleranz in verteilten Systemen," Ph.D. dissertation, Technische Universität, Darmstadt, Juli 2001. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/162/

[67] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters," in *Proceedings of the ACM/IEEE Supercomputing Conference (SC '05)*. IEEE, 2005, pp. 34–34.

[68] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276–291, 1992.

[69] R. Giroudeau, J.-C. Konig, F. K. Moulai, and J. Palaysi, "Complexity and Approximation for Precedence Constrained Scheduling Problems with Large Communication Delays," *Theoretical Computer Science*, vol. 401, no. 1, pp. 107 – 119, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397508002387

[70] F. Glover and M. Laguna, *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[71] B. Goel and S. A. McKee, "A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*, May 2016, pp. 273–282.

[72] S. Goswami and A. Das, "Deadline Stringency Based Job Scheduling in Computational Grid Environment," in *Proceedings of the 2nd International Conference on Computing for Sustainable Global Development (INDIACom '15)*. IEEE, 2015, pp. 531–536.

[73] A. Grama, A. Gupta, G. Krypis, and V. Kumar, *Introduction to Parallel Computing, Second Edition*. Addison Wesley, 2003.

[74] E. Günther, F. G. König, and N. Megow, "Scheduling and Packing Malleable Tasks with Precedence Constraints of Bounded Width," in *Proceedings of the 7th International Workshop on Approximation and Online Algorithms (WAOA '09), Revised Papers*. Springer Berlin Heidelberg, 2010, pp. 170–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12450-1_16

[75] S. Gupta, R. Rajak, G. K. Singh, and S. Jain, "Review of Task Duplication Based (TDB) Scheduling Algorithms," *SmartCR*, vol. 5, no. 1, pp. 67–75, 2015.

[76] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An Energy Efficient Feature Survey of the Intel Haswell Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*, May 2015, pp. 896–904.

[77] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring Energy Consumption for Short Code Paths Using RAPL," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2425248.2425252

[78] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[79] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Effective Scheduling of Duplicated Tasks for Fault Tolerance in Multiprocessor Systems," *IEICE Transaction on Information and Systems*, pp. 525–534, 2002.

[80] D. He and W. Mueller, "Online Energy-efficient Hard Real-time Scheduling for Component Oriented Systems," in *Proceedings of the IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '12)*. IEEE, 2012, pp. 56–63.

[81] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Fifth Edition*. Elsevier and Morgan Kaufmann Publishers, 2011.

[82] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, *Advanced Configuration and Power Interface Specification Rev. 5.0*, 2013.

[83] W. Hongxia and Q. Xin, "Dynamic Replication of Fault-Tolerant Scheduling Algorithm," *The Open Cybernetics & Systemics Journal*, pp. 2670–2676, 2015.

[84] C. U. Hönig, "Optimales Task-Graph-Scheduling für homogene und heterogene Zielsysteme," Ph.D. dissertation, FernUniversität in Hagen, 2008. [Online]. Available: https://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/forschung/berichte/bericht_342.pdf

[85] U. Hönig and W. Schiffmann, "Fast Optimal Task Graph Scheduling by Means of an Optimized Parallel A*-Algorithm." in *Proceedings of the 20th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '04)*, 2004, pp. 842–848.

[86] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele, "Energy Efficient DVFS Scheduling for Mixed-criticality Systems," in *Proceedings of the 14th International Conference on Embedded Software (EMSOFT '14)*. New York, NY, USA: ACM, 2014, pp. 11:1–11:10. [Online]. Available: http://doi.acm.org/10.1145/2656045.2656057

[87] S. Hunold, "Scheduling Moldable Tasks with Precedence Constraints and Arbitrary Speedup Functions on Multiprocessors," in *Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics (PPAM '13), Revised Selected Papers, Part II*. Springer Berlin

Heidelberg, 2014, pp. 13–25. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-55195-6_2

[88] IBM Corproation, *Blueprints: Using the Linux CPUFreq Subsystem for Energy Management*, 2009.

[89] IEA, International Energy Agency, "Energy Efficiency Indicators - Fundamentals on Statistics," 2014. [Online]. Available: /content/book/9789264215672-en

[90] E. Ilavarasan and P. Thambidurai, "Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments," *Journal of Computer Sciences*, vol. 3, no. 2, pp. 94–103, 2007.

[91] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide, Part 2*, 2016, vol. 3B.

[92] Intel Labs, "The SCC Platform Overview," May 2010.

[93] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling and Optimization of Fault-Tolerant Embedded Systems with Transparency/Performance Trade-Offs," *ACM Transactions of Embedded Computer Systems (TECS)*, vol. 11, no. 3, pp. 61:1–61:35, Sep. 2012.

[94] K. Jansen and H. Zhang, "An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints," *ACM Transactions on Algorithms (TALG)*, vol. 2, no. 3, pp. 416–434, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1159892.1159899

[95] S. Jayadivya, J. S. Nirmala, and M. S. S. Bhanu, "Fault-tolerant Workflow Scheduling Based on Replication and Resubmission of Tasks in Cloud Computing," *International Journal on Computer Science and Engineering (IJCSE)*, vol. 4, no. 6, p. 996, 2012.

[96] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-chips*. Elsevier, 2004.

[97] N. K. Jha, "Low Power System Scheduling and Synthesis," in *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD '01)*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 259–263. [Online]. Available: http://dl.acm.org/citation.cfm?id=603095.603147

[98] Z. Jun, E. H. Sha, Q. Zhuge, J. Yi, and K. Wu, "Efficient Fault-tolerant Scheduling on Multiprocessor Systems via Replication and Deallocation," *International Journal of Embedded Systems (IJES)*, vol. 6, no. 2-3, pp. 216–224, 2014.

[99] K. Kanoun, N. Mastronarde, D. Atienza, and M. van der Schaar, "Online Energy-Efficient Task-Graph Scheduling for Multicore Platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 8, pp. 1194–1207, Aug 2014.

[100] H. Kasahara, "Standard Task Graph Set," http://www.kasahara.elec.waseda.ac.jp/schedule/index.html, accessed: 2017-05-25.

[101] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 1023–1029, 1984.

[102] G. Kaur, "A DAG Based Task Scheduling Algorithms for Multiprocessor System-A Survey," *International Journal of Grid and Distributed Computing*, vol. 9, no. 9, pp. 103–114, 2016.

[103] R. Kaur and R. Kaur, "Multiprocessor Scheduling Using Task Duplication Based Scheduling Algorithms: A Review Paper," *International Journal of Application or Innovation in Engineering and Management (IJAIEM)*, vol. 2, no. 4, pp. 311–317, 2013.

[104] C. W. Kessler, N. Melot, P. Eitschberger, and J. Keller, "Crown scheduling: Energy-efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks," in *Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS '13)*, Sept 2013, pp. 215–222.

[105] V. Kianzad, S. Bhattacharyya, and G. Ou, "CASPER: An Integrated Energy-Driven Approach for Task Graph Scheduling on Distributed Embedded Systems." in *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '05)*, 2005, pp. 191–197.

[106] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Law Meets Static Power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec 2003.

[107] D. Kondo, H. Casanova, E. Wing, and F. Berman, "Models and Scheduling Mechanisms for Global Computing Applications," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002, pp. 8–16.

[108] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, vol. 5, no. 1, pp. 23–32, Jan. 1988. [Online]. Available: http://dx.doi.org/10.1109/52.1991

[109] Y.-K. Kwok, "Parallel Program Execution on a Heterogeneous PC Cluster Using Task Duplication," in *Proceedings of the 9th Heterogeneous Computing Workshop (HCW '00)*, 2000, pp. 364–374.

[110] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[111] D. Lee, D. Blaauw, and D. Sylvester, "Gate Oxide Leakage Current Analysis and Reduction for VLSI Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 155–166, 2004.

[112] W. Y. Lee, "Energy-Saving DVFS Scheduling of Multiple Periodic Real-Time Tasks on Multi-core Processors," in *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '09)*, Oct 2009, pp. 216–223.

[113] J. Lenhardt, "Energieeffiziente Verarbeitung fein granular verteilbarer Lasten auf heterogenen Rechnerverbünden," Ph.D. dissertation, Hagen, 2016. [Online]. Available: https://ub-deposit.fernuni-hagen.de/receive/mir_mods_00000548

[114] R. Lepére, D. Trystram, and G. J. Woeginger, "Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints," in *Proceedings of the 9th Annual European Symposium on Algorithms (ESA '01)*. Springer Berlin Heidelberg, 2001, pp. 146–157. [Online]. Available: http://dx.doi.org/10.1007/3-540-44676-1_12

[115] K. Li, "Performance Analysis of Power-Aware Task Scheduling Algorithms on Multiprocessor Computers with Dynamic Voltage and Speed," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 19, no. 11, pp. 1484–1497, Nov 2008.

[116] Y. Li, M. Chen, W. Dai, and M. Qiu, "Energy Optimization With Dynamic Task Scheduling Mobile Cloud Computing," *IEEE Systems Journal*, vol. 11, no. 1, pp. 96–105, 2017.

[117] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou, "Efficient Task Replication and Management for Adaptive Fault Tolerance in Mobile Grid Environments," *Future Generation Computer Systems*, vol. 23, no. 2, pp. 163–178, 2007.

[118] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[119] Y. Liu, K. Li, Z. Tang, and L. Keqin, "Energy Aware List-based Scheduling for Parallel Applications in Cloud," *Internationl Journal of Embedded Systems (IJES)*, 2015.

[120] R. Lopes and D. Menasce, "A Taxonomy of Job Scheduling on Distributed Computing Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 12, pp. 3412–3428, 2016.

[121] Y.-H. Lu, L. Benini, and G. De Micheli, "Low-power Task Scheduling for Multiple Devices," in *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES '00)*. New York, NY, USA: ACM, 2000, pp. 39–43. [Online]. Available: http://doi.acm.org/10.1145/334012.334020

[122] Y. Ma, B. Gong, and L. Zou, "Energy-efficient Scheduling Algorithm of Task Dependent Graph on DVS-Unable Cluster System," in *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GridCom '09)*, Oct 2009, pp. 217–224.

[123] ——, "Energy-Optimization Scheduling of Task Dependent Graph on DVS-Enabled Cluster System," in *Proceedings of the 5th Annual ChinaGrid Conference*, July 2010, pp. 183–190.

[124] B. S. Macey and A. Y. Zomaya, "A Performance Evaluation of CP List Scheduling Heuristics for Communication Intensive Task Graphs," in *Proceedings of the 12th International Parallel Processing Symposium (IPPS/SPDP '98)*. IEEE, 1998, pp. 538–541.

[125] G. Manimaran and C. S. R. Murthy, "An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 9, no. 3, pp. 312–319, 1998.

[126] N. Melot, C. Kessler, J. Keller, and P. Eitschberger, "Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems," *ACM Transactions on Architecture and Code Optimization (TACO '15)*, vol. 11, no. 4, pp. 62:1–62:24, Jan. 2015.

[127] Message Passing Interface Forum. (2012) MPI: A Message-Passing Interface Standard, Version 3.0. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[128] A. Mishra and A. K. Tripathi, "Energy Efficient Task Scheduling of Send-Receive Task Graphs on Distributed Multi-Core Processors with Software Controlled Dynamic Voltage Scaling," in *International Journal of Computer Science & Information Technology (IJCSIT)*, vol. 3, no. 2, 2011.

[129] R. Mishra, N. Rastogi, D. Zhu, D. Mosse, and R. Melhem, "Energy Aware Scheduling for Distributed Real-time Systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003, pp. 9 pp.–.

[130] M. Nandagopal and V. R. Uthariaraj, "Fault Tolerant Scheduling Strategy for Computational Grid Environment," *International Journal of Engineering Science and Technology (JESTEC)*, vol. 2, no. 9, pp. 4361–4372, 2010.

[131] B. Nazir and T. Khan, "Fault Tolerant Job Scheduling in Computational Grid," in *Proceedings of the International Conference on Emerging Technologies (ICET '06)*, Nov 2006, pp. 708–713.

[132] B. Nazir, K. Qureshi, and P. Manuel, "Adaptive Checkpointing Strategy to Tolerate Faults in Economy Based Grid," *The Journal of Supercomputing*, vol. 50, no. 1, pp. 1–18, 2009.

[133] L. Niu and G. Quan, "A Hybrid Static/Dynamic DVS Scheduling for Real-time Systems with (m, k)-guarantee," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*. IEEE, 2005, pp. 10–pp.

[134] V. G. Oklobdzija, *The Computer Engineering Handbook*. CRC Press, 2001.

[135] F. A. Omara and M. M. Arafa, "Genetic Algorithms for Task Scheduling Problem," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 13–22, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731509001804

[136] OpenMP Architecture Review Board. (2013) OpenMP Application Program Interface, Version 4.0 - July 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[137] B. K. Panigrahi, Y. Shi, and M.-H. Lim, *Handbook of Swarm Intelligence: Concepts, Principles and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[138] C. Papadimitriou and M. Yannakakis, "Towards an Architecture-independent Analysis of Parallel Algorithms," in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC '88)*. New York, NY, USA: ACM, 1988, pp. 510–513. [Online]. Available: http://doi.acm.org/10.1145/62212.62262

[139] B. Parhami, *Introduction to Parallel Processing*. Kluwer Academic Publishers, 1999.

[140] F. Pinel, B. Dorronsoro, J. E. Pecero, P. Bouvry, and S. U. Khan, "A Two-phase Heuristic for the Energy-efficient Scheduling of Independent Tasks on Computational Grids," *Cluster Computing*, vol. 16, no. 3, pp. 421–433, 2013.

[141] D. Poola, K. Ramamohanarao, and R. Buyya, "Fault-tolerant Workflow Scheduling Using Spot Instances on Clouds," *Procedia Computer Science*, vol. 29, pp. 523–533, 2014.

[142] ——, "Enhancing Reliability of Workflow Execution Using Task Replication and Spot Instances," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 4, p. 30, 2016.

[143] Z. Pooranian, M. Shojafar, R. Tavoli, M. Singhal, and A. Abraham, "A Hybrid Metaheuristic Algorithm for Job Scheduling on Computational Grids," *Informatica*, vol. 37, no. 2, p. 157, 2013.

[144] S. Porto and C. C. Ribeiro, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints," *International Journal of High Speed Computing*, vol. 07, no. 01, pp. 45–71, 1995. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S012905339500004X

[145] T. Prashar and D. Kumar, "Fault Tolerant ACO Using Checkpoint in Grid Computing," *International Journal of Computer Applications (IJCA)*, vol. 98, no. 10, 2014.

[146] S. B. Priya, M. Prakash, and K. Dhawan, "Fault Tolerance-Genetic Algorithm for Grid Task Scheduling Using Check Point," in *Proceedings of the 6th International Conference on Grid and Cooperative Computing (GCC '07)*. IEEE, 2007, pp. 676–680.

[147] K. Pruhs, R. van Stee, and P. Uthaisombut, "Speed Scaling of Tasks with Precedence Constraints," *Theory of Computing Systems*, vol. 43, no. 1, pp. 67–80, 2008.

[148] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Norwood, MA, USA: Artech House, Inc., 2001.

[149] S. Punnekkat, A. Burns, and R. Davis, "Analysis of Checkpointing for Real-time Systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[150] M. Rahman, S. Venugopal, and R. Buyya, "A Dynamic Critical Path Algorithm for Scheduling Scientific Workflow Applications on Global Grids," in *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*, Dec 2007, pp. 35–42.

[151] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-time Distributed Systems," *IEEE Software*, vol. 1, no. 3, p. 65, 1984.

[152] R.-D. Reiss, M. Thomas, and R. Reiss, *Statistical Analysis of Extreme Values*. Springer, 2007, vol. 2.

[153] N. B. Rizvandi and A. Y. Zomaya, "A Primarily Survey on Energy Efficiency in Cloud and Distributed Computing Systems," *Computing Research Repository (CoRR)*, vol. abs/1210.4690, 2012. [Online]. Available: http://arxiv.org/abs/1210.4690

[154] N. Romli, K. Minhad, M. M. I. Reaz, and M. S. Amin, "An Overview of Power Dissipation and Control Techniques in CMOS Technology," *Journal of Engineering Science and Technology (JESTEC)*, vol. 10, no. 3, pp. 364–382, 2015.

[155] P. Rong and M. Pedram, "Power-aware Scheduling and Dynamic Voltage Setting for Tasks Running on a Hard Real-time System," in *Proceedings of the 11th Asia and South Pacific design automation conference (ASP-DAC '06)*. IEEE Press, 2006, pp. 473–478.

[156] H. G. Rotithor, "Taxonomy of Dynamic Task Scheduling Schemes in Distributed Computing Systems," *IEE Proceedings - Computers and Digital Techniques*, vol. 141, no. 1, pp. 1–10, 1994.

[157] X. Ruan, X. Qin, Z. Zong, K. Bellam, and M. Nijim, "An Energy-Efficient Scheduling Algorithm Using Dynamic Voltage Scaling for Parallel Applications on Clusters," in *Proceedings of the 16th International Conference on Computer Communications and Networks (ICCCN '07)*, Aug 2007, pp. 735–740.

[158] R. A. Rutenbar, "Simulated Annealing Algorithms: An Overview," *IEEE Circuits and Devices Magazine*, vol. 5, no. 1, pp. 19–26, Jan 1989.

[159] H. F. Sheikh and I. Ahmad, "Dynamic Task Graph Scheduling on Multicore Processors for Performance, Energy, and Temperature Optimization," in *Proceedings of the 4th International Green Computing Conference (IGCC '13)*, June 2013, pp. 1–6.

[160] H. Shioda, K. Konishi, and S. Shin, "Optimal Task Scheduling Algorithm for Parallel Processing," in *Proceedings of the 2nd International Congress on Computer Applications and Computational Science (CACS '12)*, vol. 2. Springer Berlin Heidelberg, 2012, pp. 79–87. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28308-6_11

[161] M. Shojafar, S. Javanmardi, S. Abolfazli, and N. Cordeschi, "FUGE: A Joint Meta-heuristic Approach to Cloud Job Scheduling Algorithm Using Fuzzy

Theory and a Genetic Method," *Cluster Computing*, vol. 18, no. 2, pp. 829–844, 2015.

[162] J. Singh and N. Auluck, "DVFS and Duplication Based Scheduling for Optimizing Power and Performance in Heterogeneous Multiprocessors," in *Proceedings of the High Performance Computing Symposium*, ser. HPC '14, 2014, pp. 22:1–22:8.

[163] M. Singh, "Incremental Checkpoint Based Failure-aware Scheduling Algorithm in Grid Computing," in *Proceedings of the International Conference on Computing, Communication and Automation (ICCCA '16)*. IEEE, 2016, pp. 772–778.

[164] O. Sinnen, *Task Scheduling for Parallel Systems*. John Wiley & Sons, 2007.

[165] A. M. Sllame and V. Drabek, "An Efficient List-based Scheduling Algorithm for High-level Synthesis," in *Proceedings of the Euromicro Symposium on Digital System Design (DSD '02)*. IEEE, 2002, pp. 316–323.

[166] M. Stanisavljević, A. Schmid, and Y. Leblebici, *Reliability of Nanoscale Circuits and Systems: Methodologies and Circuit Architectures*. Springer Science & Business Media, 2010.

[167] J. Stoll, *Fehlertoleranz in verteilten Realzeitsystemen: Anwendungsorientierte Techniken*, ser. Informatik-Fachberichte. Springer-Verlag, 1990. [Online]. Available: https://books.google.de/books?id=fSq4AAAAIAAJ

[168] V. Subramani, R. Kettimuthu, S. Srinivasan, and S. Sadayappan, "Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC '02)*. IEEE, 2002, pp. 359–366.

[169] N. Tabba, R. Entezari-Maleki, and A. Movaghar, "Reduced Communications Fault Tolerant Task Scheduling Algorithm for Multiprocessor Systems," *Procedia Engineering*, vol. 29, pp. 3820–3825, 2012, 2012 International Workshop on Information and Electronics Engineering.

[170] I. Takouna, W. Dawoud, and C. Meinel, "Accurate Mutlicore Processor Power Models for Power-Aware Resource Management," in *Proceedings of the IEEE 9th International Conference on Dependable, Autonomic and Secure Computing (DASC '11)*, Dec 2011, pp. 419–426.

[171] A. S. Tanenbaum, *Structured Computer Organization, Fifth Edition.* Pearson Prentice Hall, 2006.

[172] ——, *Modern Operating Systems.* Pearson Education, 2009.

[173] J. Thaman and M. Singh, "Current Perspective in Task Scheduling Techniques in Cloud Computing: A Review," *International Journal in Foundations of Computer Science & Technology*, vol. 6, pp. 65–85, 2016.

[174] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[175] S. Tosun, N. Mansouri, M. Kandemir, and O. Ozturk, "An ILP Formulation for Task Scheduling on Heterogeneous Chip Multiprocessors," in *Proceedings of the 21st International Symposium on Computer and Information Sciences (ISCIS '06).* Springer Berlin Heidelberg, 2006, pp. 267–276.

[176] M. Treaster, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems," *Computing Research Repository (CoRR)*, 2005. [Online]. Available: http://arxiv.org/abs/cs/0501002

[177] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-tolerant Scheduling Technique for Real-time Multiprocessor Systems," in *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications (RTCSA '95).* IEEE, 1995, pp. 197–202.

[178] University of Tennessee, "Power API," http://icl.cs.utk.edu/projects/papi/wiki/Main_Page, accessed: 2017-03-21.

[179] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/-Software Introduction.* New York: Wiley, 2002.

[180] L. Wang, G. von Laszewski, J. Daya, and F. Wang, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*, May 2010, pp. 368–377.

[181] Y. Wang, D. Liu, M. Wang, Z. Qin, and Z. Shao, "Optimal Task Scheduling by Removing Inter-Core Communication Overhead for Streaming Applications

on MPSoC," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*, April 2010, pp. 195–204.

[182] J. B. Weissman and D. Womack, *Fault Tolerant Scheduling in Distributed Networks*. Division of Computer Science, the University of Texas [at] San Antonio, 1996.

[183] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[184] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[185] 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*, March 2005, pp. 468–473 Vol. 1.

[186] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," in *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS '91)*. New York, NY, USA: ACM, 1991, pp. 633–642. [Online]. Available: http://doi.acm.org/10.1145/125826.126138

[187] S. Yi, D. Kondo, B. Kim, G. Park, and Y. Cho, "Using Replication and Checkpointing for Reliable Task Management in Computational Grids," in *Proceedings of the International Conference on High Performance Computing Simulation (HPCS '10)*, June 2010, pp. 125–131.

[188] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job Scheduling for Multi-user Mapreduce Clusters," *Journal Electrical Engineering and Computer Siences (EECS)*, 2009.

[189] 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 *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC '11)*, Sept 2011, pp. 187–194.

[190] H. Zhang and H. Hoffman, "A Quantitative Evaluation of the RAPL Power Control System," *Feedback Computing*, 2015.

[191] Y. Zhang, C. Koelbel, and K. Cooper, "Hybrid Re-scheduling Mechanisms for Workflow Applications on Multi-cluster Grid," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, May 2009, pp. 116–123.

[192] Y. Zhang, Y. Inoguchi, and H. Shen, "A Dynamic Task Scheduling Algorithm for Grid Computing System," in *Proceedings of the 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA '04)*, J. Cao, L. T. Yang, M. Guo, and F. Lau, Eds. Springer Berlin Heidelberg, 2005, pp. 578–583. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30566-8_69

[193] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, "Fault-tolerant Scheduling with Dynamic Number of Replicas in Heterogeneous Systems," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC '10)*, Sept 2010, pp. 434–441.

[194] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Energy-cognizant Scheduling Techniques," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 7, pp. 1447–1464, 2013.

Curriculum Vitae

Personal Data

Name:	Patrick Eitschberger (né Cichowski)
Birth:	Dorsten, Germany — 26 August 1982
Address:	Thielenstr. 30, 44369 Dortmund, Germany

Academic Career and Education

Since 2012	**Doctoral Student**, Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Germany
Okt 2009 - Dec 2011	**Master of Computer Science**, FernUniversität in Hagen, Germany Specialization: Computing Systems
Okt 2003 - Aug 2009	**Diplom Informatiker (Diplom I)**, FernUniversität in Hagen, Germany Minor Subject: Business Studies
Aug 1999 - Jun 2002	**Assistant Tax Consultant**, Tax Consultancy W. Cichowski, Dorsten, Germany

Employments

Since Jan 2012	FernUniversität in Hagen, Faculty of Mathematics and Computer Science, Parallelism & VLSI Group, Hagen, Germany Research Assistant, full-time
Aug 2009 - Dez 2011	FernUniversität in Hagen, Faculty of Mathematics and Computer Science, Computer Architecture Group, Hagen, Germany Research Assistant, part-time while studying
Nov 2005 - May 2009	Shell-Station Sondram & Taoka GmbH, Dortmund, Germany Cashier, part-time while studying
Jul 2003 - Jun 2005	Tax Consultancy W. Cichowski, Dorsten, Germany Assistant Tax Consultant, part-time while studying
Jul 2002 - Aug 2002	Tax Consultancy W. Cichowski, Dorsten, Germany Assistant Tax Consultant, full-time