THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Runtime Management of Multiprocessor Systems for Fault Tolerance, Energy Efficiency and Load Balancing

STAVROS TZILIS

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

Runtime Management of Multiprocessor Systems
for Fault Tolerance, Energy Efficiency and Load Balancing
Stavros Tzilis
Göteborg, Sweden, 2019
ISBN: 978-91-7597-878-9

Ioannis Sourdis	Advisor	Professor at Chalmers University of Technology
Pedro P. M. Trancoso	Co-Advisor	Senior Lecturer at Chalmers University of Technology
Miquel Pericas	Co-Advisor	Associate Professor at Chalmers University of Technology
Luigi Carro	Thesis Opponent	Professor at Universidade Federal do Rio Grande do Sul
Cristiana Bolchini	Grading Committee	Professor at Politecnico di Milano
Sven Karlsson	Grading Committee	Ericsson Research
Andy D. Pimentel	Grading Committee	Associate Professor at University of Amsterdam

Copyright © Stavros Tzilis, 2019.

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie Nr 4559
ISSN 0346-718X

Technical Report No. 171D
Department of Computer Science and Engineering
Chalmers University of Technology

Contact Information:

Division of Computer Engineering
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Telephone: +46 (0)31-772 10 00
`http://www.chalmers.se/cse/`

Author's e-mail: `tzilis@chalmers.se`

Printed by Chalmers Reproservice
GÖTEBORG, Sweden 2019

This one's for Stavros,
whom I might have neglected during these seven years

Runtime Management of Multiprocessor Systems for Fault Tolerance, Energy Efficiency and Load Balancing

Stavros Tzilis
Department of Computer Science and Engineering, Chalmers University of Technology, Sweden

Abstract

Efficiency of modern multiprocessor systems is hurt by unpredictable events: aging causes permanent faults that disable components; application spawnings and terminations taking place at arbitrary times, affect energy proportionality, causing energy waste; load imbalances reduce resource utilization, penalizing performance. This thesis demonstrates how runtime management can mitigate the negative effects of unpredictable events, making decisions guided by a combination of static information known in advance and parameters that only become known at runtime. We propose techniques for three different objectives: graceful degradation of aging-prone systems; energy efficiency of heterogeneous adaptive systems; and load balancing by means of work stealing. Managing aging-prone systems for graceful efficiency degradation, is based on a high-level system description that encapsulates hardware reconfigurability and workload flexibility and allows to quantify system efficiency and use it as an objective function. Different custom heuristics, as well as simulated annealing and a genetic algorithm are proposed to optimize this objective function as a response to component failures. Custom heuristics are one to two orders of magnitude faster, provide better efficiency for the first 20% of system lifetime and are less than 13% worse than a genetic algorithm at the end of this lifetime. Custom heuristics occasionally fail to satisfy reconfiguration cost constraints. As all algorithms' execution time scales well with respect to system size, a genetic algorithm can be used as backup in these cases. Managing heterogeneous multiprocessors capable of Dynamic Voltage and Frequency Scaling is based on a model that accurately predicts performance and power: performance is predicted by combining static, application-specific profiling information and dynamic, runtime performance monitoring data; power is predicted using the aforementioned performance estimations and a set of system-specific, static parameters, determined only once and used for every application mix. Three runtime heuristics are proposed, that make use of this model to perform partial search of the configuration space, evaluating a small set of configurations and selecting the best one. When best-effort performance is adequate, the proposed approach achieves 3% higher energy efficiency compared to the powersave governor and $2\times$ better compared to the interactive and ondemand governors. When individual applications' performance requirements are considered, the proposed approach is able to satisfy them, giving away 18% of system's energy efficiency compared to the powersave, which however misses the performance targets by 23%; at the same time, the proposed approach maintains an efficiency advantage of about 55% compared to the other governors, which also satisfy the requirements. Lastly, to improve load balancing of multiprocessors, a partial and approximate view of the current load distribution among system cores is proposed, which consists of lightweight data structures and is maintained by each core through cheap operations. A runtime algorithm is developed, using this view whenever a core becomes idle, to perform victim core selection for work stealing, also considering system topology and memory hierarchy. Among 12 diverse imbalanced workloads, the proposed approach achieves better performance than random, hierarchical and local stealing for six workloads. Furthermore, it is at most 8% slower among the other six workloads, while competing strategies incur a penalty of at least 89% on some workload.

Keywords: Multiprocessors, Runtime Management, Adaptive Systems, Algorithms, Fault Tolerance, Energy Efficiency, Performance, Load Balancing

List of Publications

Parts of the contributions presented in this thesis have also been published in the following manuscripts, listed by chapter.

Chapter 2:

▷ **Stavros Tzilis**, Ioannis Sourdis, “A Runtime Manager for Gracefully Degrading SoCs”, *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October, 2014, pp. 216-221.

▷ Alirad Malek, **Stavros Tzilis**, Danish Anis Khan, Ioannis Sourdis, Georgios Smaragdos, Christos Strydis, “A probabilistic analysis of resilient reconfigurable designs”, *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October, 2014, pp. 141-146.

▷ Georgios Smaragdos, Danish Anis Khan, Ioannis Sourdis, Christos Strydis, Alirad Malek, **Stavros Tzilis**, “A Dependable Coarse-grain Reconfigurable Multicore Array”, in 21st Reconfigurable Architectures Workshop (RAW), May, 2014, pp. 141-150.

▷ Dimitris Theodoropoulos, Dionisios Pnevmatikatos, **Stavros Tzilis**, Ioannis Sourdis, “The DeSyRe Runtime support for Fault-tolerant Embedded MPSoCs”, *12th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, August, 2014, pp. 197-204.

▷ Alirad Malek, **Stavros Tzilis**, Danish Anis Khan, Ioannis Sourdis, Georgios Smaragdos, Christos Strydis, “Reducing the performance overhead of resilient CMPs with substitutable resources”, *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October, 2015, pp. 191-196.

▷ Ioannis Sourdis, Danish Anis Khan, Alirad Malek, **Stavros Tzilis**, Georgios Smaragdos, Christos Strydis, “Resilient chip multiprocessors with mixed-grained reconfigurability”, IEEE Micro. January 2016; 36(1):35-45.

▷ **Stavros Tzilis**, Ioannis Sourdis, Vasileios Vasilikos, Dimitrios Rodopoulos, Dimitrios Soudris, “Runtime Management of Adaptive MPSoCs for Graceful Degradation”, *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, October, 2016, pp. 5:1-5:10.

Chapter 3:

▷ **Stavros Tzilis**, Pedro Trancoso, Ioannis Sourdis, “Energy-efficient Runtime Management of Heterogeneous Multicores using Online Projection”, *ACM Transactions on Architecture and Code Optimization (TACO)*, December, 2018.

Chapter 4:

▷ **Stavros Tzilis**, Miquel Pericas, Pedro Trancoso, Ioannis Sourdis, “SWAS: Stealing Work using Approximate System-load information”, *13th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS), 46th International Conference on Parallel Processing Workshops (ICPPW)*, August, 2017, pp. 309-318.

The following papers are not directly related to this thesis:

▷ I. Sourdis, C. Strydis, A. Armato, C.-S. Bouganis, B. Falsafi, G. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D.N. Pnevmatikatos, D.K Pradhan, G. Rauwerda, R. Seepers, R.K. Shafik, K. Sunesen, D. Theodoropoulos, **S. Tzilis**, M. Vavouras, “DeSyRe: on-Demand System Reliability”, in Elsevier Microprocessors and Microsystems, Special Issue on European Projects in Embedded System Design, November, 2013.

▷ I. Sourdis, C. Strydis, A. Armato, C.S. Bouganis, B. Falsafi, G.N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D.K. Pradhan, G. Rauwerda, R.M. Seepers, R.A. Shafik, G. Smaragdos, D. Theodoropoulos, **S. Tzilis**, M. Vavouras, S. Pagliarini, and D. Pnevmatikatos, “DeSyRe: On-Demand Adaptive and Reconfigurable Fault-Tolerant SoCs ”, in 10th Int’l Symp. on Applied Reconfigurable Computing (ARC), pp. 312-317, 2014.

▷ Alirad Malek, Ioannis Sourdis, **Stavros Tzilis**, Yifan He, Gerard Rauwerda, “RQNoC: A Resilient Quality-of-Service Network-on-Chip with Service Redirection”, *ACM Transactions on Embedded Computing Systems (TECS)*, 15, no. 2 (2016): 28.

Acknowledgments

PhD is a challenging and lonely undertaking. And while -as you very well know is the case if you have been close to me- it proved to be especially challenging for me, the loneliness aspect was, to some degree, mitigated by the multitude of people who, although could not share the burden per se, tirelessly stood by the sideline and encouraged me to keep fighting.

I want to express my deepest gratitude to my supervisor, Ioannis Sourdis. It was neither easy nor completely smooth, as we simultaneously started exploring uncharted territory: it was my first time doing a PhD (it usually is) and his first time supervising a PhD candidate from start to finish and, on top of this, his first time leading a research project. Even when my progress was very slow, he didn't flinch and doubled down on me, putting his faith in the scenario that I would pick up speed and get the thing done. Through this painstaking process we both grew a lot against our respective new challenges. I would not have done it without your support.

I would like to extend a warm thanks to the co-supervisors that helped with parts of the work: Pedro Trancoso and Miquel Pericás. Your valuable insights made it possible to finish in a reasonable timeline. Also, I would like to thank Georgi Gaydadjiev, both for contacting me at the right moment, as a result of which I am here today and for doing his part in guiding me through the initial stages.

I am very grateful to the whole organization of Chalmers University of Technology and in particular the Department of Computer Science and Engineering. The extra support I received, enabling me to push through, is not to be taken for granted. I want you all to know that it is greatly appreciated.

Next, I want to thank all my colleagues, among them several good friends. The people with whom I had a nice discussion, who offered a warm greeting, who would take breaks with me, in general everybody who made this work environment alive and pleasant. More importantly, the ones with whom I shared after-work board games as well as the ones who were there to pick me up when things looked the darkest. A special thanks, of course, is reserved for the best office mates one could ever ask for, Alirad and Ahsen.

Dear friends in Gothenburg, you know who you are and how essential your presence was. I will never forget a single thing you have done for me. You are one of the main reasons that, looking back, the whole experience feels worthwhile.

My friends from before starting this endeavour, who reside all over the world: thank you for being in the background, a solace in bad times, a source of happiness in good times and a reminder of a long road behind me whenever we had the rare chance to get together. Also, a sincere apology to the ones of you from whom I drifted apart. I hope we get together again really soon. Among you, a special thanks to Stefanos Katmadas, possibly my only everyday contact, the guy that would be the first to know if something extraordinary, good or bad, happened to me.

Mom Lena, dad Prokopis and sis Maria: I love you and hope you are happy. Happy for me and happy in general.

This work has been partially funded by the European Union's Framework Program 7 DeSyRe project (grant agreement 287611), the EMC2 project (grant agreement 621429), the European Research Council (ERC) under the MECCA project (grant agreement 340328) and the Horizon 2020 Program ECOSCALE project (grant agreement 671632).

I would like to thank the Institute of Communication and Computer Systems (ICCS), associated with the School of Electrical and Computer Engineering (ECE) of the National Technical University of Athens (NTUA), Greece for enabling experimental verification of parts of the work presented in this thesis, on the Single-Chip Cloud Computer (SCC). The SCC was made available under aMaterial Transfer Agreement between ICCS/NTUA and Intel Corporation (Amended April 23, 2013 - Principal Investigator: Dimitrios Rodopoulos).

Many thanks to all administrative staff of the Computer Science and Engineering Department for attending to my every need throughout the whole PhD.

Here's to the best still to come! Cordially,

Stavros Tzilis
Göteborg, February 2019.

Contents

List of Figures

List of Tables

1 Introduction

Advances in semiconductor technology have created unprecedented efficiency potential for computing systems. As feature sizes shrink, more components can be packed on the same chip and more computations can be performed per unit of time. However, utilizing this potential involves numerous challenges. Failure rates become higher, disabling components or whole systems relatively early in their lifetime [1]. The total power that can be supplied to a chip is limited, forcing underutilization of resources [2]. Additionally, without proper task distribution, workloads are often not able to make use of the available parallelism, due to load imbalances [3].

A common characteristic of the above challenges is that they involve parameters the values of which only become known at runtime, when the problem appears. For instance, providing efficient fault tolerance requires knowledge of the type, location and time of failures. Providing efficient power management and resource utilization requires, at any given time, knowledge of the current workload requirements. This thesis aims to provide runtime solutions for this type of challenges, suggesting strategies for responding to unpredictable events, such as component failures and workload changes.

This introductory chapter provides an overview of the work presented in the thesis. The remainder of this chapter is organized as follows: Sections 1.1 and 1.2 present the problem statement and thesis statement respectively. Section 1.3 sets the main thesis objectives. Section 1.4 summarizes the main contributions and finally, Section 1.5 provides an outline for the rest of this thesis.

1.1 Problem Statement: Unpredictable Events Penalize Efficiency

Unpredictable events cause modern *Multiprocessor Systems* to lose efficiency, if not properly dealt with. Some examples of such unpredictable events are:

- **Component failures:** Conventionally, a component failure would mean that the whole system is disabled. A modern system with more components and elevated failure rates [1], cannot afford such a naïve approach, thus fault isolation and reconfiguration techniques are used to keep the system running. However, the proper reconfiguration choice depends on factors such as the location, type and time of failure, that only become known when the failure actually happens. This unpredictability creates the need for a response at runtime.

- **Application spawns/terminations:** Many modern multiprocessors offer flexibility by means of heterogeneity and *Dynamic Voltage and Frequency Scaling (DVFS)*. As a result, their performance, power and energy efficiency can be dynamically managed. The optimal choices depend on the number, type and requirements of the applications that constitute the workload. As individual applications start and terminate at arbitrary times (e.g., on a hand-held portable device), they create unpredictable events, requiring changes at runtime to maintain system efficiency.

- **Workload imbalances:** The efficiency of a parallel system is highly dependent on the degree of utilization of its resources. Workloads are often imbalanced by nature, making it hard to utilize all available cores constantly. A core becoming idle while other cores are overloaded is an event leading to resource waste, unless the workload is (re)balanced dynamically.

Whichever the optimization objective of a system is (which can be generically called *system efficiency*), events such as the ones described above can, if not dealt with, heavily penalize it. As will be demonstrated throughout Chapters 2 to 4, the occurrence of any of the above events creates optimization decisions, the decision space of which is too large to be searched at runtime. In conclusion, we need sophisticated strategies to deal with the dynamic changes happening on modern multiprocessors. The upcoming sections, 1.2 and 1.3 outline more specifically the desired qualities of such sophisticated strategies.

1.2 Thesis Statement: Runtime Management can provide Efficient Responses to Unpredictable Events

In this thesis, we study *the design of runtime managers that respond to unpredictable events, aiming to optimize for a relevant metric*. To design such managers, we combine static information that is known in advance, with unpredictable information that only becomes available at the time of an event. We aim to demonstrate that:

Runtime management can produce fast responses to unpredictable events, diminishing the negative impact of these events on system efficiency.

By “fast responses”, we mean that the time needed to decide on a reaction has to be short compared to the intervals between events and also scalable regarding system parameters, e.g. its asymptotic complexity should not be exponential with respect to system size. It is important, for example, that a system is up and running again as quickly as possible after a component failure. “System efficiency” is used at this point as a generic term, representing the relevant optimization objective of a given system.

Ultimately, all runtime management techniques presented in this thesis aim to balance decision quality (high system efficiency) and low overheads, such as response time. We tackle this challenge in three different contexts:

- Fault-prone multiprocessors with reconfiguration capabilities that enable fault isolation.
- Multiprocessors that enable energy efficiency management by means of heterogeneity and DVFS.
- Parallel systems running imbalanced workloads, requiring dynamic load balancing.

The next section outlines the specific objectives associated with each of these three contexts.

1.3 Thesis Objectives

In this section, we focus on each of the three different types of systems we have designed runtime managers for and provide an overview of the goals of each runtime manager.

1.3.1 Runtime Management for Graceful Degradation

In the first part of this thesis (see Chapter 2) we study multiprocessor systems with hardware reconfiguration capabilities that enable isolation of faulty components. The main objective of this study is:

to design a runtime algorithm that reacts to failures of individual components and decides on a new system configuration that provides graceful efficiency degradation with low overheads.

To elaborate on the above statement, we consider systems built on multiple, possibly heterogeneous, computational components (i.e. microprocessors), each of which suffers permanent faults independently. When a permanent fault occurs, it is possible that the affected component can continue to function in a degraded mode. Furthermore, some of the components support multiple degraded modes, each of which entails different

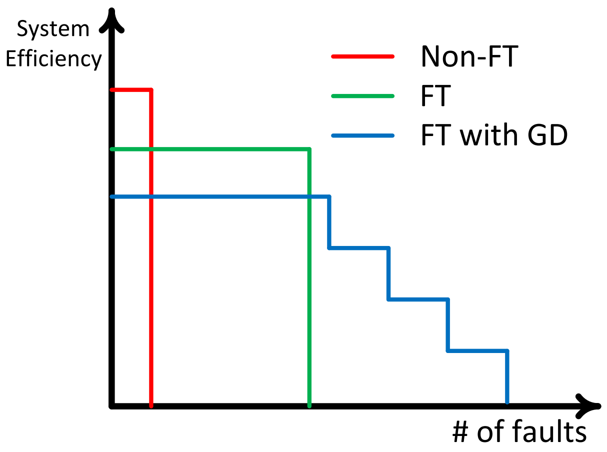


Figure 1.1: Conceptualization of the Graceful Degradation property. A conventional, fault intolerant system (Non-FT) would stop working when a permanent fault appeared. Fault tolerance (FT) sacrifices some fault-free efficiency, to keep the system running in the presence of faults. Graceful Degradation (GD) allows sacrificing, at runtime, additional efficiency, to tolerate even more faults.

penalties, creating interesting tradeoffs (e.g. degradation of either performance or energy efficiency).

Upon the occurrence of a *component failure* event, our runtime manager has to choose the appropriate configuration for the component on which the failure occurred and possibly change the configuration of other system components as well. At the same time, it has to be able to utilize possible flexibility of the workload, e.g., to decide that non-essential tasks are not performed to assure the completion of essential tasks on the remaining hardware resources. Lastly, it should choose the allocation of tasks to components utilizing heterogeneity, which is either provided by having different component types or imposed by the coexistence of fault-free and damaged components.

The concept of Graceful Degradation, as a special case of fault tolerance, is illustrated in Figure 1.1. Graceful Degradation allows a system to dynamically trade efficiency to tolerate faults and keep the system running in degraded states for as long as possible.

System efficiency in this context is defined by means of an objective function that the runtime manager aims to maximize, considering performance, energy and functionality (whether or not non-essential functions are being carried out). Alongside maximizing efficiency, the runtime manager also has to minimize the time and energy overheads of applying the decided changes to the system. Briefly, the main challenges tackled in this part of the thesis are:

- Develop a methodology for characterizing degradable hardware components, in terms of performance and energy efficiency of their degraded configurations.
- Formally describe a system that has the property of graceful degradation, in such a way that it is possible to systematically enumerate all possible system configurations and to quantify its overall efficiency in terms of performance, energy and functionality.
- Design algorithms that, triggered by a *component failure* event, quickly decide on a series of changes to be applied on the system, aiming to minimize the resulting efficiency loss as well as the overheads of applying the changes.

Related Work: There exists extensive research about enabling systems to degrade gracefully. This is achieved either by isolating faulty substitutable units at core [4–8], pipeline stage [9–11], or gate level [12] granularity, or by using alternative implementations of the workload at the algorithm [13] or instruction [14–16] level. When it comes to managing this potential, many researchers propose static, determined-in-advance strategies, such as banking [17] or pooling [18] of redundant components, or downgrading tasks from normal execution to software emulation or even to execution on slower, fine-grain reconfigurable fabric [19]. Moving on to more dynamic, runtime approaches, MAESTRO [20] utilizes microsensors to perform wearout-aware mapping of tasks. Baldassari et al. expose knobs for plugging in user-defined reliability policies [21]. The outcome of some common policies with respect to system lifetime is comparatively presented in [22]. The trade-off between tolerance of permanent and transient faults on fail-operational systems has been also explored in [23]. Dynamic management for reliability has also been combined with power awareness, in works that employ task mapping to jointly optimize the two goals [24, 25].

Compared to previous work, we address the need for responding to failure scenarios that cannot be predicted entirely at design time, allowing sequences of multiple component failures, each of them being of different type and severity. Furthermore, we expose and utilize both hardware and workload flexibility, according to what has been demonstrated in literature: we allow individual components to degrade independently and we introduce system functionality as a knob for graceful degradation decisions.

1.3.2 Runtime Management for Energy Efficiency

In the second part of the thesis (see Chapter 3), we study multiprocessor systems that offer flexibility by means of heterogeneity and DVFS. The main objective of this study is:

to design a runtime manager that reacts to application spawns or terminations on a heterogeneous, DVFS-capable multiprocessor and maximizes energy efficiency.

More specifically, the runtime manager has to take energy-efficient decisions, each time a new application spawns for execution, as well as each time an application

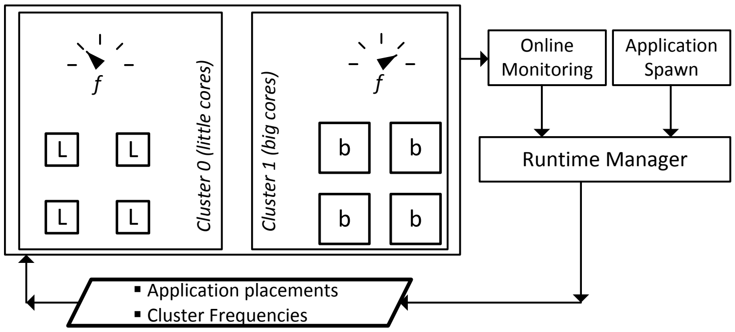


Figure 1.2: Runtime management for heterogeneous, DVFS-capable multiprocessors. Triggered by an *application spawn* event and guided by online monitoring information, the runtime manager decides application placements and cluster frequencies.

terminates. "Decisions" in this context include the choice of core that each application will execute on (application placement/allocation) and the choice of frequency for each independent cluster of cores (DVFS). For this work, the optimization objective is energy efficiency, measured in *Instructions per Second per Watt (IPS/Watt).* At the same time, individual applications might come with their own minimum performance requirements, which have to be satisfied. Figure 1.2 shows the role of the runtime manager in such a system.

To guide the aforementioned decisions, the runtime manager has to be able to predict the outcome of each possible choice. In other words, it requires the ability to accurately estimate applications' performance and system power as a result of the chosen application placements and cluster frequencies. Achieving this is especially challenging, considering that the runtime manager has to govern a system in a dynamic and unpredictable scenario, where multiple applications with different performance characteristics run concurrently, competing for shared resources. Additionally, as the evaluation of all possible choices in runtime is infeasible, the runtime manager needs to execute a low-complexity algorithm that narrows down the search, by selecting a low number of candidate solutions for evaluation.

The above challenges can be summarized in the following points:

- Design a prediction model able to accurately estimate applications' performance and system power, as a result of any possible combination of application placements and cluster frequencies.
- Design an algorithm that is triggered by an *application spawn* or *application termination* event and is able to choose the solution that maximizes energy efficiency while satisfying performance requirements of individual applications.

Related Work: Runtime management for heterogeneous, DVFS-capable multiprocessors is a very well researched topic. A number of works either explore the potential of using such platforms for energy efficiency optimization [26, 27], or utilize this potential to optimize a specific narrow application domain, such as video games or datacenters [28–30]. Both sets of works point to the conclusion that heterogeneous architectures offer great potential for performance and energy efficiency, which is not fully utilized due to the difficulty of choosing optimal configurations.

Works that propose approaches to utilize this potential, differ with respect to various aspects. When it comes to energy-saving techniques, some works employ a combination of selecting cluster frequencies with *Dynamic Voltage and Frequency Scaling (DVFS)* and selecting which system components are active or inactive with *Dynamic Power Management (DPM)*, in order to optimize single applications for standalone execution [31–35]. On the other hand, there exist works applicable to dynamic, unpredictable scenarios of multiple applications running concurrently [36, 37]. Examining this more interesting scenario complicates the problem significantly, because, among other reasons, it adds the decision of *allocating* applications to cores for execution. Thus, the algorithms proposed by these works are at least quadratic in complexity, resulting in the time needed for allocation decisions to explode as system size increases.

Approaches also differ with respect to the information utilized to make decisions. Some choose to characterize applications offline [31] but do not propose a strategy to keep this profiling stage feasible for arbitrarily large systems. Others are guided only by online monitoring data, sacrificing the potential to make better decisions aided by profiling [33, 36, 37]. Alongside the offline and/or online performance measurements, almost all approaches use a prediction model. Some models involve application-specific parameters, resulting to the need for retraining (either online or offline) every time the application mix changes [32, 33]. Others are based on categorizing applications with respect to criteria, such as compute intensiveness and memory boundedness, resulting to a less sensitive but also less accurate prediction [37, 38].

Based on the above, we utilize both offline profiling and online monitoring data, taking care to keep offline profiling feasible. We also develop a power model that only needs to be calibrated once for a given platform. Thus, compared to previous work, we perform predictions in an accurate, consistent and robust manner. Additionally, when it comes to runtime decision making, we propose algorithms with lower complexity. As a result, the response time of our runtime manager to events is both short and scalable with respect to system size.

1.3.3 Runtime Management for Load Balancing

In the third and final part of this thesis (see Chapter 4) we study multi- and many-core systems running parallel workloads, the performance of which can suffer from poor load balance. The main objective of this work is:

to design a runtime strategy that dynamically performs load balancing, to maximize overall performance.

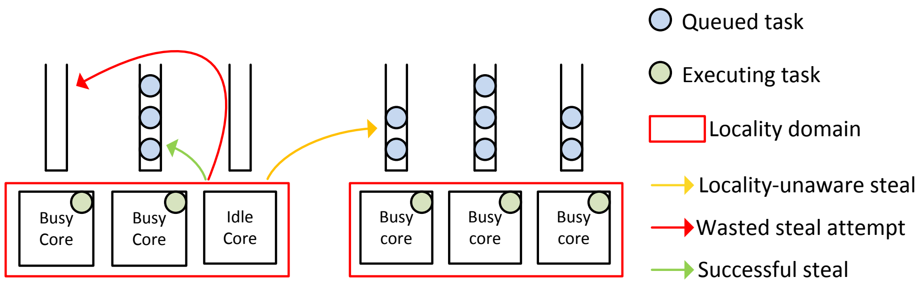


Figure 1.3: System load and locality aware work stealing. Lack of system load awareness results to wasted steal attempts. Not preferring local over remote steals can destroy data locality.

The event that needs to be responded to in this case, is a core becoming idle, that is, having no tasks available to process while other cores still have work. This situation reduces the utilized parallelism (ideally all cores should execute tasks in parallel) and, as such, reduces overall performance, prolonging execution time. In order to dynamically balance the workload and maximize utilization, the technique of *Work Stealing* has been proposed, that is, allowing idle cores to execute the work that was originally assigned to currently busy cores [39]. For Work Stealing to provide performance benefits, two factors have to be addressed: the core chosen to steal work from should currently be one of the system's most loaded cores and the overhead of migrating the task and its working set to the idle core should be as low as possible. Both of these factors' impact on performance is shown in Figure 1.3.

Thus, in the event of a core becoming idle, our strategy needs to guide work stealing decisions considering the above factors. The main challenge we face is that of efficiently selecting a busy core for stealing work from. Indeed, as system sizes increase, the cost of naïvely probing other cores randomly until work is found, increases as well. Thus, an efficient strategy needs to acquire this information in a cheap manner, so that the work stealing routines do not penalize the performance of the actual application. Subsequently, based on this information, work should be stolen in such a way that an imbalanced workload will benefit, while a balanced workload will not be penalized due to task migration overheads. In summary, in Chapter 4, we tackle the following challenges:

- Devise a strategy for each core in a multiprocessor, to acquire information about the current distribution of workload in the system.
- Use the above information to guide work stealing decisions whenever a *core idle* event happens, in a way that improves the performance of imbalanced workloads and does not hurt the performance of balanced ones.

Related Work: Work stealing has been a popular technique for dynamic load balancing [39] and as such, it is implemented in different ways in existing literature.

Random target selection [40] can yield statistical performance gains on average, but has inconsistent results and can even penalize performance by destroying data locality. Restricting target selection to neighbouring cores [41] diminishes this effect, but cannot deal with system-wide load imbalances. However, there exist more sophisticated versions of hierarchical stealing [42, 43], that have inspired us to incorporate this element in our work. Some approaches use compile-time hints to preserve locality [44–47]. This is beneficial for performance, but makes the strategy application-dependent. Making more educated target core choices can be achieved by consulting the current system load distribution, but we found this to entail unacceptable overheads, an observation made by other researchers as well [48]. We develop a technique that is aware of locality, hierarchy and system load information. Our main focus is on making the latter feasible through a lightweight implementation: an approximate view of the current system load, maintained and enriched during the necessary load querying step.

1.4 Contributions

According to the objectives outlined in Section 1.3, this section summarizes the main contributions of each part of the thesis. Section 1.4.1 focuses on optimizations for graceful degradation, Section 1.4.2 on optimizations for energy efficiency and Section 1.4.3 on load balancing.

1.4.1 Optimizing Fault Tolerant Multicores for Graceful Degradation

In Chapter 2, we present our approach to managing fault tolerant, reconfigurable multiprocessor systems in the presence of successive permanent faults that disable parts of the system. Our goal is to allow efficiency to degrade as gracefully as possible with each fault, at the same time minimizing the one-time overheads of reconfiguration. Our contributions can be summarized as follows:

> We come up with a formal description of a system that consists of heterogeneous, reconfigurable, fault-tolerant components running flexible workloads. We express each component and task as a set of mutually exclusive alternatives and annotate each component and task alternative with attributes that describe its contribution to overall system performance, energy consumption and functionality. Subsequently, we design algorithms that are triggered by a *component failure* event and choose one of the mutually exclusive options for each component and task, as well as an allocation of tasks to components. The algorithms are based on the concepts of incremental changes, i.e. determining a series of small changes that need to be performed on the system, instead of finding a solution from scratch; and of precomputed partial solutions, i.e. a set of static, offline-determined guidelines for coarse-grain, common failure scenarios. Our algorithms are able to produce a solution one to two orders of magnitude faster than known optimization

algorithms, such as simulated annealing and genetic. Furthermore, the solutions they produce keep system efficiency close the one achieved by an exhaustive, optimal solver. Lastly, we develop a methodology for characterizing degradable hardware components in terms of performance and energy efficiency, when transitioning from a fault-free status to a degraded configuration because of a fault. As such, we demonstrate that annotating said components as mentioned above is feasible.

1.4.2 Optimizing Heterogeneous Multiprocessors for Energy Efficiency

In Chapter 3 we study the runtime management of heterogeneous, DVFS-capable multiprocessor systems. Focusing on a dynamic scenario of multiple concurrently executing applications, we aim to select appropriate application-to-core placements and cluster frequencies, so as to maximize energy efficiency while satisfying individual applications' performance requirements. Our main contributions for this part of the thesis are the following:

We develop a model capable of predicting all running applications' performance and power, for every possible decision to be taken whenever an *application spawn* or *application termination* event takes place. The model uses compact, static, application-specific offline profiling information and projects them to the dynamic, application mix-dependent current status of the system, measured by online performance monitors. Additionally, a separate linear model uses these performance predictions together with static, system-specific power parameters, to predict system power. On top of the predictions, we develop three different runtime heuristics, that use the above model to evaluate a small subset of candidate solutions and select the one that maximizes energy efficiency while satisfying performance requirements. We achieve a fast response time that is also scalable with respect to the growing system size, i.e. the total number of cores in the system. This is made possible by aggressively limiting the search space at runtime, evaluating only a fraction of the available choices.

1.4.3 Optimizing Multiprocessor Performance with Load Balancing

In Chapter 4, we study load balancing by means of work stealing, to optimize the performance of parallel workloads on multicores. We aim to minimize the overheads of finding a suitable core to steal work from (i.e. a busy core) and of migrating the work to an idle core. The main contributions of this work are the following:

We introduce a lightweight method to construct an *approximate system load view*. This is a compact, cheap to maintain structure, kept by every core of the system. It contains information about which cores and/or system regions are most likely to be currently busy. The *system view* is constructed and maintained in a manner integrated with work stealing attempts. Subsequently, we propose a work stealing algorithm that utilizes the aforementioned *system view* to select a target core for stealing work from, whenever a core becomes idle. The selection is guided by the perceived current workload distribution, but is also hierarchical, prioritizing steals that do not require migration of the task's working set from a remote cache. We have evaluated our approach for a variety of workloads and found that, for workloads with different kinds of imbalance, it outperforms other alternatives, such as random or hierarchical work stealing, while also not penalizing a balanced workload.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

In Chapter 2, our work on graceful degradation for fault-tolerant multiprocessors is presented. We begin by providing a formal, high level description of a gracefully degradable system, based on mutually exclusive options for system components and workload tasks, each option annotated with performance, energy and functionality attributes. Subsequently, we use this description to sketch the system configuration space and as a result determine that an optimal, exhaustive search approach is not feasible at runtime, especially considering the ever-increasing number of components in a system. Thus, we continue by presenting our runtime approach, a set of custom heuristics capable of finding a configuration close to the optimal one, in time that is one to two orders of magnitude lower than other optimization algorithms, such as simulated annealing and genetic. We also develop versions of the latter two algorithms, both as alternatives and as comparison points for our custom heuristics. We evaluate all algorithms by comparing them to an optimal solver, both through simulations and through actual runs on a real machine. Lastly, before concluding the chapter, we present a methodology of characterizing degradable hardware components in terms of performance and energy efficiency. This methodology produces the necessary annotations to the high level system description, showcasing the applicability of our overall strategy. The work on runtime management for graceful degradation has been presented in [49] and [50]. Respectively, the principles of our component characterization methodology have been presented in [51] and augmented by advances on reconfigurable processor arrays presented in [52–54]. The methodology itself has been used for the evaluation of the approach proposed in [55].

In Chapter 3, a runtime manager for heterogeneous, DVFS-capable multiprocessors is presented. We start the presentation proposing a scalable profiling strategy, with which performance characteristics of individual applications can be determined offline, with a sequence of standalone runs. The number of runs in the sequence is linear with respect to the number of cores in the system. Next, we describe our technique of *online projection*, used to adapt the compact, static, application-specific offline profiling information to the dynamic, application mix-dependent current status of the system, measured by online performance monitors. This adaptation enables performance prediction for independent, concurrently executing applications. Subsequently, we present our power model, that uses these performance predictions, together with platform-specific, offline-estimated power parameters, to predict system power. We go on to propose three different low-complexity heuristics that partially search the solution space whenever an application spawns or terminates, using the prediction model to select the application placements and cluster frequencies that maximize energy efficiency while satisfying performance requirements of individual applications. Lastly, we evaluate our proposed approach, both through exhaustive experiments for specific events and through a plethora of dynamic scenarios. The proposed approach has been presented in [56].

In Chapter 4, a strategy for load balancing, using work stealing based on approximate system-load information is presented. In the first part of the chapter, we describe the *approximate system load view*, a lightweight structure maintained by all cores of the system and containing information pointing to the busiest system cores or regions. Subsequently, we present the integrated process of enriching the *system view* while using it to guide work stealing attempts. The proposed work stealing approach is then compared to existing techniques such as random and hierarchical work stealing, using a wide variety of workloads, including a perfectly-balanced workload that should not be penalized by work stealing, a workload described by a dynamically generated Directed Acyclic Graph (DAG) making it inherently imbalanced and a stencil workload which creates load imbalance due to its multiple task dependencies. The study that makes up Chapter 4 has also been presented in [57].

Lastly, Chapter 5 concludes the thesis mentioning the main takeaway points. It provides a summary of the previous chapters, a more detailed and (where applicable) quantitative description of thesis contributions and a discussion of interesting potential future research directions.

2 Runtime Support for Graceful Degradation

The shrinking of transistor dimensions comes with increased performance potential, but also with challenges for utilizing it efficiently. One of the greatest such challenges is reliability. Conventional testing of chips becomes more costly and complex as transistor counts increase, while component variability makes it harder to assume consistent behavior of components and forces conservative clocking. In addition to the above phenomena, there is also an increase of wearout (aging) effects [1]. This is especially important, as conventionally chips are tested just after fabrication and the possibility of aging effects renders the results of this type of testing insufficient for the chip lifetime as a whole. Such aging effects can result to system failure at an unpredictable time. As a result, the prospect of reliable systems built on unreliable components is now a design alternative, if not a necessity.

On the other side of reliability challenges, we find performance potential. Part of this potential can be sacrificed for mitigating reliability hazards, through error detection and correction, online testing for defects and aging effects and, as demonstrated in this chapter, life-long system-level management for dealing with faults. This management can take advantage of adaptive hardware and software properties of modern multiprocessor systems, such as: (i) reconfigurable components [9, 55, 58, 59] for isolating permanent faults to ensure graceful performance and energy degradation and (ii) flexible workloads facilitating graceful degradation of functionality when system components fail; examples of flexible workloads are found in mixed criticality systems [60] as well as in approximate computing [61].

Graceful Degradation (GD) is, the transition to a lower state of some *system aspect(s)* as a response to the occurrence of an *event* that prohibits the manifestation of full fledged system behavior [62]. In the context of adaptive multiprocessors, the aforementioned event is a permanent fault caused by aging, limiting the set of working hardware components on the chip. As explained in Chapter 1, the unpredictability of the time, type and location of these faults, creates the need to make GD-related decisions at runtime, after each such event. When it comes to the system aspects that can be degraded, we identify three of them, which we touched upon in the previous paragraph: reconfigurable hardware allows fault isolation and replacement of faulty parts, resulting to performance and energy efficiency degradation, while workload flexibility allows for functionality degradation. The impact of subsequent faults on various system aspects, gradually leading to system failure, is illustrated in Figure 2.1. Simply put, runtime system management for GD aims to minimize the negative impact of unpredictable permanent faults, both to prolong system lifetime and maximize system efficiency during this lifetime.

The purpose of the work presented in this chapter is to provide means for fast and effective runtime system management for GD. In particular, the runtime manager has to decide on the *configuration of hardware components*, the version of a *flexible workload* and *the mapping* of this workload to the hardware components. We call the set of these three elements a *system configuration*. The decision for a new system configuration has to be as quick as possible, keep all degradable aspects within acceptable limits and maximize an objective function that expresses overall system efficiency. Alongside the above objectives of runtime management, this chapter also introduces a methodology for characterizing hardware components in terms of performance and energy efficiency, providing required inputs to the runtime manager. To summarize, this chapter makes the following contributions:

- A formal description of the problem of GD of adaptive multiprocessors, leading to an objective function expressing overall system efficiency under successive permanent faults.
- Multiple runtime algorithms, triggered by a *component failure* event and aiming to quickly find a new *system configuration* that maximizes the aforementioned objective function and minimizes the one-time overheads of applying the new configuration.
- Comparative evaluation of the above algorithms in terms of solution quality and execution time.
- A methodology to characterize degradable hardware components in terms of performance and energy efficiency, producing the necessary information to the runtime GD manager.

The rest of this chapter is organized as follows: Section 2.1 summarizes related work, showcasing means to expose degradable aspects to the runtime manager; in Section 2.2 we formulate GD as an optimization problem; Section 2.3 presents our algorithms for

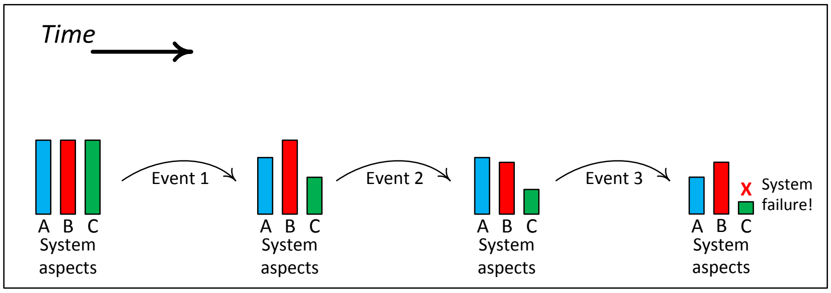


Figure 2.1: Runtime system aspect degradation. Each event (permanent fault) causes one or more system aspects to degrade. Any system aspect being reduced to an unacceptable value results in system failure.

tackling the problem; in Section 2.4 the experimental setup is described and quantitative evaluation of the different algorithms takes place; Section 2.5 contains our degradable component characterization methodology, leading to a design space exploration with regards to reconfiguration granularity; finally, Section 2.6 summarizes and concludes the chapter.

2.1 Related Work

This section reviews related work that motivates the study presented in this chapter. Section 2.1.1 showcases how modern systems have flexibility that can be used to mitigate the effect of permanent faults. We identify certain degradable aspects that are exposed by this flexibility, thus making runtime management for GD an attractive choice. Subsequently, Section 2.1.2 summarizes some other runtime strategies that, just like our approach, make use of this potential.

2.1.1 Flexibility in Modern Multiptocessor Systems

There exist multiple strategies and paradigms that allow manipulation of either the hardware or the workload of a system, to allow this system to keep working despite multiple permanent faults, by degrading one or more aspects.

Manipulating hardware is done via reconfiguration. A relevant, well researched topic is this of reconfigurable pipelines [9–11, 55]. In this paradigm, a processor array consists of processors with interchangeable and interconnected pipeline stages. Upon failure of some of these pipeline stages, the fault-free stages of different processors are connected to form new, working ones, as illustrated in Figure 2.2. These processors are less efficient than the original ones, in terms of both performance and energy, as they have to use interconnection logic and their parts are further apart. Consequently, we can claim that hardware reconfiguration exposes the degradable aspects of *performance* and *energy*

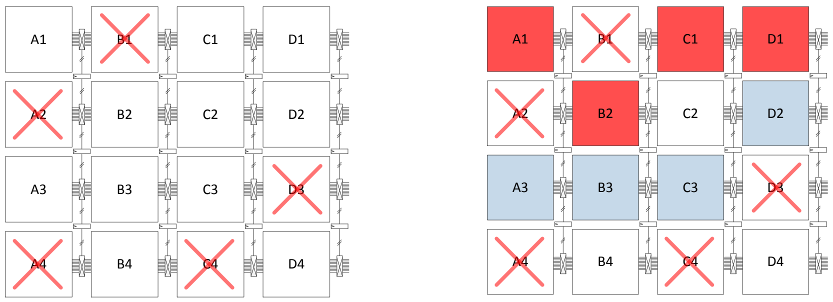


Figure 2.2: The concept of reconfigurable pipelines. An array of four microprocessors (A, B, C and D), divided into pipeline stages with interconnections between them. If some of the stages are faulty (denoted by a red X), the rest can be connected to form working processors (red and blue sets on the right).

efficiency, for tolerating permanent faults. In order to make educated decisions on such arrays, it is important that the performance and energy efficiency degradation of a faulty array, compared to a fault-free one, can be predicted, something that we demonstrate in Section 2.5.

The aforementioned interchangeable processor parts are generically called *substitutable units* and can be of various granularities: whole cores [4–8], pipeline stages [9–11], functional units (e.g. ALUs) [17, 18, 63, 64], even logic gates [12]. Fine-grain reconfigurable fabric (similar to FPGA logic) has also been suggested for implementing slower and more power-hungry versions of faulty functional units [65]. The choice of granularity exposes an interesting tradeoff, as finer granularities provide more flexibility, but at the same time require more complex decisions and have higher implementation overheads.

System workload can also be manipulated in ways that mirror hardware reconfiguration. There also exists a granularity spectrum for workload-driven GD. The coarser-grain methods involve dropping a subset of the workload tasks [7, 8, 66], in particular taking advantage of the concept of *mixed criticality*, i.e., a workload consisting of both essential and optional computations [60]. In the middle of the granularity spectrum, we find methods based on multiple diverse implementations of the same task. Alternative implementations might use a different set of hardware resources to allow adaptation when components fail [13], or might produce results of reduced precision as per the paradigm of *approximate computing* [61]. Workloads can also be manipulated at the fine-grain instruction level, using the method of *alternative microcode* [14–16]. In this case, instructions that would normally be executed on faulty components are substituted by less efficient instruction sequences, that can be executed on components that are still working. For instance, a multiplication can be performed as a sequence of shifts and additions, if the multiplier is faulty.

Much like hardware reconfiguration, alternative microcode exposes the degradable aspects of performance and energy efficiency: the alternative instruction sequence performs the same computation, but in more time and using more energy. However, other kinds of workload manipulation (diverse implementations and task dropping) expose a new degradable aspect, which we subsequently call *functionality*. The term *functionality degradation* refers to a faulty system delivering, in some way, fewer results than its fault-free counterpart. This is implemented either by dropping less important tasks, or choosing a less precise (approximate) version.

In summary, the work presented in this chapter considers three degradable aspects: performance, energy efficiency and functionality, exposed both via hardware reconfiguration and flexible workloads.

2.1.2 Other Runtime Approaches for Graceful Degradation

Several published studies address problems related to the one tackled in this chapter. This section mentions the most relevant ones and clarifies our contributions in comparison.

Transistor aging has motivated the study of lifetime extension of aging-prone multiprocessors. Component aging is strongly correlated to the utilization of said component. Thus, techniques for load distribution among system components have been proposed, with the goal of extending system lifetime, e.g. [20–22]. Researchers compare the effect of balancing the workload against stressing some components more than others, possibly according to their varying susceptibility to wearout. As important as this research is, it tends to overlook modern multiprocessors' adaptability, in the form of hardware reconfigurability [9, 58, 59] and workload flexibility [60, 61], both of which can be instrumental to extracting more efficiency, as they expose degradable aspects (see Section 2.1.1). Furthermore, our work focuses more on decision making for minimizing the impact of permanent faults, rather than avoiding the faults. Another work focusing on the same objective utilizes both fine-grain hardware reconfiguration and software emulation to compensate for the failure of dedicated hardware components [19, 67]. Despite considering GD from a different perspective, all works mentioned in this paragraph recognize the importance of studying the cumulative effect of multiple successive faults taking place throughout the system's lifetime - a central aspect of our study as well.

Other researchers study task (re)mapping strategies as a reaction to permanent faults, with the objective of maintaining acceptable system functionality [68, 69]. This research direction is also significant, and highlights the importance of minimizing the impact of faults. However, our view is that in adaptive systems, task (re)mapping is just one of the knobs that can be used for GD and, as such, it is sub-optimal to view it as a standalone problem. More specifically, a component may not be restricted to either being fully functional or failed: it can be functioning in a degraded mode using hardware reconfigurability, being able to carry out a subset of the tasks it originally could - complicating the problem of (re)mapping. Additionally, under multiple successive faults, system management should be free to choose to drop non-essential tasks or transition to lighter/approximate task versions (functionality degradation) rather than just remapping them on other components. Furthermore, although the importance of considering successive component failures is stated, the aforementioned works do not evaluate their proposed methods for fault sequences longer than two faults.

Another important challenge of shrinking transistor dimensions, is balancing lifetime, performance and resilience to transient faults. Having system cores working in Triple or Double Modular Redundancy (TMR or DMR) helps against transient faults, but keeps multiple components busy executing the same workload (penalizing performance) and burdens them (penalizing lifetime). These tradeoffs are studied in [23], stressing the concept of fail-operational systems - that is, keeping the system working under successive permanent faults as we do. We do not consider transient faults in this study, however our system model can support alternative task versions that are reliable but entail a performance penalty, (e.g. using DMR or TMR). In that case, the runtime management would have to also be evaluated for the degradable aspect of *resilience to transient faults* alongside its performance, functionality and energy efficiency.

Reliability and fault tolerance of adaptable multiprocessors have also been studied in combination with other optimization goals, mainly power efficiency. As an example, Bolchini et al. [25] have proposed a runtime manager that is able to choose the most energy-efficient placement for an application on a heterogeneous multicore, at the same time prioritizing components that have aged the least, in order to also extend the system lifetime considering transistor aging.

In summary, the work presented in this chapter provides means of efficient runtime management for GD. The manager has to react to a sequence of multiple subsequent faults, every time providing a new system configuration with minimal overheads. The new system configuration should utilize both hardware and workload flexibility and maximize system efficiency considering the effect of three separate degradable aspects - performance, energy efficiency and functionality. Lastly, in the latter part of the chapter, we demonstrate that our assumption of knowing the performance and energy efficiency of degraded versions of the system hardware is reasonable, by presenting a methodology for such characterization.

2.2 Graceful Degradation Problem Formulation

This Section contains our formulation of GD as an optimization problem. In Subsection 2.2.1 we describe a high-level model of a heterogeneous multiprocessor that consists of reconfigurable hardware components and runs a flexible workload. Subsection 2.2.2 outlines the solution space constructed from the system description and defines the objective function that the runtime GD manager has to optimize. Lastly, Subsection 2.2.3 contains expressions for the one-time reconfiguration overheads that the GD manager needs to minimize.

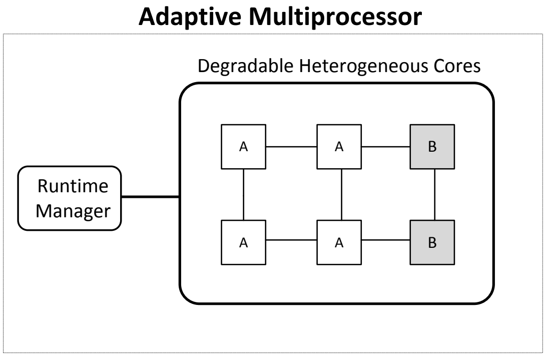


Figure 2.3: An adaptive heterogeneous multiprocessor, which incorporates system management for graceful degradation.

2.2.1 High Level System Description

This study considers an adaptive heterogeneous multiprocessor system such as the one of Figure 2.3. It consists of a reliable administrator core (runtime manager) which is in charge of the runtime system and manages a set of heterogeneous worker cores (e.g, big, high-performance cores and little, power efficient cores), on which the workload runs. The worker cores are considered unreliable, but degradable: permanent faults do not render them completely unusable, but can be worked around by means of hardware reconfiguration. The workload running on the worker cores is a collection of essential tasks and optional features, both of which can have multiple versions, e.g. supporting different levels of precision. The above characteristics allow for an overall adaptive system. A good example of this paradigm is the DeSyRe SoC [55, 70].

At this point we state some assumptions which help us focus on the problem at hand:

- A fault detection and, if needed, data recovery mechanism is present.
- No faults happen on the reliable administrator core.
- Reliability of the interconnect is an orthogonal consideration (see, e.g., [71]).

Our high-level description of the system's adaptive properties is illustrated in Figure 2.4. Each hardware component is called C_i^j, where i is the component index and j is the component type (e.g. big or little core). Each of these components is represented by one row at the left side and supports multiple configurations, each denoted by a capitalized letter next to the component type, j. One configuration corresponds to the component's fault-free version (the leftmost circle of each row labelled C_i^{jA}), one to its failed (useless) version (the rightmost circle labelled C_i^{jF}) and one or more for degraded modes, meaning that at least one permanent fault exists on the component but reconfiguration has successfully isolated it. Note that the status of each component (fault-free, degraded or failed) renders some configurations meaningless or impossible:

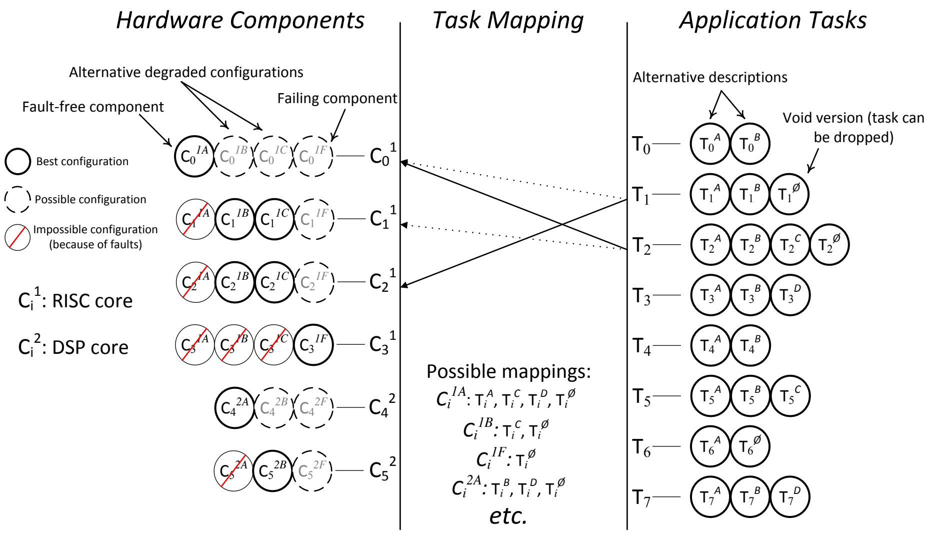

Hardware Components
Task Mapping
Application Tasks
Alternative degraded configurations
Fault-free component
Failing component
Best configuration
Possible configuration
Impossible configuration (because of faults)
C_i^1: RISC core
C_i^2: DSP core
Alternative descriptions
Void version (task can be dropped)
Possible mappings:
C_i^{1A}: $T_i^A, T_i^C, T_i^D, T_i^\emptyset$
C_i^{1B}: $T_i^C, T_i^\emptyset$
C_i^{1F}: $T_i^\emptyset$
C_i^{2A}: $T_i^B, T_i^D, T_i^\emptyset$
etc.

Figure 2.4: The high-level system model for a gracefully degrading multiprocessor, on which the problem formulation is based.

a fault-free component will generally work in its original, fault-free configuration; a degraded component has to work in one of its degraded modes; for a failed component, only the rightmost configuration can be chosen, rendering it unusable.

Similarly, each workload task is called T_i, i being the task index. Each of these tasks is represented by a row at the right side of the figure and also has multiple versions: One normal/full version (the leftmost circle of each row, labelled T_i^A), one or more alternative versions (e.g., lightweight or approximate) and possibly, one void version (labelled $T_i^\emptyset$)) corresponding to dropping a non-essential task.

The center part of Figure 2.4 illustrates the possible mappings of tasks to hardware components, essentially listing the set of task versions that each component is capable of executing. These depend on the current component configurations and chosen task versions.

Note that the presented system model is generic with respect to the origin of components' and tasks' different versions. As such, it can describe different kinds of adaptive systems. For instance, to describe a multiprocessor capable of independent DVFS per cluster [72], we can define one component version (on the left side of Figure 2.4) for each voltage/frequency configuration of each core. If the aforementioned system runs an application with different approximation levels or tasks of mixed criticality, we can define task versions appropriately (at the right side of Figure 2.4). We propose a generic manner to express such adaptability and to guide the relevant runtime decisions.

Two different application models are considered in this study. The first is a collection of independent tasks to be executed in any order. The second is a Directed Acyclic Graph

(DAG) of tasks, expressing task dependencies. Our methods are applied on one use case of each application model, described in Section 2.4.1.3.

Based on this system description, the job of a runtime manager is to choose one of the possible options for each of the aforementioned system elements:

- one configuration for each component on the left,
- one version for each task on the right and
- a mapping of each task on exactly one component

The aggregate of these choices is henceforth collectively called *system configuration*, not to be confused with the configuration of hardware components. Similarly, the process of transitioning from one system configuration to another, is called *system reconfiguration*. We also assume that this reconfiguration phase is reliable.

2.2.2 Solution Space and Objective Function

Given the high level system description of the previous subsection, the next step is to sketch and explore the solution space, define a way to evaluate the quality of different solutions (objective function) and assess the complexity of optimally solving the problem.

To describe the solution space, let us initially focus on the side of hardware configuration. Still referring to Figure 2.4, a hardware configuration for the system consists of a choice of exactly one option for each of the components. To enumerate all possible system hardware configurations, we choose the notation HW_n, where $n \in \{0, 1, ..., N-1\}$ and N is the total number of possible hardware configurations.

To calculate the value of N, we have to focus on each type of component (j) separately. Consider that $mult\_C^j$ is the multitude of components of type j (i.e. how many of them exist in the system) and $conf\_C^j$ is the number of possible configurations of each. Then the total number of possible configurations for this subset of components $total\_conf(j)$, is the number of combinations of choosing $mult\_C^j$ times among $conf\_C^j$ choices, with repetition allowed and without considering order. This is given by the following formula:

$$total\_conf(j) = \binom{mult\_C^j + conf\_C^j - 1}{mult\_C^j} = \frac{(mult\_C^j + conf\_C^j - 1)!}{mult\_C^j! * (conf\_C^j - 1)!}$$

Given the number of possible configurations $total\_conf(j)$ for each group of components with the same type j, the number of configurations for the whole system is their product:

$$N = \prod^{j} total\_conf(j)$$

As an example, for the system of Figure 2.4 which consists of four components of type 1 with four possible configurations and two components of type 2 with three possible configurations, we have:

$$total\_conf(1) = \frac{(4+4-1)!}{4! * (4-1)!} = \frac{7!}{4! * 3!} = 35,$$

$$total\_conf(2) = \frac{(2+3-1)!}{2! * (3-1)!} = \frac{4!}{2! * 2!} = 6 \ \ and$$

$$N = total\_conf(1) * total\_conf(2) = 35 * 6 = 210$$

In particular, hardware configuration HW_0 is $\{C_0^{1A}, C_1^{1A}, C_2^{1A}, C_3^{1A}, C_4^{2A}, C_5^{2A}\}$, while HW_{209} is $\{C_0^{1F}, C_1^{1F}, C_2^{1F}, C_3^{1F}, C_4^{2F}, C_5^{2F}\}$. Note that, in the presence of faults, some of these configurations are not eligible for selection.

Focusing next on the right side of Figure 2.4, we can enumerate the possible workloads that can be selected for execution on the multiprocessor. Each system workload consists of exactly one task version for each task T_i. This results in

$$M = \prod^{i} number\_of\_versions(T_i)$$

different workloads, indexed WL_m where $m \in \{0, M-1\}$. A particular workload is a listing of the selected option for each of the tasks, e.g. $WL_0 = \{T_0^A, T_1^A, T_2^A, T_3^A, T_4^A, T_5^A, T_6^A, T_7^A\}$.

Lastly, let us enumerate and index the options for tasks-to-components mapping. The amount of possible mappings cannot be determined statically, since it depends on the currently selected hardware, HW_n and workload WL_m. Based on these, $mapping\_options(T_i)$ can be determined for each task T_i and the possible mappings calculated as

$$L(n,m) = \prod^{i} mapping\_options(T_i)$$

L is upper bounded by κ^τ, with κ being the number of components and τ the number of tasks. A particular mapping is called MAP_l, $l \in \{0, 1, ..., L-1\}$ and is a list of the components which execute each task, in task order, as such: $\{0, 0, 4, 3, 5, 1, 1, 2\}$.

A solution ($S(n,m,l)$) to the GD problem is essentially a system configuration and as such it consists of all three elements, HW_n, WL_m and MAP_l. As an example, for the system snapshot of Figure 2.4, one possible configuration is shown in Figure 2.5

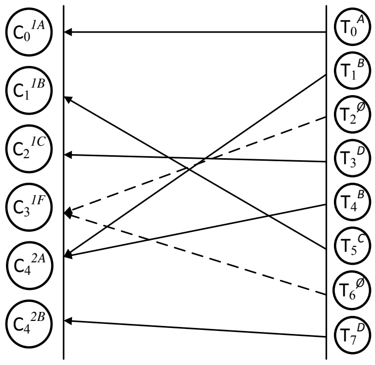


Figure 2.5: A valid system configuration for the snapshot illustrated in Figure 2.4.

and consists of the following elements: $\{C_0^{1A}, C_1^{1B}, C_2^{1C}, C_3^{1F}, C_4^{2A}, C_5^{2B}\}$ for the hardware configuration, $\{T_0^A, T_1^B, T_2^\emptyset, T_3^D, T_4^B, T_5^C, T_6^\emptyset, T_7^D\}$ for the workload and $\{0, 4, 3, 2, 4, 1, 3, 5\}$ for the task mapping.

Figure 2.6 depicts a representation of the solution space using a four-level tree, the root level of which is an empty solution and the leaf nodes are complete solutions (system configurations). Each of levels 2, 3 and 4 of the tree add one element to the solution: the workload, hardware configuration and mapping respectively. Note that levels 2 and 3 can be interchanged, since these two elements do not pose any restriction to each other. Mapping, on the other hand, has to be placed at the deeper level, since it is limited by both other elements.

To assess the quality of each solution we consider the three degradable aspects identified in Section 2.1.1:

- System Performance (P): how fast the particular configuration can execute one iteration of the chosen workload.
- System Energy (E): how little energy is required to carry out one iteration of the chosen workload.
- System Functionality (F): how much of the full functionality of the application (including non-essential extra features) is being served.

To be able to quantify the above metrics, each task version has to come annotated with a *task functionality* (f) value. Task functionality f corresponds to the fraction of the overall functionality this task version represents: normal versions score better than lightweight (e.g. approximate) ones, while void versions score zero. Furthermore, each combination of a component version with a task version is annotated with a *task execution time* (t) value and a *task energy* (e) value. *Task execution time* (t) shows how long it takes the component to complete the task, while *task energy* (e) shows how much energy is required to do so. In Section 2.5 we propose one methodology to determine t and

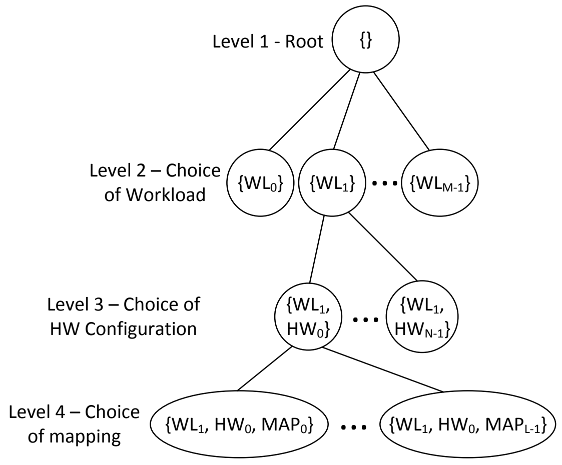

Level 1 - Root
{}
Level 2 – Choice of Workload
{WL0}
{WL1}
{WLM-1}
Level 3 – Choice of HW Configuration
{WL1, HW0}
{WL1, HWN-1}
Level 4 – Choice of mapping
{WL1, HW0, MAP0}
{WL1, HW0, MAPL-1}

Figure 2.6: The solution space, modeled as a tree. Leaf nodes are solutions of the problem (possible system configurations).

e values for tasks on degraded configurations of components. In general, profiling for execution time and energy is also being separately researched, e.g. [73, 74].

Based on the above t, e and f annotations for all tasks, the three aspects, P, E and F for the system can be quantified. The values for energy and functionality for the whole system are obtained by summing up the values of the chosen task versions, while the performance value is determined by the component that takes the longest to complete its assigned workload. By considering the best and worst possible values for each task's t, e and f individual contributions, we can also define a range for each of the aspects, P, E and F. As the three aspects are measured in different units, each of the three ranges is subsequently normalized in the interval [0, 1], 1 representing the best possible attainable for each aspect (highest performance, highest functionality, lowest energy). The normalized values have no units and are thus comparable to each other. Eventually, an objective function can be defined, which we name *System Efficiency* (SE): a weight is assigned to each of the three aspects, the weights summing up to 1, and *System Efficiency* is the sum of the three value-weight products. More precisely:

$$SE = w_1 * P + w_2 * F + w_3 * E \qquad (1)$$

$$\{w_1, w_2, w_3\} \in [0, 1]$$

$$P, F, E \in [0, 1] \qquad (2)$$

$$w_1 + w_2 + w_3 = 1 \qquad (3)$$

The weights are determined by the priorities of each individual system, e.g. E is more important when saving battery capacity is critical, while P is more important for high-performance systems. (2) and (3) combined mean that the value of *SE* lies in [0, 1]. *SE* is the objective function we need in order to define GD of adaptive multiprocessors in the presence of permanent faults, as an optimization problem. Given that the number of possible solutions is finite, one of them maximizes the objective function and is, as such, optimal in the specified context.

In addition to maximizing the objective function, a system configuration also has to respect some minimum quality of service constraints. Referring to Figure 2.1, each of the degradable aspects should remain above a certain value - inability to achieve this results in a system failure. Also, all workload tasks that constitute essential functionality should obviously be executed.

Before proceeding to the next subsection, it is useful at this point to quantify the size of the solution space, in order to showcase the infeasibility of finding the optimal solution at runtime. The amount of possible solutions (or leaf nodes in the tree of Figure 2.6), S, is upper bounded as follows:

$$S \leq N * M * L_{max} = N * M * (\kappa^{\tau})$$

One can argue that the actual number of solutions is significantly lower than the above upper limit, based on the following arguments:

- The number of possible hardware configurations for a given system status is lower than N, especially when there are multiple faults on the chip - making certain options invalid.
- The amount of possible mappings also depends on the choice of hardware and workload and can be significantly lower than κ^{τ}.
- For the mapping element of the solution, there are options which are equivalent by symmetry - if there exist identical components currently working at the same configuration option.

However, the complexity of the problem is still exponential with respect to the number of tasks, τ. Indeed, if μ is the minimum number of versions for one task over all tasks, it holds that

$$M \geq \mu^{\tau}, \quad \mu = \min_{i=0,\dots,\tau-1} number\_of\_versions(T_i).$$

Thus, in the product $N * M * L$, at least one factor increases exponentially with respect to the size of the problem input, making exhaustive search impractical at runtime.

In this subsection we obtained an objective function for the problem of runtime GD management. This objective function expresses the overall efficiency of the system in its new configuration. However, moving from one system configuration to another also incurs some one-time overheads, which are analyzed in the next subsection.

2.2.3 System Reconfiguration Cost

When a *component failure* event triggers the runtime manager, the latter has to come up with a new system configuration, adapting the system to the reduced set of hardware resources, aiming to maximize *System Efficiency*. However, another factor that has to be taken into account is the one-time cost of applying this new system configuration.

This cost comes in the form of time and energy required to modify the system elements explained in Section 2.2.1. Each hardware component reconfiguration, task migration and change of task version (possibly requiring the load of a different binary) has an associated time (and energy) cost. These costs depend on characteristics of the system, such as the interconnect and hardware reconfiguration granularity. Without loss of generality, in this study we model them as follows:

To reconfigure a number of hardware components, a flat time cost c_i needs to be paid to initiate reconfiguration, plus some amount c_j for each of the α affected components:

$$C_{HW} = c_i + \alpha * c_j$$

To migrate a task from one hardware component to another, the cost C_{mig} is proportional to the distance between the two components.

$$C_{mig} = c_k * dist(comp_a, comp_b)$$

The cost to migrate β tasks depends on the interconnect mechanism of the system. We assume a worst-case value, which is the sum of all individual migration costs:

$$C_{remap} = \sum^{\beta} C_{mig}$$

Lastly, to change the version of a task, a flat time cost c_l has to be paid. To change versions of γ tasks, again we have to assume the worst case of them being executed on the same hardware component, resulting to the total cost being the sum of the individual costs:

$$C_{SW} = \gamma * c_l$$

The total worst case time cost (assuming parallelization of different steps is not possible) for system reconfiguration is thus:

$$C = C_{HW} + C_{remap} + C_{SW}$$

Similar models can be constructed for the energy cost. The costs c_i, c_j, c_k, c_l are system-specific and entered to our experiments as constant parameters.

We consider the reconfiguration cost as a constraint - that is, the runtime manager should find a new system configuration which can be applied in at most C_t time. The algorithms presented in Section 2.4 are evaluated with and without such a constraint.

2.3 Proposed Solutions

In this Section, the various algorithms implementing a runtime manager for GD are presented and compared in terms of complexity. We have developed custom heuristics for both application models described in Section 2.2.1 (standalone tasks and DAG), as well as a simulated annealing and a genetic algorithm, tailored to the characteristics of the particular problem.

All algorithms receive the following inputs:

- The list of n hardware components and their possible configurations.
- The list (or DAG) of m application tasks and their different versions.
- The possible mappings of tasks to components, and corresponding execution times and energy costs.
- The current system status, that is the list of working hardware components and their degradation levels.

All algorithms produce a new system configuration, consisting of three arrays of integers:

- An n-sized array called HW, indicating the chosen configuration index of each hardware component.
- An m-sized array called SW indicating the chosen version index for each task.
- An m-sized array called MAP indicating the hardware component index that each task is mapped on.

All approaches are, to some extend, based on two key concepts:

- **Incremental solutions:** Instead of searching for a system configuration from scratch, the current system state is taken into account, attempting to find what to change in the current configuration. This limits the search to a fraction of the solution space.
- **Precomputation:** As the kinds of events that can occur are known in advance, partial precomputed solutions are created offline for common scenarios, such as the complete failure of any number of components.

As in the following presentation we calculate the asymptotic complexity of our algorithms in terms of the problem size, we repeat that the number of hardware components in the system is κ and the number of tasks in the workload is τ.

2.3.1 Heuristics for Independent Tasks

We developed three different heuristics for the simple application model. Two of them (*INCR-1* and *INCR-2*) attempt to form an incremental solution based on the current system configuration, while the third (*PREC*) uses a list of precomputed workloads and attempts to construct a system configuration using one of them.

The first incremental algorithm is shown in the pseudocode of Algorithm 1. It checks for still working hardware configurations of the component on which the triggering event (permanent fault) happened, C_e (line 1). For each of these, it cycles through the tasks that C_e was executing before the event (line 4) and attempts to find a version that can still be executed by the degraded component (lines 5-13). Note that at this step, for non-essential tasks, there is the option of selecting their void version, $T_j^{\emptyset}$. If all these tasks can still be accommodated on C_e (that is, C_e can still perform its duties to an acceptable degree), a new system configuration is returned based on the above modifications (line 16).

To calculate the time complexity of *INCR-1*, we observe the boundaries of the three *for* loops. The outer loop has a constant number of iterations - the problem size does not matter for how many alternative configurations hardware components have. The same is true about the inner loop - each task of the workload has a constant number of alternative options. The middle loop iterates a maximum of τ times, one for each task currently running on C_e. Thus, *INCR-1* has time complexity of $\mathcal{O}(\tau)$.

Algorithm 1: *INCR-1*

```
1: for each valid configuration $C_e^i$ of $C_e$ do
2:     $HW[e] = [i]$
3:     $success = 1$
4:     for each task $T_j$ mapped on $C_e$ do
5:         for each version $T_j^k$ of $T_j$ do
6:             if $T_j^k$ can be executed on $C_e^i$ then
7:                 $SW[j] = k$
8:                 break
9:             end if
10:        end for
11:        if no version $T_j^k$ succeeded then
12:            $success = 0$
13:        end if
14:    end for
15:    if $success = 1$ then
16:        Return resulting system configuration
17:    end if
18: end for
19: Return algorithm failure
```

The difference of *INCR-2* to *INCR-1* is that it can map the tasks previously performed by C_e on any component of the system. In other words, if C_e is not able to still perform its duties to an acceptable level, *INCR-1* will fail, whereas *INCR-2* has a good chance of success. *INCR-2* is described by the pseudocode of Algorithm 2. The ability to remap tasks on components different than C_e is described between lines 6 and 12. It can be observed that the added flexibility comes at the cost of a fourth *for* loop which iterates a maximum of κ times, resulting in time complexity of $\mathcal{O}(\kappa * \tau)$.

Algorithm 2: *INCR-2*

```
1: for each valid configuration $C_e^i$ of $C_e$ do
2:     $HW[e] = [i]$
3:     $success = 1$
4:     for each task $T_j$ mapped on $C_e$ do
5:         for each version $T_j^k$ of $T_j$ do
6:             for each HW component $C_l$ do
7:                 if $T_j^k$ can be executed on $C_l$ then
8:                     $SW[j] = k$
9:                     $MAP[j] = l$
10:                    brake
11:                end if
12:            end for
13:            if version $T_j^k$ was mapped then
14:                break
15:            end if
16:        end for
17:    end for
18:    if some task $T_j$ was not mapped then
19:        $success = 0$
20:    end if
21:    if $success = 1$ then
22:        Return resulting system configuration
23:    end if
24: end for
25: Return algorithm failure
```

Note that both algorithms operate on only a subset of the τ system tasks, so the τ term of the $\mathcal{O}$-notation is quite a pessimistic upper bound. Thus, their execution time in practice is expected to scale somewhat better than that.

The third heuristic, *PREC*, makes use of the partial precomputed solutions. Each such solution is a predefined workload, that is, a set of values for array SW of the system configuration. For each system component that has failed completely (offering only configuration C_i^{jF} as an option), some non-essential tasks are dropped and/or some tasks are degraded to a more lightweight (e.g. approximate) version. The precomputed workloads have been defined statically offline, to be suitable for a system of a certain degree of degradation. Thus, the more components are still functional, the more functionality the corresponding precomputed workload preserves. Coming to the heuristic, which is described by the pseudocode of Algorithm 3, it fetches a precomputed workload (line 3) and attempts to map it on the working system components (lines 5-15). If that fails, it repeats this procedure for an even lighter workload and continues as such for a constant number of workloads, c (see loop boundary in line 2). c is a parameter which affects execution time, but not its asymptotic complexity since it is constant. The two inner *for* loops iterate for a maximum τ and κ times respectively, resulting to asymptotic complexity of $\mathcal{O}(\kappa * \tau)$.

Looking back at the three heuristics, it can be observed that *INCR-1* is very often unsuccessful in finding an acceptable system configuration. Indeed, in case the triggering event has disabled component C_e completely and one of the tasks running on it is an essential one, *INCR-1* cannot remap it on a different component. On the other hand,

Algorithm 3: *PREC*

```
1: $i$ = overall system degradation level
2: for $j = i$ to $(i + c)$ do
3:     fetch precomputed workload $j$
4:     $success = 1$
5:     for precomputed version $T_k^l$ task $T_k$ do
6:         $SW[k] = l$
7:         for each HW component $C_r$ do
8:             if $T_k^l$ can be executed on $C_r$ then
9:                 $SW[k] = l$
10:                $MAP[k] = r$
11:            else
12:                $success = 0$
13:            end if
14:        end for
15:    end for
16:    if $success = 1$ then
17:        Return resulting system configuration
18:    end if
19: end for
20: Return algorithm failure
```

PREC is the most conservative and pessimistic of the three heuristics, as it works with a version of the workload that is tailored to the degraded status of the hardware. Motivated by the above analysis and by indicative results of the three heuristics, we have decided to use *INCR-1* and *PREC* executing one after the other, as one strategy. Thus, if the fastest algorithm fails, it is backed up by the most conservative one. The complexity of this combined algorithm *INCR-1 + PREC* is still $\mathcal{O}(\kappa * \tau)$.

2.3.2 Heuristic for DAG

In case the application tasks have dependencies, the simple application model does not suffice. Practical heuristics for this case have to take into account task dependencies as they are described by a DAG. Furthermore, communication costs for transferring the output of a parent task to a child task have to be considered when evaluating the performance of any system configuration.

The heuristic we have developed for this application model (*DAG-GD*) is comprised of two parts:

- A greedy mapping algorithm (see pseudocode of Algorithm 4) which maps each task of a taskset on the first component it finds for which the critical path of the DAG is not extended. Between lines 8 and 25, the algorithm estimates the critical path of a partial mapping, ignoring interference between different paths, to keep the algorithm complexity low. Of course, such interference affects system performance and is taken into account during evaluation.

- An iterative wrapper (see pseudocode of Algorithm 5), which modifies the hardware components' configuration (line 3) and workload (lines 5-9) before calling the greedy mapping (line 10).

Algorithm 4: *GREEDY MAPPING*

```
1: Input: A set of tasks, $(\mathcal{T})$
2: tasks are ordered by index
3: tasks in the longest paths of the DAG have lower index
4: The set of system components, C
5: components are ordered by current load
6: less loaded components have lower index
7: for version $T_i^j$ of each task $T_i$ in $(\mathcal{T})$ do
8:     if $T_i$ belongs to critical path AND is not the origin then
9:         Let $k$ be the index of a parent task of $T_i$
10:        if $T_i^j$ can be executed on component with index $MAP[k]$ then
11:            $MAP[i] = k$
12:        else
13:            for each HW component $C_l$ do
14:                if $T_i^j$ can be executed on $C_l$ then
15:                    $MAP[i] = l$
16:                end if
17:            end for
18:        end if
19:    else
20:        for each HW component $C_l$ starting from least busy do
21:            if $T_i^j$ can be executed on $C_l$ then
22:                $MAP[i] = l$
23:            end if
24:        end for
25:    end if
26:    Update path lengths with possible communication costs
27:    Update load of component with index $MAP[i]$
28: end for
```

The principles of incremental solutions and precomputation are also applied in this case. The iterative wrapper attempts different configurations for the component C_e on which the triggering event took place and assembles the workload consulting the partial precomputed solutions for the degradation level of the system, similarly to the *PREC* heuristic. Furthermore, the greedy mapping operates only on the tasks of component C_e, much like algorithms *INCR-1* and *INCR-2*.

Concerning time complexity, the two loops of *DAG-GD* have been constructed with constant boundaries. As for the greedy mapping, it consists of two loops which will iterate a maximum of τ and κ times respectively, resulting to a complexity of $\mathcal{O}(\kappa * \tau)$. Upper bounds for both loops are quite conservative.

2.3.3 Adaptation of Existing Optimization Strategies

Alongside our custom heuristics, we decided to adapt existing optimization strategies to our problem. In the following subsections we will talk about a simulated annealing algorithm [75] and a genetic algorithm [76]. In both descriptions we will mostly focus on our contribution, that is, the algorithms' adaptation to the problem of runtime management for GD.

Algorithm 5: *DAG-GD*

```
1: $i$ = overall system degradation level
2: for each valid configuration $C_e^i$ of $C_e$ do
3:     $HW[e] = [i]$
4:     for $j = i$ to $(i + c)$ do
5:         fetch precomputed workload $j$
6:         for precomputed version $T_k^l$ task $T_k$ do
7:             Add $T_k^l$ to taskset $(\mathcal{T})$
8:             $SW[k] = l$
9:         end for
10:        Call GREEDY MAPPING with taskset $(\mathcal{T})$
11:        if Resulting configuration is valid then
12:            Return resulting system configuration
13:        end if
14:    end for
15: end for
16: Return algorithm failure
```

2.3.3.1 Simulated Annealing

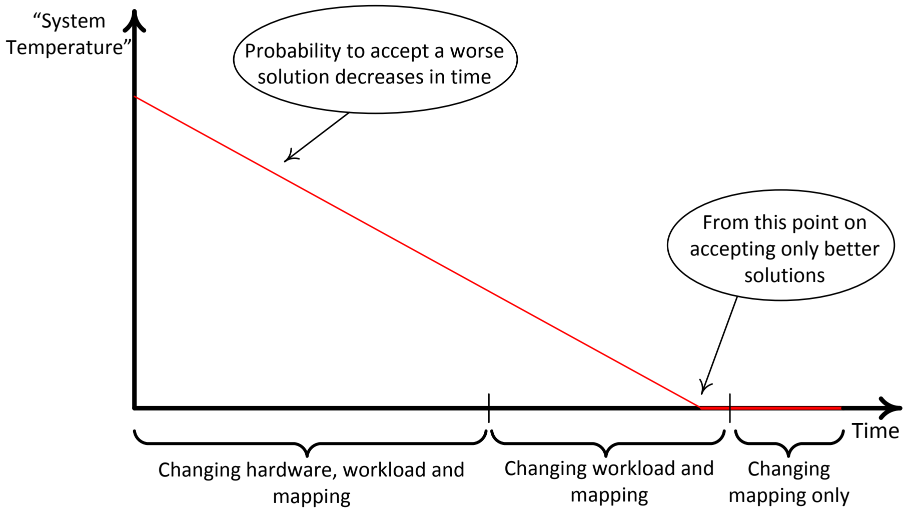
Simulated annealing (SA) is a probabilistic heuristic for optimization problems like the one defined in this chapter [75]. Its name comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to reduce its defects. The notion of slow cooling is implemented in the simulated annealing algorithm as a slow decrease in the probability of accepting solutions that are worse than the current best. During the early steps of the algorithm the “system temperature" is high and there is a good chance to accept a new candidate solution which scores lower than the current one, to facilitate escaping relatively low local maxima and widen the part of the solution space that is actually explored. As the “system temperature" drops, this probability is reduced.

The following steps were taken to adapt SA to our problem:

- **Initial solution formulation:** The solution from which the search starts is based on the precomputed partial solution for the particular scenario, as described in Section 2.3.1. This results in a good chance that the initial solution will be valid. This is not absolutely required, but it speeds up the convergence process, because forming new solutions is somewhat incremental (see next point), thus it takes time to move from an invalid solution to a good one.

- **Candidate solution formulation:** The next candidate solution is formed in three steps: First, for components that have alternative configurations in the same degradation level (that is, not fault-free or entirely failed components), there is a probability to change to the alternative configuration. Second, two tasks of the workload are chosen at random and the next available version is chosen for each of them. Third, these two tasks are remapped, on a random suitable component.

- **Candidate solution acceptance:** A candidate solution with better quality than the current one is always accepted. A worse solution has a chance to be accepted,



"System Temperature"
Probability to accept a worse solution decreases in time
From this point on accepting only better solutions
Time
Changing hardware, workload and mapping
Changing workload and mapping
Changing mapping only

Figure 2.7: Simulated annealing: The function of temperature over time (linear decrease) and its effect on the elements of the solution that can be changed. Note that, as mentioned in the text, hardware configuration and workload selection are locked after a certain number of iterations. Also, when temperature reaches zero, so does the probability of accepting a solution worse than the current one.

depending on the current "system temperature" and how lower the quality of the new solution is. "System temperature" decreases in a linear manner and reaches zero before the end. Thus, during the last iterations, there is zero probability to accept a worse solution, to allow the algorithm to climb whichever local maximum it has approached until then.

- **Additional effects of temperature:** During the first steps, all three elements of the solution are open for modification. After half of the steps have been completed, the hardware configuration currently selected is locked in place and the first step of forming a new solution is removed. During the last 20% of the steps, the same happens with workload selection, so the last steps modify only the task mapping. We have thus made use of the temperature concept to better tailor SA to the particular problem.

When all steps are completed, the algorithm returns the best solution it has visited during the run - which is not necessarily the current solution at the end of the run. The effect of "system temperature" on the algorithm process is illustrated in Figure 2.7.

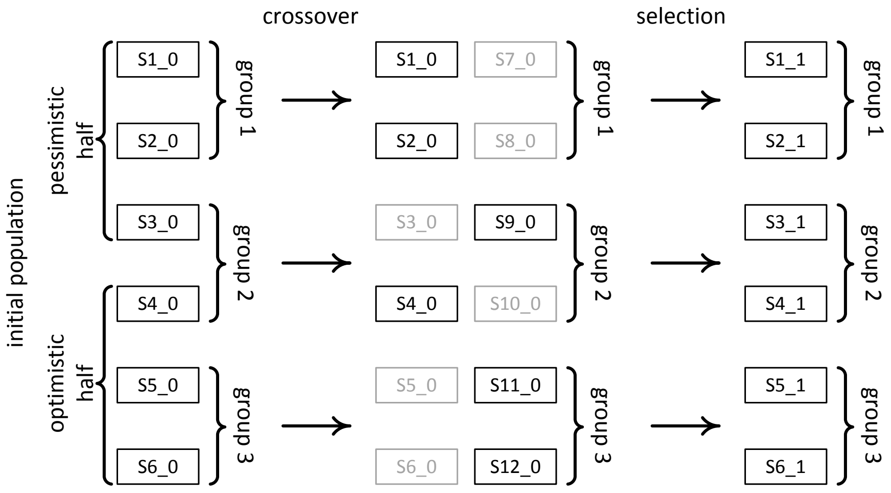


Figure 2.8: Genetic algorithm: the process of obtaining generation 1 (through crossover and selection) from the initial population (generation 0) is illustrated. For each solution Si_j, i is the solution index while j is the generation index.

2.3.3.2 Genetic Algorithm

A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection [76]. It functions by keeping a population of solutions, which it modifies by combining two or more existing members into new ones. In principle, worst solutions are discarded in favor of better ones, so that the best ones make it to the end. The upkeep of a population is the main disadvantage of the GA compared to all other algorithms we have described: While our heuristics and SA keep a few solutions stored at any given moment (the current one, a temporary one and, in the case of SA, the best solution so far), the GA is based on constantly having a population of solutions stored. On the other hand, it is also expected to produce better results.

When adapting the method of GA to our problem, the biggest challenge we faced was the fact that randomly combining parts of different solutions has a low probability of producing a valid offspring (e.g., random remapping of a task can result in illegal mappings). To mitigate this effect, operations which are constrained to produce valid solutions can be defined, with the caveat that they have to perform time-costly checks and are thus slower. As a result of this, we had to choose between a non-checking approach that produces some valid solutions by maintaining a large population and a more guided approach that produces fewer, but valid solutions with slower operations. After experimenting with both options, we opted for smaller population, fewer iterations (called generations in a GA) and slower, guided operations. In particular:

- **Initial population:** We define the initial population in such a way that most of its members are valid solutions. The hardware configuration is determined by selecting a valid option at random for each component. For half of the members the workload is determined based on the partial precomputed solution used also in the previous algorithms (henceforth called pessimistic half). For the second half, a random workload is selected (henceforth called optimistic half). Each member is annotated with the half it belongs to, to be used during crossover (see below). The mapping part happens by selecting a random *legal* mapping for each task. The time invested in shaping a strong and diverse initial population pays off in the long run. The initial population is considered "generation 0". A sample initial population of six solutions is shown on the leftmost part of Figure 2.8. Note that each solution on the figure is named Si_j, where i is the solution index, while j is the generation index. Thus, the initial population consists of solutions S1_0 to S6_0.

- **Crossover:** During the first generation crossover, solutions are paired up. Three groups of pairs are formed: one with both members of each pair belonging to the pessimistic half (group 1), one with both belonging to the optimistic half (group 3) and one mixed (group 2). The pairs of each group are then crossed over in different ways. For instance, pairs of group 2 and 3 cross their workloads, which wouldn't make sense for group 1, since all its members have the same workload. All crossover functions are guided to have a good chance of producing valid solutions as offsprings. In any case, each pair of solutions produces two new ones. Figure 2.8 shows how, after crossover, solutions S7_0 to S12_0 are formed. Note that for the small sample population in the figure, there is only one pair of solutions in each group, which is not the case in the real implementation.

- **Selection:** After each pair of solutions produces two new ones, the two best solutions are chosen between parents and offsprings, to be parents in the next generation. This is a type of *tournament selection* and we choose to use it to keep the pairs fixed until the end of the algorithm run (unless a mutation takes place - see next point), in order to apply the suitable crossover function to them. Additionally, it saves time compared to selecting the top 50% of the whole population, which would require sorting all solutions in terms of quality. Returning to Figure 2.8, the selection step reduces the population back to six solutions, by selecting the two best (not grayed out) from each group of four. Note that the result of selection, solutions S1_1 to S6_1, is the population of generation 1.

- **Mutation:** The choices we made for crossover and selection, reduce the entropy of the population a lot - making it possible to become stale. To balance this, we defined three mutation functions: The first causes members of group 1 to have their (otherwise fixed) workload randomly changed to inject some diversity. The second randomly changes the hardware configuration of a random solutions. The third swaps the places of two random members of the population before crossover, to stir up the static pairings.

When the predefined number of generations is produced, the best out of all alive solutions is selected and returned by the algorithm.

2.4 Evaluation

This section presents the experimental results we obtained by simulating our runtime manager on a desktop computer and emulating it on a real multicore machine, the Intel SCC [77]. We evaluate all our algorithms versus an optimal solver, by measuring the *System Efficiency* (SE) achieved by them as multiple faults occur on the system successively, as well as their average response time for generating a new system configuration after each event.

Subsection 2.4.1 describes our experimental setup, outlining how we performed the experiments, how we injected permanent faults throughout the system's lifetime and how we built our workloads. Subsection 2.4.2 explains how we fine-tuned our implementations of SA and GA. Subsection 2.4.3 presents the results of each algorithm with respect to *System Efficiency* throughout the system's lifetime. Lastly, Subsection 2.4.4 compares the algorithms with respect to their execution time, with an increasing problem size (number of components and tasks).

2.4.1 Experimental Setup

To set up our simulations, we consider a heterogeneous system such as the one shown in Figure 2.3, with four cores, two of which are little, power efficient cores (e.g. RISC) and two of them faster, big processors. Referring to the objective function (see Section 2.2.2), we set the weights to $w_1 = 0.4$, $w_2 = 0.2$ and $w_3 = 0.4$.

2.4.1.1 The Intel SCC as an Emulation Platform

To strengthen the evaluation of our GD manager, we looked for a platform with as many of the characteristics of the system illustrated in Figure 2.3 as possible. We found the Intel Single Chip Cloud Computer [77] to be a good choice. First of all, it consists of a number of identical worker cores, divided into six power domains. The platform supports independent frequency scaling for each of the six domains, enabling performance degradation. On top of that, the worker cores are controlled by a host PC, which can play the role of the reliable runtime manager of Figure 2.3.

Although the SCC helps us recreate many important properties, it misses the characteristic of heterogeneity. For this reason, we are forced to emulate it by "naming" each core as, e.g. little or big, and using different frequencies for the two kinds, to emulate performance heterogeneity. According to its kind, each core runs an appropriate version of the tasks, each version annotated differently to reflect the core characteristics.

2.4.1.2 Event Generation

As already mentioned, for this study, “events” are essentially permanent faults, taking place during the system’s lifetime. Based on that, the event generator has to provide a series of permanent faults, each of them annotated with a time stamp and a location (denoting the hardware component on which the fault appeared). Each fault is further annotated with a severity level. Up to this point, we have assumed that faults are isolated using hardware reconfiguration. There are permanent faults, however, that are inconveniently placed (e.g. on single points of failure), making it impossible to salvage any part of the component. The event generator takes this possibility into account as well.

We based our approach on traditional reliability analysis [78], according to which, the reliability R and probability of failure F of a component as functions of time, have the failure rate λ as a parameter, which is for now assumed constant. Intuitively, we would like the probability of a component failing within a small interval Δt to be equal to $\lambda \cdot \Delta t$. Relaxing the assumption of constant λ, we can generalize and arrive to the following:

$$P(\text{fault in } \Delta t) = \int_{t}^{t+\Delta t} \lambda(S)\ \text{dS}$$

Observe that if $\lambda(t)$ is constant, the above is reduced to $\lambda \cdot \Delta t$, which was our initial intention.

According to traditional reliability analysis, the reliability $R(t)$ of a hardware component has the following form, which is the cumulative distribution function of an exponential distribution with parameter λ.

$$R(t) = e^{-\lambda t}$$

The probability of failure up to any moment in time $F(t)$ can be accumulated accordingly:

$$F(t) = 1 - R(t) = 1 - e^{-\lambda t} \qquad (4)$$

For this work, we need to predict both faults that cause complete failure of a component and faults that cause it to degrade and perform simulations with sequences of multiple faults. Thus, we define for each component of the system multiple $R(t)$ functions, being evaluated concurrently. Each one of them decides that a fault has occurred, when the non-decreasing function of t, $\int_0^t \lambda(S)\ \text{dS}$ reaches a random, exponentially distributed threshold E.

The aforementioned threshold E is constructed as follows: Equation (4) expresses the probability that a generator determines the occurrence of the corresponding fault on some moment up to and including time t. By solving for t, we get the inverse of F:

$$F^{-1}(y) = -\ln(1-y)/\lambda \qquad (5)$$

and using inverse transform sampling as explained in [79], we can generate exponentially distributed numbers. More specifically, by plugging a uniformly distributed number

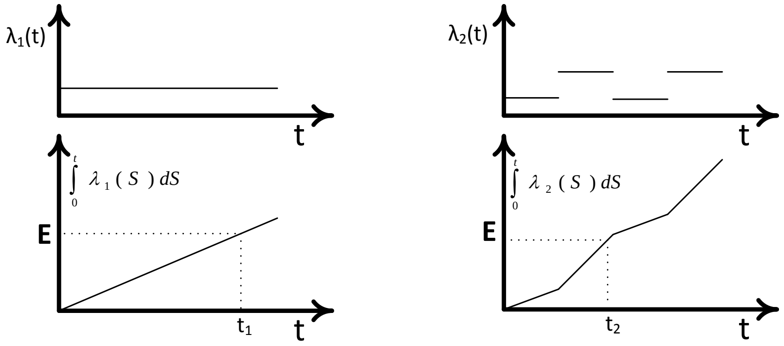


Figure 2.9: Fault generation visualization for two different functions for $\lambda(t)$: constant λ (left) and λ following a step function (right). The integral of $\lambda(t)$ determines the climbing rate, while the exponentially distributed threshold E introduces the necessary randomness. When the integral function catches up to the threshold, an event is generated (timestamps t_1 and t_2 for each of the two cases respectively).

$y \in [0, 1)$ in $-\ln(1 - y)$, which is (5) without the impact of λ, we get exponentially distributed number E, which will be used as a threshold in the fault prediction.

The climbing of $\int_0^t \lambda(S)$ dS towards the limit E is illustrated in Figure 2.9. Two different versions of $\lambda(t)$ are drawn, constant (λ_1) and a step function (λ_2). The figure demonstrates how the magnitude and behavior in time of $\lambda(t)$ determines the rate of the climb towards E, while E introduces the necessary randomness. The timestamps t_1 and t_2 are the moments of fault occurrence for each of the two cases.

Being able to deal with any function $\lambda(t)$, makes our event generator generic: We can use it to generate fault sequences under any kind of assumptions. The rates of various effects depend on a variety of factors, both internal to the system (e.g. workload of components) and external (e.g. ambient temperature or radiation). These factors can easily be plugged in our tool as parameters of $\lambda(t)$, as needed for future experiments.

We used the event generator to define two different fault scenarios:

- A “baseline” scenario. According to this, on every component there exist four fault generators (corresponding to four potential faults). Three of these represent partial failures that can be isolated with reconfiguration and have a constant fault rate of $\lambda_1 = 2.55$ faults in the expected total lifetime. The fourth generator injects a critical fault with a rate $\lambda_2 = 1.7$ severe faults in the same time.

- A “weak component” scenario, in which one random system component has 2.5 times the above rates.

We set the expected lifetime to 1000 days. Thus, the above fault rates projected to one day are $\lambda_{1d} = 0.00255$ and $\lambda_{2d} = 0.0017$ faults/day. Based on these scenarios, the event generator predicts what faults will occur on the system during this period of

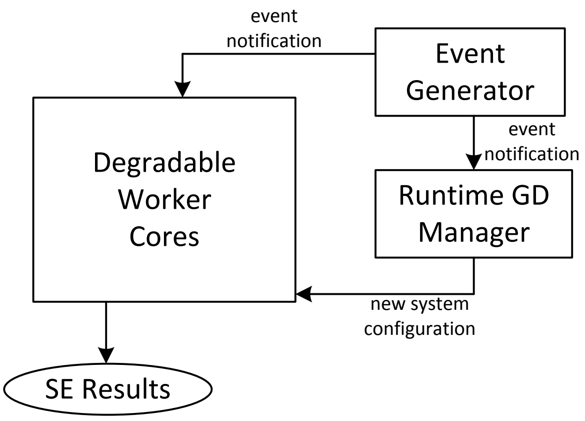


Figure 2.10: Process followed by simulations and emulations: in simulations, System Efficiency (SE) results are obtained by evaluating the objective function based on profiling data, while during SCC emulations, measured performance is considered instead.

1000 days and generates a list. Multiple such lists are generated for each scenario, each corresponding to a different sequence of faults, each of which is random but based on the same defined fault rates. These "fault lists" are then used during simulations and SCC emulations, to trigger the runtime GD manager, which in turn has to produce a new system configuration, as shown in Figure 2.10. Between events, System Efficiency is evaluated, either completely based on profiling data for each task's execution time and energy (simulations) or by measuring the actual execution times of tasks on the SCC (emulations).

The SE function in time for a single simulation or emulation is thus a step function, as each event causes a SE degradation. Between successive events, SE is stable. The SE-in-time results we present here are the averages over of a number of such experiments: each algorithm is evaluated through 400 simulations and 30 SCC emulations for each fault scenario. Any system that fails is considered to have an efficiency of 0 until the end of its expected lifetime. For fairness, we use the same 400 (or 30) fault lists for all algorithms.

2.4.1.3 Workloads

On the workload side, we need to have workload flexibility as our description dictates. We construct one flexible use case for each of the two application models.

For the simple application model, without task dependencies, we base our use case on Mibench benchmarks [80], three from the automotive suite (basicmath, bitcount, qsort - all considered essential tasks) augmented with four more benchmarks from the network, consumer and telecommunication suites - which represent the optional features delivered by the system (Susan, GSM, Dijkstra, Audio Decoder).

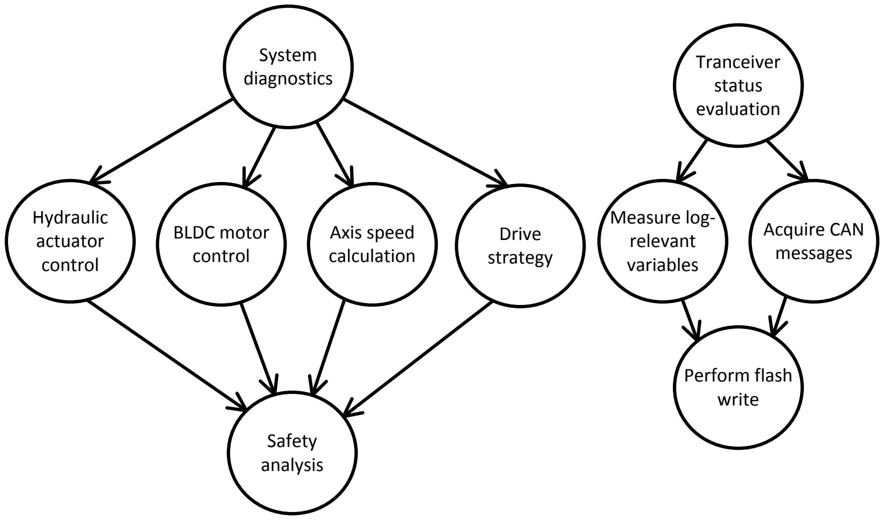


Figure 2.11: The DAG of the automotive use case.

For the case of an application described by a DAG, we use a mock-up of a real-world example, a typical automotive application from the Powertrain domain, developed according to the AUTOSAR specification [81]. It is an embedded system consisting of sensors and actuators, tasked with controlling the gearbox of a vehicle. Some of the tasks (the ones in the leftmost DAG of Figure 2.11) are, according to ISO26262 [82] considered safety relevant, and failure of any one transitions the system to a safe state, resulting, for our intents and purposes, to a system failure. Specifically, the top node of the DAG runs system diagnostics. In the middle level, from left to right, tasks are responsible for: performing closed loop control of the hydraulic actuator; controlling the *Brushless DC motor* (BLDC); using sensor data to calculate axis speed; and determining the drive strategy (gear selections) based on driver input. The bottom node integrates the above, performing safety analysis and reacting to errors. The rest of the tasks (belonging to the rightmost DAG) are part of the application layer which is not safety relevant and are tasked with logging system data. Specifically, the top node checks the status of the transceiver and memory. The left and right middle nodes measure relevant system data and acquire relevant messages respectively. Lastly, the bottom node updates the log file in the flash memory. Inability of the system to perform these tasks results to degraded functionality.

2.4.2 Fine-Tuning of SA and GA Algorithms

In Section 2.3.3 we presented our implementations of the SA and GA algorithms. At this point we present some indicative results of these algorithms for different sets of parameters and justify the choice of version for each of them, to be used in the subsequent experiments.

Simulated Annealing: We have implemented SA with and without a convergence criterion. A convergence criterion means that SA will terminate if, within a window of iterations, it does not improve its result by a significant value. The size of the window as well as the expected improvement are parameters. This early termination cuts down on execution time, sacrificing a chance to find a more efficient solution in the remaining iterations. At the top part of Figure 2.12, we observe that this efficiency penalty is less than 2% of the system efficiency scale, thus we choose the convergence criterion version for the subsequent experiments. However, no version is strictly better and the decision depends on the combination of optimization goals.

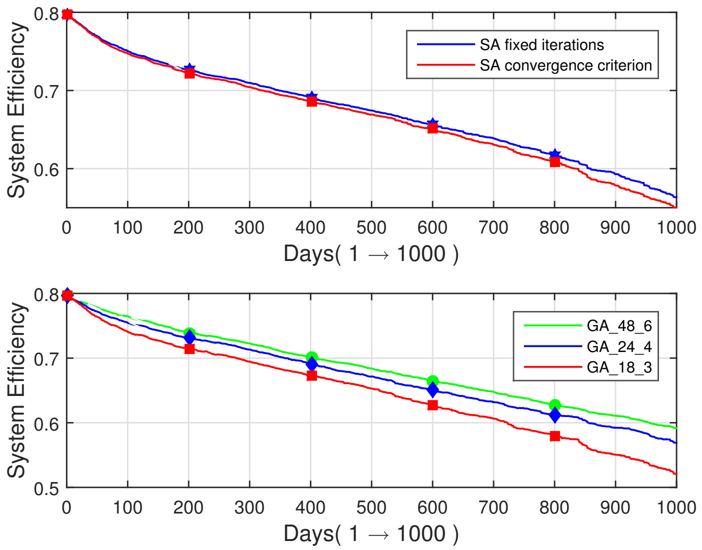


Figure 2.12: Fine-tuning of SA (top) and GA (bottom) algorithms.

Genetic Algorithm: The genetic algorithm on the other hand, operates on a population of predefined size for a sequence of generations. The lower part of Figure 2.12 illustrates part of our exploration for fine-tuning these parameters. We call the genetic algorithm versions GA_X_Y, where X is the population size and Y the number of generations. We observe that, close to the end of the simulation, GA_24_4 trails behind GA_48_6 by less than 2.5% of the System Efficiency scale, whereas GA_18_3 trails GA_24_4 by more than 4% of the same scale. We choose to use GA_24_4 for the subsequent experiments, again noting that no version is strictly better, as scaling down the algorithm has execution time and memory utilization benefits, while scaling it up provides more efficient system configurations.

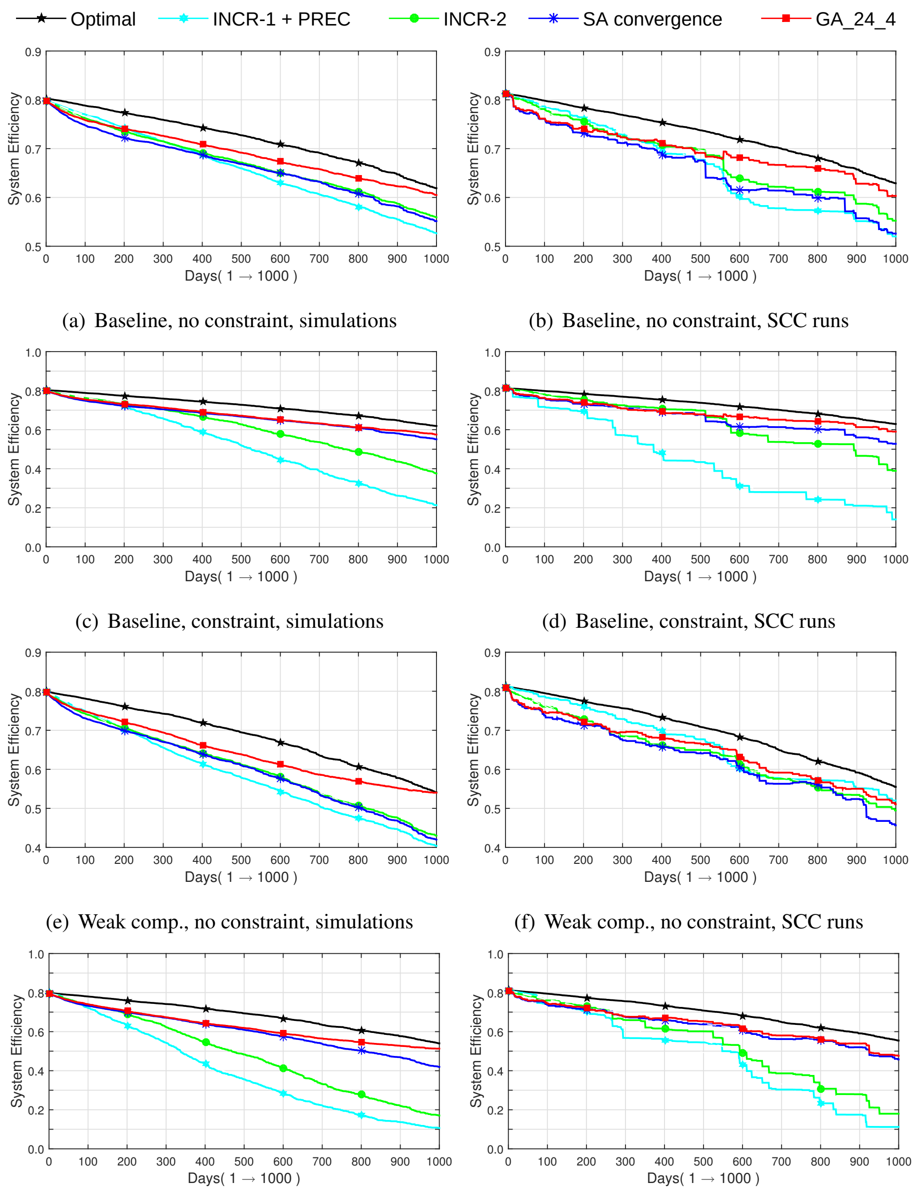
2.4.3 Lifetime System Efficiency Evaluation

After settling with one version of each algorithm, we go on to compare them in terms of achieved System Efficiency (SE), using the experimental setup described in Section 2.4.1. For this evaluation, we use the "baseline" and "weak component" fault scenarios described in Section 2.4.1.2 and run the experiments without and with system reconfiguration cost constraint, as this was defined in Section 2.2.3. For the simple application model we compare the following algorithms: INCR-1 followed by PREC (*INCR-1 + PREC*), *INCR-2*, SA with convergence criterion (*SA convergence*) and *GA_24_4*. For the application described by a DAG, we compare the following algorithms: *DAG-GD*, SA with convergence criterion (*SA convergence*), and *GA_24_4*.

Results for SE over time for the simple use case are illustrated in the plots of Figure 2.13. The topmost solid line of each plot indicates the SE achieved by a runtime manager which always chooses the optimal configuration. Colored solid lines show the SE of the various algorithms. The left-side plots illustrate simulation results, while the right-side plots show SCC emulation results. Note that each line consists of 1000 data points (one for each day). We stress one point every 200 days with a different symbol for each algorithm, merely to make visual comparison easier. Also note that SCC emulation results show more abrupt changes (occasionally resembling step functions) because of the much lower number of total experiments, whereas for simulation results, the increased number of experiments smoothens the lines.

We observe that, when reconfiguration cost constraints are not considered, the custom heuristics keep up quite well with the more complex algorithms. Specifically, in the baseline scenario, both *INCR1+PREC* and *INCR2* deliver higher efficiency than *SA convergence* and *GA_24_4* for the first 20% to 30% of the total expected lifetime and are only 8% (*INCR2*) and 13% (*INCR1+PREC*) worse than *GA_24_4* and on par with *SA convergence* near the end of the lifetime. The early advantage is due to the incremental nature of the solutions produced by the custom heuristics, resulting to linear degradation in the early stages, while the other algorithms show a small dip in efficiency and only after this they start to degrade linearly. In the weak component scenario, this advantage is less apparent, as the weak component often produces an early critical failure, thus the early advantage in the simulations lasts only for the first 10% of the expected lifetime and the disadvantage compared to *GA_24_4* near the end grows to 20% and 26% for the two custom heuristics. However, the custom heuristics improve substantially in the SCC experiments of this scenario, as *INCR1+PREC* remains better for the first 55% of the lifetime and is on par with the more complex algorithms near the end.

When introducing a reconfiguration cost constraint, the custom heuristics show comparatively worse results. The early advantage period is now shorter (between 5% and 20% of the total lifetime) and, near the end of the experiment, they lag behind *GA_24_4* by 34% to 44% (*INCR2*) and 60% to 80% (*INCR1+PREC*). The deterministic nature of the custom heuristics makes it harder to satisfy the constraint, contrary to the other algorithms that are allowed to try a higher number of random solutions and, during the process, discard invalid ones. As non-satisfaction of the constraint leads to system failure, this penalizes custom heuristics' efficiency heavily.



(a) Baseline, no constraint, simulations

(b) Baseline, no constraint, SCC runs

(c) Baseline, constraint, simulations

(d) Baseline, constraint, SCC runs

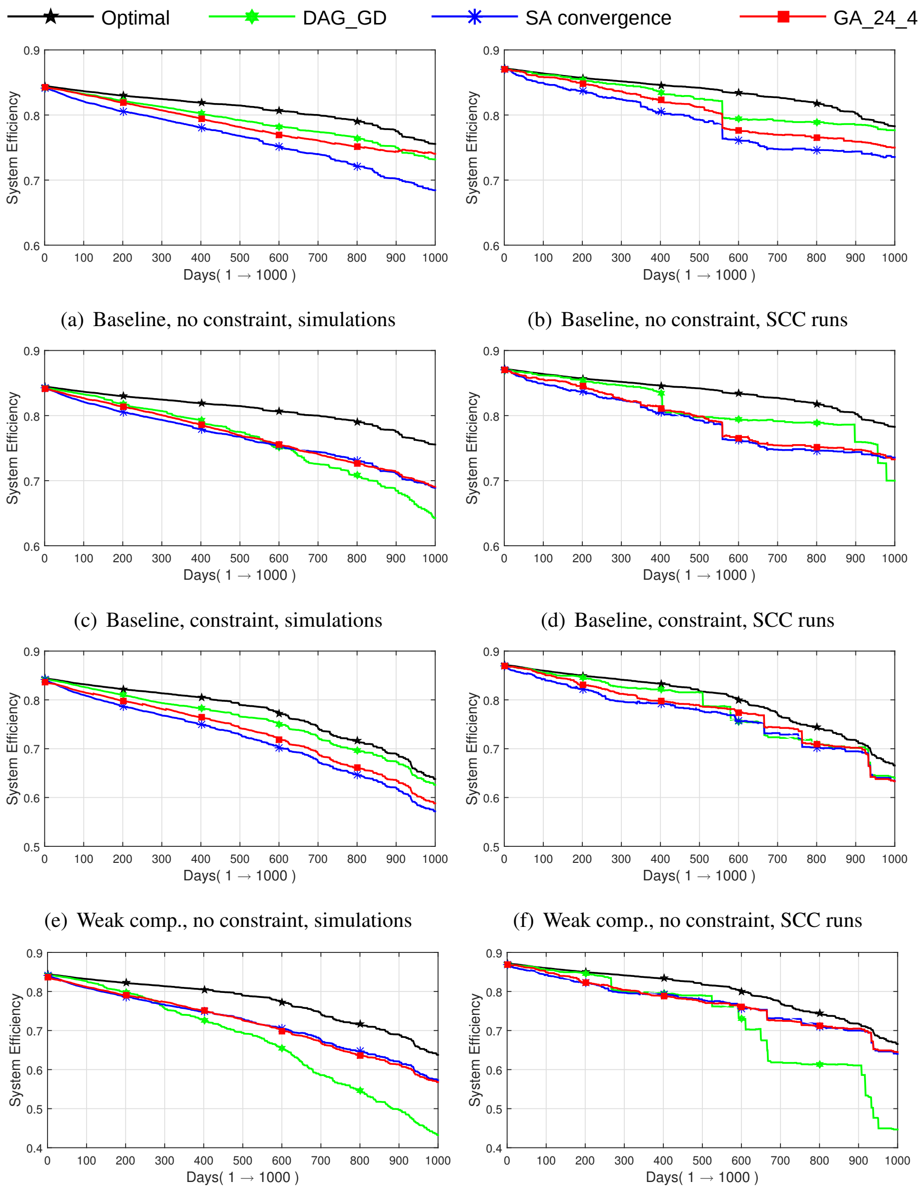
(e) Weak comp., no constraint, simulations

(f) Weak comp., no constraint, SCC runs

(g) Weak comp., constraint, simulations

(h) Weak comp., constraint, SCC runs

Figure 2.13: System Efficiency in time for the use case of standalone tasks, both fault scenarios, without and with reconfiguration cost constraint. The black line at the top of each plot is the optimal SE. For each scenario, simulation results are shown on the left plot and SCC results on the right.



(a) Baseline, no constraint, simulations

(b) Baseline, no constraint, SCC runs

(c) Baseline, constraint, simulations

(d) Baseline, constraint, SCC runs

(e) Weak comp., no constraint, simulations

(f) Weak comp., no constraint, SCC runs

(g) Weak comp., constraint, simulations

(h) Weak comp., constraint, SCC runs

Figure 2.14: System Efficiency in time for the DAG use case, both fault scenarios, without and with reconfiguration cost constraint. The black line at the top of each plot is the optimal SE. For each scenario, simulation results are shown on the left plot and SCC results on the right.

Lastly, comparing to the optimal manager, we observe that, even at the end of the expected lifetime, at least one algorithm (usually *GA_24_4*) delivers average system efficiency less than 15% worse than the optimal. This is the widest margin observed throughout the whole expected lifetime. In most cases the best algorithm stays within a margin of 10% from the optimal until the end.

The same results for the DAG application are shown in Figure 2.14. In this use case, we observe that the custom heuristic, *DAG-GD* does better in both fault scenarios when there is no reconfiguration constraint. It stays ahead of *SA convergence* and *GA_24_4* for 55 to 100% of the total lifetime, and is 7% better to 2% worse at the end. When a constraint is enforced, *DAG-GD* is punished less than the simple case custom heuristics, *INCR1+PREC* and *INCR2*. It still remains better for the first 25% to 60% of the total expected lifetime and only shows a sharp decrease of efficiency during the second half of the experiments, resulting to a final efficiency only 6% to 30% worse than *SA convergence* and *GA_24_4*.

Furthermore, for the DAG use case, the best algorithm always stays within a 10% efficiency margin compared to the optimal. The largest discrepancy we can observe is for the simulations of both the baseline and weak component scenario with a constraint, for which the best algorithm is 9% worse than the optimal, close to the end of the experiments. Lastly, for this use case, *GA_24_4* is not as much better than *SA convergence* as it was in the simple use case. In six of the eight plots we see the two algorithms performing similarly, and *GA_24_4* is visibly better only for the baseline scenario without constraint.

2.4.4 Response Time

In this Section we present results about the response time of each algorithm, for the system which we used for our experiments, as well as systems consisting of 2, 4 and 8 times more components as well as tasks, to observe how this response time scales with the problem size. Response time is important for applications that need to be back up and running as soon as possible in the event of a component failure. This observation also stresses the importance of a system reconfiguration cost constraint, as was used in half of our experiments: The sum of the algorithm response time, plus the system reconfiguration time is equal to the time the system will remain offline because of a *component failure* event.

Execution times for all algorithms and for both use cases are given in Table 2.1. In each subsequent column, we multiply both numbers of system components (κ) and application tasks (τ) by two, resulting to a four times larger problem size. For this experiment, we mainly care about how execution times of different algorithms compare to each other and how they scale with respect to the input size. Thus, execution time is measured on the machine most conveniently available to us - one logical core of an Intel i7-2600 processor of an office PC, clocked at 3.4 GHz. All execution times are expressed in μsecs. Lastly, the table also lists some versions of SA and GA that were rejected in fine-tuning, to give a better perspective of the execution time - solution quality tradeoff.

We observe that most algorithm response times roughly agree with their asymptotic

Table 2.1: Execution time (in μsecs) on a typical office PC for different algorithms and problem sizes, expressed in number of components (κ) and tasks (τ).

Individual Tasks Use Case				
Problem Size:	$\kappa = 4$ $\tau = 7$	$\kappa = 8$ $\tau = 14$	$\kappa = 16$ $\tau = 28$	$\kappa = 32$ $\tau = 56$
INCR1 + PREC	1.8	2.2	12.2	12.4
INCR2	1.6	2.1	18.8	22.5
SA fixed	60.8	119.1	165.4	208.5
SA convergence	22.7	36.1	51.6	81.0
GA_48_6	128.3	210.2	348.2	597.8
GA_24_4	59.0	91.1	165.0	306.1

DAG Use Case				
Problem Size:	$\kappa = 4$ $\tau = 9$	$\kappa = 8$ $\tau = 18$	$\kappa = 4$ $\tau = 36$	$\kappa = 32$ $\tau = 72$
DAG_GD	3.5	6.7	21.4	29.1
SA fixed	73.7	142.2	222.1	280.9
SA convergence	27.3	49.7	73.7	91.6
GA_48_6	159.3	312.3	463.8	720.9
GA_24_4	77.3	146.9	219.2	332.0

complexity. One might have expected the response time of a $\mathcal{O}(\kappa * \tau)$ algorithm to increase by a factor of 4 when both κ and τ double, but we have already mentioned that the loop boundaries which determine the complexity are quite pessimistic. Specifically, for a problem size increase of 64 times (that is, comparing the first and fourth column of the table), the execution time of the three heuristics grows by a factor of 6.9 (*INCR1 + PREC*), 14 (*INCR2*) and 8.3 (*DAG-GD*). Thus, they can be used on arbitrarily large systems, without concerns about their execution time.

Furthermore, despite the fact that execution time of Simulated Annealing and Genetic Algorithm grows at a relatively slower pace, these algorithms are noticeably slower for all problem sizes. *SA_convergence* is one order of magnitude slower for the smallest problem size and 3 to 7 times slower for the largest problem size, compared to custom heuristics. Respectively, *GA_24_4* is 22 to 37 times slower than custom heuristics for a small size and remains 11 to 25 times slower for the largest size. However, the difficulty of custom heuristics to satisfy reconfiguration cost constraints hints at the idea to use a combination of algorithms to achieve both low average response time and constraint satisfaction. A custom heuristic can be used first and, in case it fails to find a valid configuration, it can be backed up by *SA* or *GA*.

2.5 Degradable Component Characterization

So far in this chapter we have assumed the existence of degradable hardware components. Furthermore, we have claimed that it is possible to characterize these components with respect to their performance and energy efficiency degradation, whenever reconfiguration is used to isolate a permanent fault. This characterization produces the necessary annotations that are part of the high-level system description provided in Section 2.2. In this section, we present one methodology to perform such characterization, focusing on a popular paradigm of fault-tolerant components: reconfigurable arrays of microprocessors consisting of Substitutable Units (SUs) [9, 11, 52, 55, 59].

Subsection 2.5.1 that follows, contains some necessary background on how these reconfigurable processor arrays work and a brief description of the steps we take for their characterization.

2.5.1 Background

One strategy which has been very popular in recent years is reconfigurable processor arrays [52] [10] [11] [9]. It seems that pipeline stage granularity is currently considered a promising choice in combined terms of permanent fault tolerance and related overheads. However, as was demonstrated in [55], the ideal SU granularity can be systematically determined based on the overheads imposed by reconfigurability.

The concept of reconfigurable processor arrays is illustrated in Figures 2.15 and 2.16. In a nutshell, the array consists of a number of *components* (e.g. microprocessors), all of which work normally in the fault-free case. Each component is comprised of smaller parts called *Substitutable Units* (SUs). As Figure 2.15 shows in detail, there are interconnections between SUs, which can be configured using switches. As Figure 2.16 shows, this reconfigurability can be used to connect SUs coming from different faulty components, to form new fault-free components. Optionally, the array can be augmented with FGPA-like, fine-grain reconfigurable fabric, providing more flexibility: any type of SU can be instantiated on this fabric, allowing the formation of more components in the case that many SUs of the same type become faulty. As an example, for the array of Figure 2.16 three components can be salvaged using the fine-grain fabric, but only two without it. This flexibility comes at a significant area cost.

Referring to the visual representation of Figure 2.16, the components are also called *rows* and the sets of identical SUs are also called *columns* of the array. Furthermore, modifying connections between SUs is henceforth called *coarse-grain reconfiguration*. When the fine-grain fabric is also used, the respective term is *mixed-grain reconfiguration*.

In both means of reconfiguration, connecting SUs of different rows creates longer connections, penalizing the formed components in terms of performance. We have explored two options for this penalty to be expressed: longer connections resulting to significantly lower clock frequency and registers between the array columns resulting to extra cycles. Intermediate solutions have also been explored, like registers on every second row. In this thesis, we focus on the case of registers between columns on every row, as illustrated in Figure 2.15. We call these registers *bubble stages* (or simply *extra stages*), since each elongates the pipeline it belongs to by one stage that does not perform any actual computation. Clearly, having:

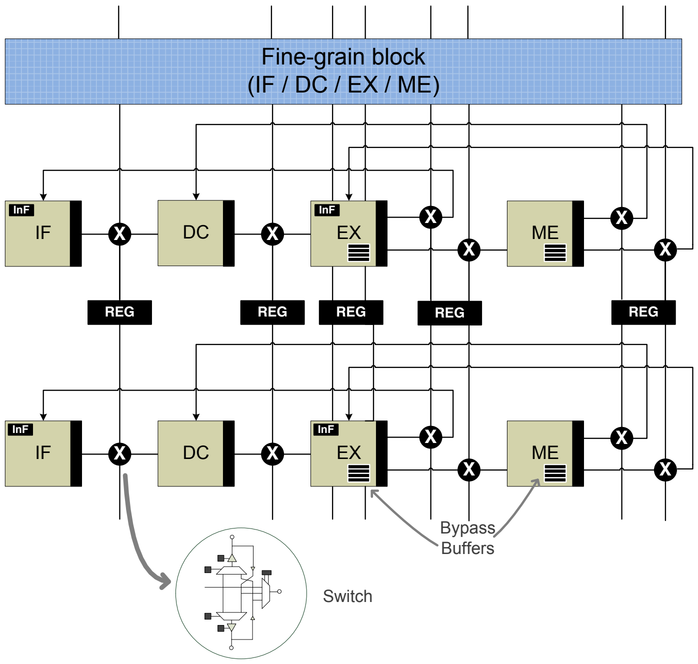


Figure 2.15: A reconfigurable processor array of two components with extra fine-grain fabric. Note the reconfigurable switches between columns, used to implement the connections between SUs and the registers used to avoid long wires that would incur clock frequency overheads.

- non-uniform interconnect delays,
- microprocessors of variable pipeline depth and
- the option to use the slower fine-grain reconfigurable fabric

results in performance and energy efficiency degradation, that is dependent on the array configuration. This fact stresses the need for characterization of the various components that can be formed, as well as the array as a whole.

The characterization process consists of four distinct steps. First we run a series of benchmarks on individual components made of SUs in the array and measure their performance and energy efficiency degradation, with respect to a non-reconfigurable component (Section 2.5.2). Second, we need a systematic way to obtain the status (also called *fault map*) of the array, in terms of which SUs are faulty and which ones still work, for any given scenario (Section 2.5.3). Third, we obtain, for each given fault map, an efficient configuration of the array, telling us how many components are formed and the type of each component (Section 2.5.4). Fourth, we combine the performance and

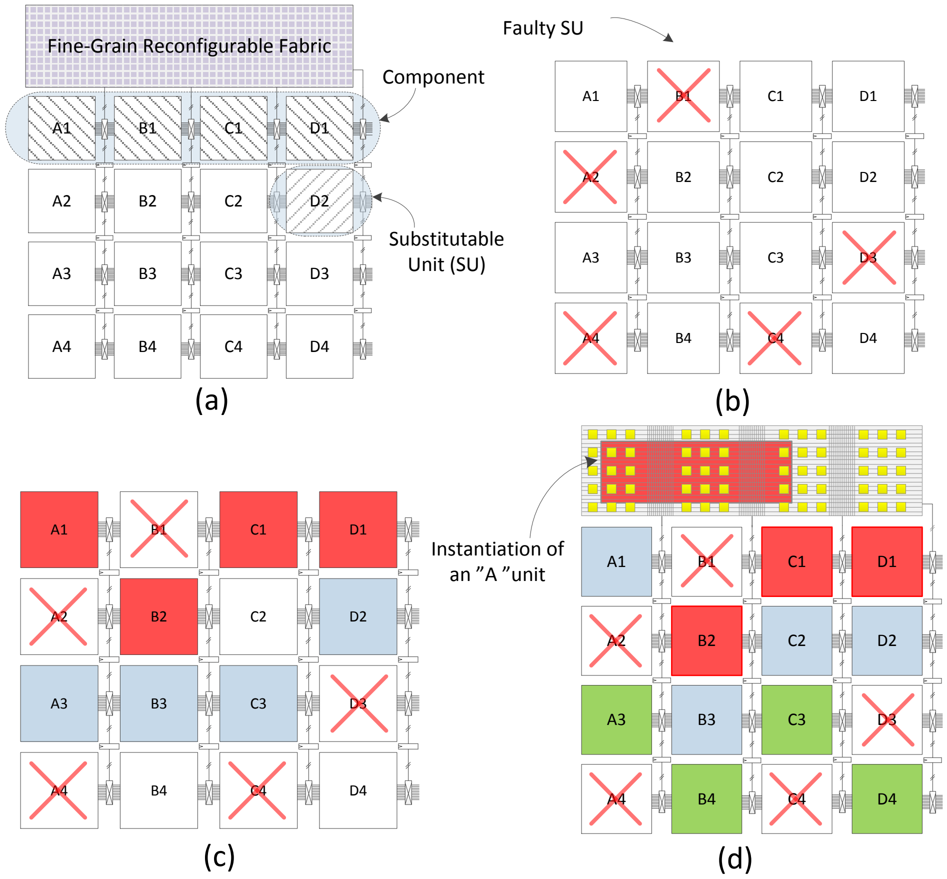


Figure 2.16: Illustration of permanent fault tolerance in reconfigurable processor arrays. An array with four components of four SUs each and fine-grain fabric (a). A status of the same array (with some faulty SUs) without fine-grain fabric (b) and a valid configuration given this status, producing two working components (c). If there is fine-grain fabric, the same status can produce a configuration of three working components by instantiating an extra SU of type “A” in it (d).

energy efficiency characteristics of each component, to characterize the array as a whole (Section 2.5.5). Section 2.5.6 contains some indicative characterization results for our implementation [52, 55], in the form of design space exploration for every combination of reconfiguration granularity and fault density.

2.5.2 Characterizing Individual Microprocessors

As explained in Section 2.5.1, a component which is result of reconfiguration is generally less efficient than a component of the fault-free array, which in turn is less efficient than a non-reconfigurable component. This degradation appears in the following ways:

- To make the processor array reconfigurable, we have to pay a flat performance, area and power penalty per component. All of these are due to the extra hardware required: Registers and switches between array columns, bypass buffers and instruction-flow registers for architectural consistency and extra wiring to connect all the above. The (optional) inclusion of fine-grain reconfigurable fabric incurs an additional area penalty for providing the extra flexibility. Given these flat overheads, the following points will focus on comparing a component in the fault-free state of the array with one that results from reconfiguration after the appearance of faults.
- In terms of performance (measured in Instructions per Second - IPS), any configuration other than the fault-free is slower than the latter. For coarse-grain reconfigured components, this performance penalty depends on the number of bubble stages in the component. For components making use of the fine-grain reconfigurable fabric, on the other hand, the performance penalty can be considered flat with respect to the bubble stages and only depends on which SU is instantiated on the fine-grain fabric, since the fine-grain part is much slower and limits the component performance severely - thus making the use of the extra registers pointless.
- In terms of power dissipation (measured in Watts), coarse-grain reconfiguration activates auxiliary components on the chip (switches and registers between SUs), thus making a reconfigured component more power-hungry. In the case of mixed-grain reconfigured components, power dissipation is drastically reduced by the major reduction of the clock frequency, sometimes resulting in a component less power hungry than the baseline one. This however does not mean that the component can be as efficient as a fault-free one (see below).
- The above two factors can be combined in the metric of *energy efficiency*, which expresses how much computation a component can perform with a given amount of energy and is measured in IPS/Watt. In the case of coarse-grain reconfigured components, the combined performance and power dissipation penalties guarantee a reduction in energy efficiency. In the case of components making use of the fine-grain fabric, what we gain in power dissipation is lost in performance (both being effects of the lower clock frequency). However, the performance penalty

is not redeemable, whereas the power gain is at least partially outweighed by the power dissipation of the fine-grain fabric, resulting in a less energy-efficient component overall in this case as well.

In order to be able to obtain performance, power and energy efficiency results for the whole array and perform the design space exploration presented in Section 2.5.6, we need to characterize the following kinds of individual components:

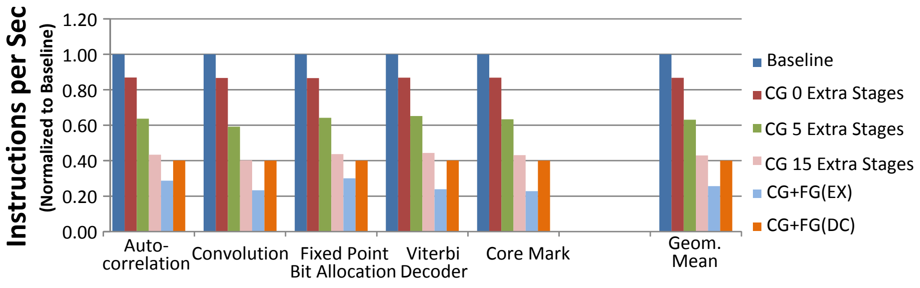
- A baseline, non-fault-tolerant component. That would be the processor we use to build the array, but without any reconfiguration capabilities.
- A reconfigurable, but fault-free component. In other words, one formed only by SUs of the same row in the reconfigurable array.
- Coarse-grain reconfigured components with various numbers of bubble stages, corresponding to the various processors that can appear on a faulty array.
- Mixed-grain reconfigured components. Performance and energy efficiency degradation depends on the type of the SU instantiated on the fine-grain fabric.

For a rapid processor implementation, a high-level description language (Lisa 2.0) was used, through the Synopsys Processor and Compiler Designer tool. With this toolset, RTL descriptions of our processors, as well as a simulator and a compiler were automatically generated. Of course, this advantage of rapid implementation comes at the cost of a less optimized RTL code, compared to a manually developed processor. Thus, the results presented here should be considered pessimistic. Cadence Encounter RTL compiler was used to synthesize our design at STM 65nm SP technology and to acquire measurements of area, power, energy and timing. Additionally, for the mixed-grain array, the different SUs were instantiated using a Xilinx Virtex-5 65nm FPGA substrate.

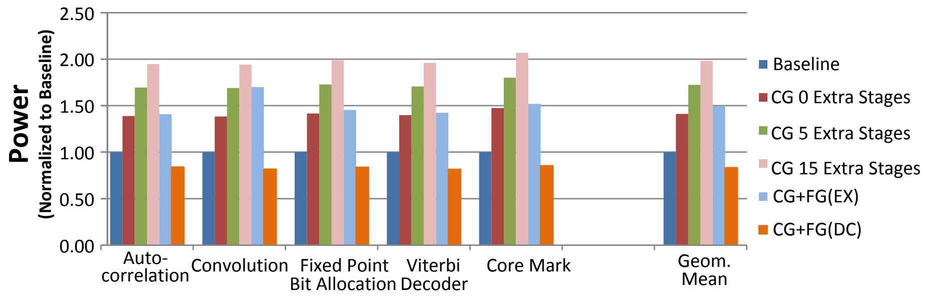
When using only CG parts, our adaptive processor maintains 90% of the baseline frequency, dropping from 500 MHz to 450 MHz. Pipelines also using FG logic (denoted as CG+FG) operate at 40% of the baseline speed (200 MHz), limited by the slowest FG implementation of an SU. These clock frequencies were used to obtain performance and power results.

For our evaluation we use a set of benchmarks taken from the EEMBC suite: Autocorrelation, convolution, fixed-point bit allocation, viterbi decoder and core mark. The results for the above benchmarks in terms of performance, power dissipation and energy efficiency for the various kinds of components are shown in Figure 2.17. "Baseline" refers to the non-reconfigurable component, while "CG N extra stages" is a coarse-grain reconfigurable component that has N bubble stages. Thus, "CG 0 extra stages" is the fault-free reconfigurable component. The last two columns represent mixed-grain components, with the execution and decode stage instantiated on the fine-grain fabric respectively. Based on the results illustrated in the graphs, we can make the following observations:

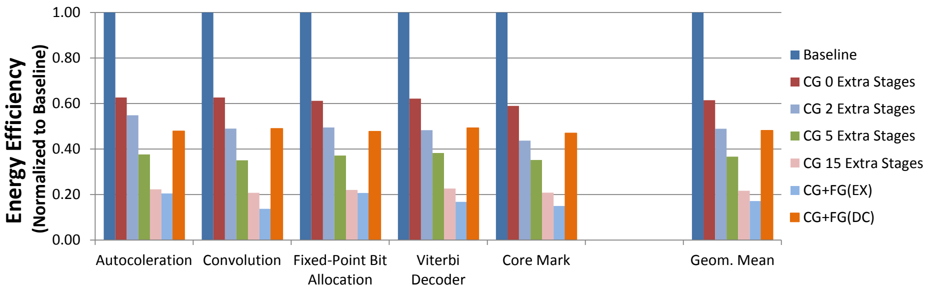
Performance: The fault-free CG configuration maintains 86% of the baseline performance and gradually drops to 73%, 63% and 43% when adding 2,5 and 15 bubble



(a)



(b)



(c)

Figure 2.17: Characterization of various kinds of individual components using the EEMBC benchmarks on an array of 6 rows and 4 columns. Performance (a), power (b) and energy efficiency (c). All results are normalized to the value of the baseline non-reconfigurable core.

stages, respectively. Implementing a spare DC or EX in the FG logic maintains 39% and 26% of the original performance, respectively, due to the lower frequency. Note that the missing numbers of bubbles between 0 and 15 follow the same trend.

Power: Our fault-free CG configuration consumes about 1.4 times more power than the baseline. The power consumption increases to 1.5, 1.7 and 2 times when 2, 5 and 15 bubble stages are inserted, respectively. When using FG logic for the DC stage, power drops to 83% of the baseline, due to the lower clock rate. In contrast, replacing the EX stage incurs a power penalty of 50%.

Energy Efficiency: CG configurations maintain 62%, 49%, 37%, and 22% of the baseline energy efficiency, for zero, 2, 5 and 15 extra stages, respectively. For a processor that uses FG logic for its DC or EX stage, energy efficiency drops to 48% and 18%, respectively.

2.5.3 Obtaining a Fault Map

The reason for having reconfigurability to begin with, is the potential existence of faults on the chip. To evaluate any solution based on a reconfigurable array, we need to have a systematic methodology of determining which SUs are faulty and which ones still working. Our methodology starts from a fault density (faults/unit of area) and produces a fault map for the array. Our fault injection methodology is based on the following assumptions:

- The spatial distribution of faults on the chip is uniform random. This is a pessimistic assumption, considering that faults which tend to appear in clusters would produce more favorable fault distributions: if more faults appear on the same SU, fewer SUs in total are faulty.
- The SUs in which each component is divided are of the same size.

As the starting point is the probability of failure of one transistor, obtaining the fault map is approached in a probabilistic manner. Indeed, a defect density does not directly imply how many (and which) SUs are functional or faulty. The relevant probabilistic analysis has been thoroughly presented in [51]. For the methodology presented here, the following paragraph summarizes the required knowledge.

As already mentioned, the input of our probabilistic model is the fault density. This parameter is used in two ways: First, it is projected to the total area of the processor array, which results in calculating an *expected value* for the total number of faults on it. This expected value is the one visible on the "# of faults" axis in the three dimensional graphs of Section 2.5.6. Secondly, the fault density is used to calculate the probability of *any* SU to be faulty.

Allow us at this point to take one step back and focus on the possible reconfigurable array statuses (fault maps). A status is a unique combination of fault-free and faulty SUs. Supposing an array with R rows and C columns and excluding potential fine-grain reconfigurable fabric, there are a total of $R * C$ SUs. Each SU has two possible statuses, fault-free and faulty, resulting in $A = 2^{R*C}$ possible statuses for the whole array.

Returning to obtaining a fault map, we have already calculated the probability for an SU to be faulty as a function of the fault density. Based on the assumption for uniform random spatial fault distribution, the status of each SU is statistically independent from the status of every other SU. This, combined with the same size assumption for SUs, means that the probability of occurrence of a status (P_{status}) for the whole array can be trivially computed as follows, based on the probability of one specific SU to be faulty (p), and the number of faulty SUs in the particular status (F_{status}):

$$P_{status} = p^{F_{status}} * (1-p)^{R*C-F_{status}}$$

Thus, at this point we know the probability of occurrence for each of the A different array statuses (fault maps). To make use of this to evaluate the array, we need to also find a proper configuration for the array based on each fault map. This process is described in the next section.

It is worth noting that the analysis that follows is motivated by the need to explore the design space defined by the possible degrees of granularity of a reconfigurable array. As such, the results of Section 2.5.6 present performance and energy efficiency for each combination of array granularity and fault density, based on the probabilistic fault maps explained in this section. However, the methodology as a whole is directly applicable to the problem of runtime management for GD. In fact, applying it in this context is even simpler, as the fault map is provided, in a deterministic manner, by the event generator described in Section 2.4.1.2. All other steps of the process (outlined in Sections 2.5.2, 2.5.4 and 2.5.5, do not change.

2.5.4 Obtaining a Configuration for the Array

As has been stressed throughout this chapter, whenever reconfigurability is used to react to the presence of permanent faults in a system, the problem is balancing the following two desirable but contradicting qualities:

- Finding an efficient configuration and
- finding a configuration fast, so the system can resume normal operation as soon as possible.

For the particular case of the reconfigurable processor array without fine-grain fabric, this problem has been studied in depth by Vasilikos et al. in [58]. A number of different algorithms were developed and evaluated in terms of execution time and accuracy compared to the optimal, for various array sizes. Out of these, we selected a *Greedy* algorithm for our evaluation, as its execution time is both very short and scalable (it is very mildly affected by the size of the input array) and its accuracy is very close to much slower algorithms, rarely dropping below 80% of the optimal solution, according to the efficiency metrics defined in that paper.

The algorithm is outlined in the pseudocode labelled Algorithm 6. It is based on the concept of *bottleneck column*, meaning the column that has the minimum number of fault-free SUs, among all columns of the array. The amount of fault-free SUs in the bottleneck column is also the maximum amount of working components the algorithm can produce for the particular status. The algorithms starts by finding the bottleneck column and the amount of fault-free SUs in it (lines 7-13). Subsequently, it focuses on each of these SUs and forms a working component with the following steps:

- It connects the SU of the bottleneck column with the closest available one of the column to the right. It continues this process until it reaches the last (rightmost) column of the array (lines 16-19).

- It returns to the SU of the bottleneck column and follows the same procedure to the left, until reaching the first (leftmost) column (lines 20-23).

Algorithm 6: The greedy algorithm which produces a configuration for the coarse-grain reconfigurable array

```
 1: Greedy algorithm for coarse-grain arrays
 2: Inputs:
 3: Fault map $A$. Array dimensions $R$, $C$
 4: Output:
 5: Configuration $X$
 6: //Find first (leftmost) bottleneck column
 7: $b \leftarrow R$
 8: for $j = 1$ to $C$ do
 9:     if $A[j].workingElements < b$ then
10:         $bottleneck \leftarrow j$
11:         $b \leftarrow A[j].workingElements$
12:     end if
13: end for
14: //"bottleneck" now has the index of the first bottleneck column
15: //Assign SUs to each processor
16: for $j = bottleneck$ to $(C - 1)$ do
17:     //from bottleneck towards the end
18:     $X[i][j] \leftarrow closestWorkingElement(A[j + 1])$
19: end for
20: for $j = bottleneck$ downto 2 do
21:     //from bottleneck towards the beginning
22:     $X[i][j] \leftarrow closestWorkingElement(A[j - 1])$
23: end for
24: if X is a valid configuration then
25:     return $X$
26: else
27:     return failure
28: end if
```

As [58] does not account for the existence of fine-grain reconfigurable fabric, we have to modify the greedy algorithm for this case. The result is shown in Algorithm 7. The modifications are summarized below:

- In the beginning, it is decided whether the bottleneck column for the particular array status is unique or not (lines 8-18). Note that using the fine-grain fabric is beneficial only in the case of unique bottleneck column, as it can increase the number of available SUs of this column by one. This would not help in case of multiple bottleneck columns - adding an available SU to one of them would not modify the minimum number of available SUs in any one column.
- In case the bottleneck column is unique, the fine-grain fabric is added as an extra available SU to it, with the row index 1, as the fine-grain block is considered to be row 1 of the array (lines 22-27). The algorithm then proceeds as in its original version. It starts the component formation by the SU which is instantiated on the fine-grain fabric, since it has the lowest row index from all available SUs of the bottleneck column.
- In case the bottleneck column is not unique, the fine-grain fabric is ignored and the algorithm continues as in the case of coarse-grain reconfigurability only.

Using the two versions of the greedy algorithm presented in this section, we can obtain, for every possible status of the array, its configuration. This means we know, for each status:

- The number of salvaged components.
- Whether or not there is a component that makes use of the fine-grain fabric and which SU is instantiated on it.
- For the rest of the components, the number of bubble stages of each.

This is all the information we need to characterize the whole array for each possible status, using the individual component characterization results of Section 2.5.2 and the fault map obtained as explained in Section 2.5.3. In the next section we explain how this last step is done.

2.5.5 Characterizing the whole array

Up to this point, we have presented our methodology for obtaining the following results:

- Characterization of each individual component, in terms of performance (in IPS) and power dissipation (in Watt) based on number of bubble stages and possible use of the fine-grain fabric (see Section 2.5.2).
- Enumeration of all possible array statuses and annotation of each status with a probability of occurrence, based on the fault density (see Section 2.5.3).
- The number and type of components that can be formed by a fast algorithm for each of the possible array statuses, both with or without fine-grain reconfigurable fabric in the array (see Section 2.5.4).

Algorithm 7: The greedy algorithm modified to also make use of the fine-grain block

```
1: Greedy algorithm for mixed-grain arrays
2: Inputs:
3: Fault map $A$. Array dimensions $R + 1, C$
4: (row 1 is the fine-grain block, rows 2 to $R + 1$ correspond to the components)
5: Output:
6: Configuration $X$
7: //Find first (leftmost) bottleneck column and check if it is unique
8: $b \leftarrow R$
9: $u \leftarrow 0$
10: for $j = 1$ to $C$ do
11:     if $A[j].workingElements < b$ then
12:         $bottleneck \leftarrow j$
13:         $b \leftarrow A[j].workingElements$
14:         $u \leftarrow 1$
15:     else if $A[j].workingElements = b$ then
16:         $u \leftarrow 0$
17:     end if
18: end for
19: //“bottleneck” now has the index of the first bottleneck column
20: //$u$ is 1 if bottleneck column is unique
21: //Place the fine-grain block in the bottleneck column
22: if $u = 1$ then
23:     for $j = 1$ to $C$ do
24:         $A[1][j] \leftarrow nonworkingElement$
25:     end for
26:     $A[1][bottleneck] \leftarrow workingElement$
27: end if
28: //Assign SUs to each processor
29: for $j = bottleneck$ to $C - 1$ do
30:     //from bottleneck towards the end
31:     $X[i][j] \leftarrow closestWorkingElement(A[j + 1])$
32: end for
33: for $j = bottleneck$ downto 2 do
34:     //from bottleneck towards the beginning
35:     $X[i][j] \leftarrow closestWorkingElement(A[j - 1])$
36: end for
37: if X is a valid configuration then
38:     return $X$
39: else
40:     return failure
41: end if
```

Based on the above, we can, for each array status and fault density, trivially calculate the performance (in IPS) and power dissipation (in Watt) of the whole array, by means of simple summation of the respective figures for the individual components. Subsequently, using the probabilities of occurrence of each status as weights, we can calculate the *expected* values (E) for the overall performance ($E[Perf]$) and power ($E[Power]$) of the array for a particular fault density, as means of weighted average of the respective figures for each status. Lastly, the expected energy efficiency ($E[Energy\_eff]$) can be trivially calculated as the product of performance divided by power:

$$Perf_{status} = \sum_{componenets} Perf_{component}$$

$$E[Perf] = \sum_{statuses} P_{status} * Perf_{status}$$

$$Power_{status} = \sum_{components} Power_{component}$$

$$E[Power] = \sum_{statuses} P_{status} * Power_{status}$$

$$E[Energy\_eff] = E[Perf]/E[Power]$$

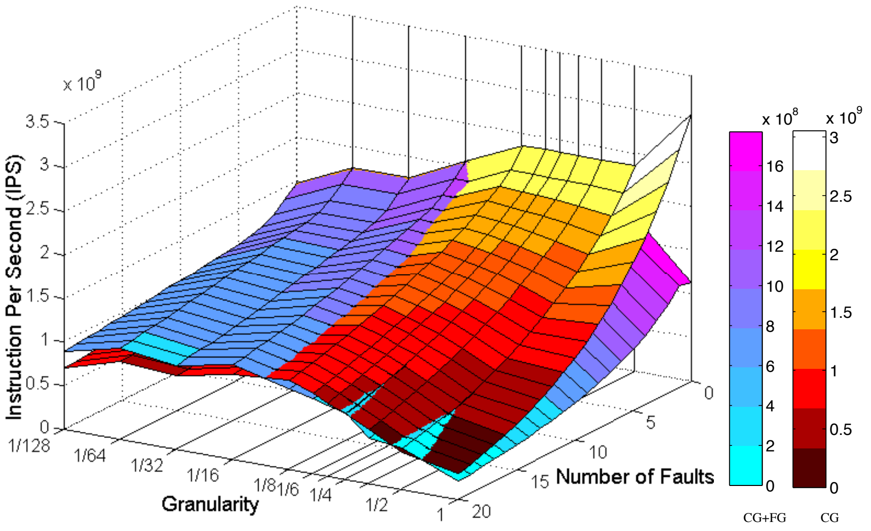
These results are used to perform design space exploration as explained in the next section.

2.5.6 Results

Throughout sections 2.5.2 to 2.5.5 we have established a methodology to obtain the expected performance and energy efficiency of a reconfigurable processor array (with or without fine-grain fabric) as a function of the defect density. In this section, we use this result to perform design space exploration, aiming to determine the ideal granularity of reconfiguration for each fault density.

Our design space is thus defined by the following parameters:

- The fault density, expressed as the expected number of faults on the whole array. As mentioned in Section 2.5.3, a fault density implies an *expected* value for the total number of faults on the array. This value is the “# of faults” shown on the Y axis of the graphs of Figure 2.18.
- The granularity of the SU, expressed as a fraction of the whole component. Thus, when dividing the component into C SUs, the granularity is $1/C$. Granularity occupies the X axis in the graphs of Figure 2.18. Note that granularity of 1 means that the components making up the array are not reconfigurable and subsequently fault intolerant. Thus, it does not even suffer the flat penalties for reconfiguration that the components of the reconfigurable array do.



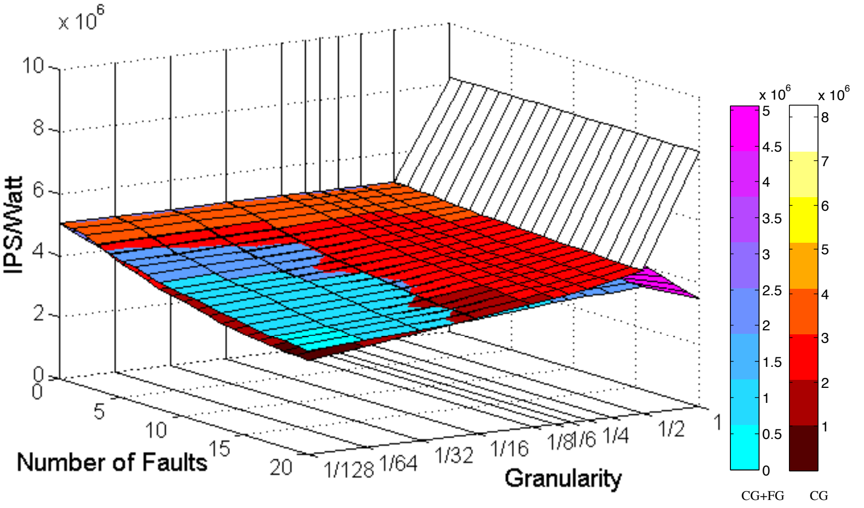


Figure 2.18: Design space exploration of coarse-grain and mixed-grain reconfigurable multiprocessor arrays of various granularities: performance (top) and energy-efficiency (bottom) for different fault densities. All designs use area smaller or equal to 9 baseline cores. Fault density is measured as the expected number of permanent faults on the area of 9 baseline cores. Granularity is the fraction of a component that constitutes a SU. Warm/cool colored planes correspond to coarse-/mixed-grain reconfigurable arrays.

- The existence (or not) of fine-grain reconfigurable fabric. As previously explained, this choice allows trading some extra flexibility with fault-free case efficiency. On the graphs of Figure 2.18, the warm/cool colored planes correspond to arrays without/with fine-grain fabric.

For all points of the design space (except for the ones with granularity 1), we take into account the flat area overheads incurred by (coarse- and, whenever applicable, fine-grain) reconfigurability. According to this, we calculate the number of components that fit in a given area. Indicatively, in the area of nine baseline, non-reconfigurable processors, we can fit eight reconfigurable ones of granularity $1/2$ *without* fine-grain fabric and six of the same granularity *with* fine-grain fabric. For each point of the design space, we illustrate the following:

- The expected performance of the array in terms of IPS (top plot of Figure 2.18).
- The expected energy efficiency of the array in terms of IPS/Watt (bottom plot of Figure 2.18).

Performance degradation when increasing the fault density is affected by two factors:

1. the number of components able to be formed
2. the fact that reconfigured components are slower than fault-free ones

The design space exploration presented above, produces the following conclusions for different ranges of the fault density:
Low fault densities (0-4 faults): As long as the array is fault free, the non-reconfigurable array with granularity CG(1) is obviously better, being 35% faster and 1.7 times more energy efficient. This is also true for low fault densities of up to four faults. As there are not many faults to isolate, the overheads incurred by reconfigurability stand out in this case, not providing sufficient benefits to outweigh the penalties.
Medium fault densities (5-13 faults): In this range, the non-reconfigurable array starts losing several components, making it beneficial to isolate faults with coarse-grain reconfiguration. Indeed, the coarse-grain reconfigurable array with eight SUs per component, CG($\frac{1}{8}$) has much more stable performance as number of faults increases from 5 to 13, while CG(1) has a rapid performance degradation. Thus, CG($\frac{1}{8}$) catches up and is better in this range. Additionally, CG($\frac{1}{8}$) offers better performance than any mixed-grain alternative and 3% better energy efficiency than its mixed-grain equivalent, CG+FG($\frac{1}{8}$). The best mixed-grain alternative for this range is CG+FG($\frac{1}{16}$), but it still provides at least 10% lower performance and 3% lower energy efficiency than CG($\frac{1}{8}$).
High fault densities (13 or more faults): In this range of the fault density several SUs of the array are faulty. A finer granularity means that every faulty SU corresponds to a smaller loss of hardware resources. Thus, in this range, more SUs per component, coupled with fine-grain fabric, yield better performance results. We observe that the mixed-grain reconfigurable array with 16 SUs per component, CG+FG($\frac{1}{16}$) provides the

highest and most stable performance, being around 8% faster than CG($\frac{1}{8}$). However, it remains marginally less energy efficient than CG($\frac{1}{8}$).

In general, coarser granularities offer better performance for low fault densities, while finer granularities provide better and more stable performance when the number of faults increases. When it comes to energy efficiency, it has to be noted that for every fault density, the best coarse-grain array is more energy efficient than the best mixed-grain one, for a margin of up to 12%. Lastly, despite the rapid performance loss, the non-reconfigurable array CG(0) remains roughly 1.7 times more energy efficient for the whole spectrum of the fault density.

2.6 Chapter Summary

In this chapter, we present our techniques for utilizing, at runtime, the adaptability that is present in modern multiprocessor systems, to minimize system efficiency degradation each time a *component failure* event happens. The aforementioned adaptability comes in the form of hardware reconfigurability as well as flexibility of the workload, that can consist of tasks with different criticality levels, or alternative (e.g., lightweight) implementations of each task.

We provide multiple implementations of a runtime manager for graceful degradation of such adaptive multiprocessors: custom heuristics based on incremental and precomputed solutions, as well as tailored implementations of known optimization algorithms (simulated annealing and genetic algorithm). Our custom heuristics are found to be at least one order of magnitude faster than SA and GA, and produce solutions of comparable quality: making use of their ability to find an incremental solution, they provide better efficiency for at least the first 20% of the expected total system lifetime, while they are at most 13% worse towards the end of the lifetime, often staying on par with SA and GA.

In some contexts, custom heuristics have difficulty satisfying reconfiguration cost constraints. In these cases, the average efficiency they achieve is up to 80% worse than SA and GA, because non-satisfaction of a constraint leads to system failure and a system efficiency value of 0. However, in these cases, the cheap and fast custom heuristic can be backed up by a slower but more consistent algorithm, such as SA and GA, to achieve both low average response time and constraint satisfaction.

In all studied fault scenarios, our runtime manager was less than 15% worse in terms of overall system efficiency compared to an optimal solver, and was even closer than this for the most part of the system's lifetime. Furthermore, the response time of all algorithms scales in a slower than linear manner as the problem size increases, making them suitable for systems of arbitrary size.

As the runtime manager assumes that hardware components are annotated with the performance and energy efficiency degradation that comes with reconfiguration, we have also presented a methodology to obtain these degradation values. The characterization methodology focuses on a particular type of degradable component, processor arrays with mixed-grain reconfigurability.

By performing design space exploration we have determined the best reconfiguration granularity for each fault density. We have found that for up to four faults in an area of nine baseline cores, non-reconfigurable arrays (CG(1)) maintain their advantage, as the overheads incurred by flexibility are higher than the associated benefits. However, when the number of faults on the same area increases above four, the performance of a non-reconfigurable array degrades rapidly, and a coarse-grain array with reconfiguration granularity of eight substitutable units per component (CG($\frac{1}{8}$)) becomes faster than CG(1) by around 10%. When the fault density further increases to 13 or more faults on the same area, the best design is even more flexible: a mixed-grain reconfigurable array with a granularity of 16 substitutable units per component (CG+FG($\frac{1}{16}$)) is 8% faster than and as energy efficient as CG($\frac{1}{8}$).

3 Runtime Management for Energy Efficiency

In the previous chapter we focused on heterogeneous, fault-prone, reconfigurable multiprocessors running flexible workloads. The main objective was to minimize the negative effects of permanent faults. In this chapter, we will shift the optimization objective to energy efficiency. We still target heterogeneous systems, albeit limiting the focus to *single-ISA* heterogeneity with *Dynamic Voltage and Frequency Scaling* (DVFS) capability. The main challenge we choose to tackle is the unpredictably dynamic nature of workloads. While in Chapter 2 the workload running on the system was flexible but static (thus, its flexibility was a tool we could use for optimizing efficiency), in this chapter we consider workloads consisting of multiple applications running concurrently, competing for shared resources and having unpredictable spawn and termination times. In this context, our runtime manager has to react to *application spawn* and *application termination* events.

Heterogeneity and DVFS capability of modern multiprocessors offer flexibility, which in turn provides knobs for trading performance for energy and vice versa, thereby improving system efficiency. Commercial examples of such, single-ISA, architectures include the Samsung Exynos [83] and Qualcomm Snapdragon [84] processors. They employ multiple types of cores (power-efficient and high performance), and multiple voltage-frequency pairs per cluster of cores. In this manner, they can potentially adapt to the fluctuating system workload and its needs at runtime. Such runtime variations stem from the number, type and performance requirements of the applications running at any given time on the system. Adapting the system to these dynamic changes can balance smooth application operation and performance with energy efficiency.

In this context, choosing the optimal system configuration is often not straightforward. Executing an application on a power-efficient core might prolong execution enough to reduce energy efficiency, while choosing a high-performance core poses the opposite risk - an increase of power consumption outweighing performance gains. DVFS adds more complexity to this problem, as it is unclear how a high-performance core at a low frequency and a power-efficient core at a high frequency compare in terms of energy efficiency [85]. Interference between applications sharing system resources, i.e., levels of the memory hierarchy, makes the runtime decision even harder. In conclusion, today's multiprocessors offer great potential for performance and energy efficiency, which is not fully utilized due to the difficulty of choosing optimal configurations [26–28, 30].

In the past, the challenge of the system load being dynamic and unpredictable has been answered in the following two ways. Some solutions make an one-off decision whenever an event happens, such as an application spawning or termination. They predict the impact of such events using offline application characterization coupled with existing system monitoring information. Subsequently, they choose a new system configuration aiming to maximize energy efficiency, requiring little to no corrective action until the next event. However, predicting the performance of different applications running concurrently and competing for shared resources, is extremely complex. This leads to other solutions that avoid this challenge by following a trial and error approach. They begin with a naïve, even random, placement of applications and subsequently use online monitoring to gradually converge to a more energy-efficient system configuration, i.e., choice of cluster frequencies and/or application placements [33, 37]. Such approaches may require a long time to converge to a satisfactory configuration and may suffer from poor scaling of response time when system size increases.

In the work presented in this chapter, elements of both above approaches are combined. In response to an event (application spawning or termination), the space of possible system configurations is partially searched, predicting the energy efficiency of different options and selecting the best one. Subsequently, this decision can be revisited when more accurate system monitoring information becomes available. However, our main focus is on the initial prediction and decision step to minimize any need for subsequent adjustments. Gradually adapting the system can be a slow and costly process, especially when the initially selected configuration is inefficient. Thus, it is important that the first response to an event is an efficient one.

The two main challenges of this approach are (i) creating a model which accurately predicts performance and power for arbitrary candidate system configurations and (ii) developing a fast and scalable decision algorithm, since it is not practical to evaluate all possible configurations at runtime. In an attempt to face these challenges, this work makes the following novel contributions:

- A model capable of estimating all running applications' performance and power, for every possible decision to be taken whenever an event (application spawning or termination) happens. The model uses a combination of (i) online performance monitoring, (ii) online projection of offline-acquired application performance characteristics and (iii) static platform characterization parameters, to predict the result of candidate decisions.

- Three runtime heuristics that use the above model to choose a new system configuration whenever an event happens. They select a small number of candidate solutions to evaluate with the prediction model and make a decision that respects all applications' performance requirements while maximizing energy efficiency of the system in terms of IPS/Watt.

The rest of the chapter is organized as follows: Section 3.1 summarizes related work, positioning our novel contributions among existing literature. Section 3.2 describes our prediction model, used at runtime to estimate performance and power of arbitrary configurations. Section 3.3 presents our decision algorithms and analyzes them in terms of complexity. Section 3.4 contains the experimental evaluation of our approach. Section 3.5 discusses some limitations and alternatives. Lastly, Section 3.6 summarizes the chapter and states the main conclusions.

3.1 Related Work

There exists a substantial amount of research on energy-efficiency in heterogeneous multiprocessor systems. We hereby attempt to identify the main features that differentiate these works and point out our novelty with respect to these features. The most representative works are categorized in Table 3.1.

An important categorization of existing works is with respect to the application scenario they focus on. Some focus almost exclusively on single application scenarios [31, 33, 34], while others consider concurrent execution of multiple applications [37, 86]. A representative example of the latter group is SPARTA by Donyanavard et al. [37]. SPARTA proposes a task allocation strategy handling various mixes of single- or multi-threaded applications, treating threads of the same application as separate tasks. As was mentioned in the chapter's introductory part, we specifically focus on the challenges posed by this multiple application scenario. Nevertheless, tackling multiple, independent applications is interesting, as it is closer to real-life use cases such as mobile devices. Thus, we aim to maximize overall system IPS/Watt for concurrent execution. While we don't use multi-threaded versions, we allow multiple instances of the same application to exist simultaneously. To further describe the targeted scenario, alongside the number and type of applications, another important facet is the existence of requirements for individual applications. On the currently trending cluster-based architectures, a single application that needs to maintain a minimum level of performance can impact the energy efficiency of a whole cluster, which is a significant part of the whole system. Thus, it is important that the runtime manager does not raise the frequency higher than needed to fulfill the performance requirement, as this typically results to lower energy efficiency. Similarly to [29, 33, 37], we consider this aspect of runtime management important, thus we equip our algorithms with the ability to satisfy individual applications' requirements, something that is not present in works such as [31, 32, 36].

Another important component is the combination of techniques utilized to extract energy savings. The three main alternatives are the following:

- Dynamic Voltage and Frequency Scaling (DVFS) [87], that focuses on selecting cluster frequencies,
- Dynamic Power Management (DPM) [88], that allows for components that are not currently used to be switched off and
- application placement (allocation) [89], that involves the decision of which type of core every application will execute on.

Existing works utilize various mixes of the three techniques. Works with more focus on single-application scenarios focus on DPM, seeking the optimal configuration of active and inactive cores [31]. On top of this, some strategies decide the application placement and predict the decision of the frequency governor without taking control of the DVFS aspect [37, 38], while others also control voltage and frequency, but still evaluate their approach for single applications [33–35]. In this work, we fully control DVFS and application placements, while at the same time allowing the DPM governor to turn off idle cores. Thus, our approach covers all three techniques of energy saving, as the DPM decision is implicit given a specific placement of applications: cores that don't currently execute an application can be set to an idle state.

The central component of every strategy, is the set of information considered for making decisions. In the context of energy efficiency runtime management for heterogeneous multiprocessors, we identify three possible sources of information:

1. offline application profiling;
2. online, runtime performance monitoring;
3. predictions performed with a suitable model, each explained separately bellow.

1. Offline application profiling: Many approaches utilize application profiling data obtained offline. For instance, DyPO by Gupta et al. [31], uses extensive offline application profiling to create a library of possible system statuses with respect to important metrics such as instructions retired, memory accesses and cache misses. For each of these statuses, the optimal configuration is identified and stored. At runtime, the system manager measures the same features online, in order to identify the library entry that is the closest to the current status and apply the configuration that offline profiling dictates to be the best. We agree that offline-acquired performance characteristics can guide a decision at runtime and consider reasonable to use profiling to extract this information. However, we identify and solve a scalability concern of such strategies: the offline nature of profiling creates a misconception of unlimited available time, which is not realistic. As an example, for a Samsung Exynos board with only four ARM A7 and four ARM A15 cores, there are 4004 possible configurations. DyPO [31] tackles this problem by selecting a representative subset of these configurations, but unfortunately does not propose a scalable extension of this strategy to larger systems. Considering that core count tends to increase and that an unpredictably high number of applications can run on a system during its lifetime, we go one step further and propose a scalable strategy for choosing configurations for offline application characterization. Our strategy requires a total amount of runs that is linear with respect to the number of cores. To further ensure the scalability of this offline stage, we limit ourselves to standalone profiling of each application, regardless of the various mixes that can appear at runtime. As will be explained, the information acquired through profiling is adapted at runtime using our online projection technique.

2. Online, runtime performance monitoring: Regardless of whether or not offline profiling is used, most approaches rely on online performance monitoring, especially works such as [37], that target dynamic multi-application scenarios, as we also do. Online monitoring is essential for capturing varying performance characteristics of individual applications. Such performance variations are caused by the dynamic nature of the system workload: spawns and terminations of applications affect the contention for shared resources, which in turn affects other applications' performance. Taking compact, static, application-specific offline profiling information (see previous point) and projecting it to the dynamic, application mix-dependent status of the system, as it is measured by online performance monitoring, is one of the core contributions of our work.

3. Predictions performed with a suitable model: Although most strategies involve some kind of modelling of performance and power [90], there is an important distinction in whether or not this model is used to perform explicit predictions, which in turn guide the decisions. For instance, NORNIR proposed by De Sensi et al. [33] is a representative example of a strategy utilizing explicit predictions. In NORNIR, a newly spawned application, of which no prior knowledge is assumed, is first tried on a sequence of different configurations in order to refine a prediction model during a calibration phase. The refined model is subsequently used to choose a close-to-optimal configuration for the steady phase. During this final configuration selection step, explicit routines predict application performance and system power for various candidate configurations, and the configuration with the highest predicted efficiency is selected. On the contrary, in DyPO [31] the runtime decision for a system configuration is driven by identifying an offline-profiled system status that is similar to the current runtime status. A configuration that was efficient for the profiled status, is expected to also be efficient in the current similar runtime situation, without explicit prediction of particular metrics for performance or power. Furthermore, if we scrutinize approaches that use explicit prediction, we can identify qualitative differences among their prediction models. The prediction model of [37] is based on the lightweight but coarser-grain approach of simply categorizing (binning) applications with respect to a number of compute- and memory-boundedness criteria. In [32], the prediction model is workload-specific, requiring retraining every time a new application mix appears. In NORNIR [33], as explained above, online calibration (refinement) is performed instead of retraining, making it a more realistic solution for dynamic, unpredictable scenarios. Unfortunately, the calibration phase has to attempt an unbounded number of non-optimal configurations, thus it is still difficult to predict its effect on the end-to-end efficiency. We achieve performance prediction by projecting offline profiling data to the online-monitored current system status and power prediction through characterization of the actual board, that produces a static, platform-specific power model involving no application-specific parameters. As such, our models perform fine-grain performance and power prediction, without the need for retraining or runtime refinement.

Lastly, existing works differ with respect to the strategy used to make runtime decisions. Liu et al. [36] established that low-complexity heuristics can produce solutions very close to the optimal, in time two to three orders of magnitude lower than optimal solvers. However, their heuristic's execution time scales quadratically with the number of cores in the system, n. The allocator proposed in SPARTA [37] also includes an iterative step, resulting in an execution time that explodes when going from a 32-core system to a 64- and 128-core system. On the other hand, Aaslaud et al. propose a linear-complexity algorithm in [32]. Unfortunately, the fact that this heuristic is based on a model that needs to be retrained for each application mix, is a major drawback for a runtime optimization strategy, as it reduces its applicability when it comes to unpredictable, dynamically changing scenarios. It is clear, however, that the complexity of the proposed heuristics has to be carefully chosen, to trade well between solution quality and execution time. We propose different heuristics with complexity of at most $\mathcal{O}(n * log\ n)$.

Table 3.1 summarizes the most representative existing works with respect to the aspects described above. The first column, "Mult. App.", indicates whether or not the proposed approach is evaluated for dynamic, unpredictable scenarios of multiple concurrent applications, while the next column, "Perf. Req." states whether or not it considers performance requirements of individual applications. The next three columns ("Energy-saving Techniques") indicate which of the identified energy-saving knobs are used. A parenthesized tick mark means that the technique is utilized but not explicitly controlled. The next three columns ("Decision guided by") indicate what kind of information is used to guide runtime decisions. The last column, "Decision Strategy" states if the approach uses heuristics or some other method to make decisions at runtime, and the complexity of the heuristic where applicable.

In summary, the energy efficiency optimization approach we propose in this chapter:

- Finds a sweet spot for utilizing both offline profiling and online monitoring information, keeping profiling scalable and confined to single applications, while using monitoring and online projection to incorporate the dynamic, unpredictable characteristics of the current application mix;

- Decouples performance and power prediction, performing the former as described in the previous point, and the latter with a model trained only once for a specific platform, since it involves only architecture-specific parameters;

- Targets a realistic scenario, as it considers concurrent execution of arbitrary application mixes and takes care of individual applications' performance requirements.

The following two sections present the two main components of the proposed strategy: Section 3.2 describes the method used to predict performance and power of different candidate configurations, while Section 3.3 outlines our runtime algorithms that perform

Table 3.1: Categorization of existing works with respect to various aspects.

	Mult. App.	**Perf. Req.**	**Energy-saving Techniques**			**Decision guided by**			**Decision Strategy**
			DVFS	DPM	Place-ment	Profiling	Monitoring	Prediction	
Liu et al [36]	✓				✓		✓	✓	Heuristic ($\mathcal{O}(n^2)$)
NORNIR [33]		✓	✓		✓		✓	✓	Runtime model refinement
SPARTA [37]	✓	✓	(✓)		✓		✓	✓	Heuristic ($\mathcal{O}(n^2)$)
Aalsaud et al [32]			✓	✓		✓		✓	Heuristic ($\mathcal{O}(n)$)
DyPO [31]			✓	✓		✓	✓		Matching profiled status
Our approach	✓	✓	✓	(✓)	✓	✓	✓	✓	Heuristics ($\mathcal{O}(n * logn)$)

partial search of the configuration space to make decisions whenever an *application spawn or termination* event happens.

3.2 Performance and Power Prediction

This section provides a detailed description of our model, used to predict the performance and power impact of runtime decisions before applying them. Application performance is predicted by projecting profiling information to the current system status. In turn, the predicted performance is used as input to our power model.

Our approach revolves around different performance metrics, namely retired instructions and accesses of all levels of the memory hierarchy, per unit of time. Thus, on a system with two levels of cache, such as the ODROID XU3 [91] (see left part of Figure 3.1, labelled "Hardware"), we are interested in the following metrics:

- Retired instructions per second (IPS)
- L1 data cache access per second (L1aps)
- LLC (L2) access per second (LLCaps)
- DRAM access per second (DRAMaps)

Predicting future values of these performance metrics is complicated by the dynamically changing system status. We demonstrate this with a very simple motivating example. We run the SP and CG benchmarks of the NAS suite [92], on a cluster of big (A15) cores of an ODROID XU3 board, in two different setups at the same frequency: first, standalone (one by one, on a core of the cluster) and then in parallel (on different cores of the same cluster, but simultaneously). We observe that the performance of each degrades by almost 20% when they execute in parallel, because of the competition for shared resources (L2 cache and DRAM). In literature, this challenge is tackled by using prediction models that involve parameters specific to each application mix [32]. This, in turn, creates the need for retraining or refining the model, every time the application mix changes.

To answer the above drawback, we introduce the technique of online projection of profiling information. The core idea of projection is that performance characteristics acquired by profiling a standalone application offline, can be combined with online performance monitoring at runtime, to extrapolate the application's behavior when it runs concurrently with other applications. The offline profiling determines the individual characteristics of each single application, while the online monitoring of the system's current status incorporates the effect of competition for shared resources, eliminating the need for model retraining. More specifically, when profiling each application, we force it to compete with other instances of itself for shared system resources, such as the LLC and the DRAM. We vary the amount of competition and record how the application's performance varies as a result. At runtime, we use performance counters to obtain

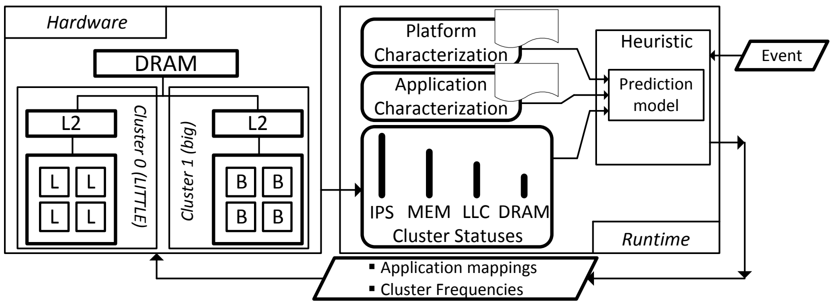


Figure 3.1: The ODROID XU3 hardware and our runtime.

awareness of the current level of competition for shared resources in the system. When the profiled application spawns, we match the online measurement to the closest profiled value and use it as a starting point to predict its performance in the current situation.

The above process is illustrated in Figure 3.2. It involves profiling an application in several different configurations - how these are determined will be explained shortly. For now, to keep this example simple, let us suppose that we only use two configurations: 2 and 4 instances of the application running in parallel, evenly divided between the big (B) and little (L) cluster of an ODROID XU3, resulting to the *initial* states (1B, 1L) and (2B, 2L), listed and marked with circles on the "DRAMaps initial" axis. We measure total DRAM accesses per second (DRAMaps) for each of these scenarios. Then, for each scenario, we spawn one extra instance on the big cluster and record the change in the DRAMaps metric. We do the same for one extra instance on the little cluster. These *final* states are marked with squares on the "DRAMaps final" axis, at the ends of the "+1B" and "+1L" arrows. The upward-pointing arrows at the right side of the picture represent the DRAMaps increment for each final state. At runtime, we monitor the total amount of system DRAMaps online - marked with a horizontal dashed line labelled "DRAMaps current". When the specific application is spawned, we find the initial state closest to the current DRAMaps value, in this example (2B, 2L) ①. This is called *resembling state*, since it is the one closest to the current system state with respect to the particular performance metric. Then, if we want to place the application on the big cluster, we follow the "+1B" line and retrieve the DRAMaps value of this final state ②, which we call *projected state* since it represents the impact of the spawned application on the system. To quantify the projection, we use the difference of DRAMaps between projected and resembling state, which in this example is represented by the solid upwards-pointing arrow ③. Adding this difference to "DRAMaps current", gives us the predicted value of DRAMaps ("DRAMaps predicted") ④, if we proceed with the placement of the application on a big core.

Aside the competition for shared resources, application performance is also affected by the operating frequency. To avoid profiling for every supported frequency, the frequency effect on performance is considered by estimating a *scaling factor* for each

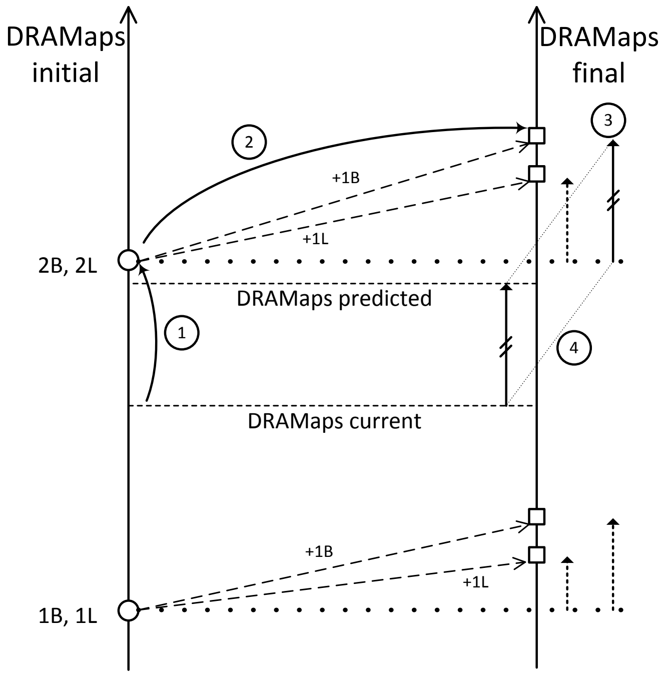


Figure 3.2: Online projection for performance prediction. DRAM accesses per second (DRAMaps) of an application have been profiled for various (B, L) core counts and for their corresponding (+1B) and (+1L) configurations. At runtime, the current value of total system DRAMaps is found to be closest to the (2B, 2L) profiled value ①. Assuming we want to predict DRAMaps for placing the application on a big core, we retrieve the profiled value for the (+1B) configuration ②. The difference between the two profiled values ③ is added to the current DRAMaps, to get the predicted DRAMaps after application placement ④.

application. This factor depends on the compute- versus memory intensiveness of the application and will be further explained in Section 3.2.1.2. Consequently, profiling information is used to characterize each application in terms of compute- vs memory-boundedness. At runtime, we use this information to predict the impact of frequency on application performance. Our profiling strategy will be elaborated in Section 3.2.1 and the resulting prediction routines will be described in Section 3.2.2.

On top of performance, we also need to be able to predict the impact of our decisions on system power. For this purpose, we have calibrated a linear regression model which estimates the power costs of the predicted performance metrics. This model only involves platform-specific parameters which only have to be determined once for a given system. Subsequently, the model can be used for any application mix. The power model is described in Section 3.2.3.

3.2.1 Application Profiling

As explained above, to guide the prediction of performance metrics under changing circumstances, we perform offline application profiling. Since this has to be done in advance, we cannot assume knowledge of the application mixes that will appear at runtime, thus we restrict ourselves to standalone profiling of each one application, contrary to works such as [32], which propose model training for each mix separately. Furthermore, we note that the number of applications that can appear on a system during its lifetime is unpredictable. Thus, it is important that offline profiling for each one remains scalable with respect to system size, both in terms of time needed to complete and in terms of memory required to store the results that are used at runtime.

3.2.1.1 Performance Profiling at a Set Frequency

To find out how an application performs under different levels of system workload (knowledge that, as explained, guides online projection), we run multiple instances of it concurrently, varying the number of these instances from one up to the total number of cores in the system minus one. For each number of instances, we use three different policies for placing them to cores:

1. use the biggest available core;
2. use the smallest available core;
3. use the type that currently has more available cores (resolving ties in favor of the biggest).

For instance, when we run six instances on a system with four big and four little cores, they are placed in the following ways by each of the three policies:

1. four big and two little cores;
2. two big and four little cores.
3. three big and three little cores.

The outcome of these runs is the set of *initial states*, i.e., the circles on the left side axis of Figure 3.2 for each of the performance metrics used in our prediction.

Additionally, for each of the initial states, we perform one more run for each type of core in the system, with one extra instance spawning on such a core, if possible. For example, after running the initial state of three big and three little cores, we also profile the following two states:

1. three big and four little cores (+1L);
2. four big and three little cores (+1B).

These runs correspond to *final states*, i.e. the squares on the right side axis of Figure 3.2. For each initial and final state we store the values of the four performance metrics.

The above application profiling process is performed at a single, intermediate frequency (f_c) of each cluster. We define f_c to be 1.4GHz for the big cluster and 1.0GHz for the little cluster. However, as described earlier in this section, the performance of each application is also affected differently by frequency. It would be possible to repeat the process in every frequency, but this would multiply both the time needed for profiling and the size of the resulting performance characterization file by a factor equal to the number of available frequency levels. Thus, we deal with the impact of frequency in an orthogonal manner, explained in the following subsection.

3.2.1.2 Frequency Scaling

The effect of frequency on performance depends on the memory- or compute-boundedness of the application. For an entirely compute-bound application, Instructions per Cycle (IPC) are the same at every frequency, resulting to IPS scaling linearly. In simpler terms, an entirely compute-bound routine that takes 1.0 seconds to complete at 500MHz, would take 0.5 seconds at 1.0GHz. For a more memory-bound application, the effect of DVFS is diminished, meaning that IPC is lower at higher frequencies and the effect on IPS is sub-linear. A memory-bound routine that takes 1.0 seconds to complete at 500MHz, would take less than 1.0 but more than 0.5 seconds at 1.0GHz, as doubling the frequency will only speed up computations, not memory accesses.

To trace the effect of frequency on each application, we run a single instance of it on each core type, at every available frequency level and compute a *scaling factor (SF)*, which represents the average IPC degradation for a frequency increase of 100MHz. Thus, in the general case, $SF \leq 1.0$ and for an entirely compute-bound application, $SF = 1.0$. We obtain one scaling factor for each core type. At runtime, the scaling of IPC is converted to IPS. More specifically, the estimated performance IPS_{new} of an application when changing the frequency from f_1 to f_2 is:

$$IPS_{new} = IPS_{old} * (f_2/f_1) * SF^{\alpha} \tag{3.1}$$

where f_2/f_1 is the speedup (or slowdown) of computational parts due to the frequency change and SF^{a} is the part of this speedup (or slowdown) that is mitigated by memory accesses, with

$$\alpha = (f_2 - f_1)/100MHz$$

being the number of 100MHz steps required to go from f_1 to f_2. Furthermore, for standalone execution of an application, the rest of the performance metrics (L1aps, LLCaps, DRAMaps) scale in the same manner as IPS, as the data access pattern remains the same across different frequencies, thus the amount of instructions and memory accesses is the same.

In summary, to profile one application on a generic board with n cores, l core types and r frequency levels for each core type, we need the following runs:

- $n * 3 * l$ runs for profiling at a set frequency, using 3 policies to create the initial states and l extra runs for the final states, all of the above for each of the n core counts.
- $l * r$ runs for frequency scaling profiling, as we run the application once for each of the r available frequencies of each of the l core types.

The above defines a maximum of $n * 3 * l + l * r$ runs. While the number of system cores n can be expected to increase in the future, the frequency levels r and the different core types l are not expected to increase as radically. Thus, the asymptotic complexity of our application profiling would be $\mathcal{O}(n)$, considering constant l and r. This is a loose upper bound, as many of the configurations will occur more than once, but only have to be profiled once. In other words there is overlap between the three placement policies and between initial and final states. In total, on our available board of $n = 8$ cores, $l = 2$ core types and using $r = 11$ different frequency levels on each core type, the loose upper bound is

$$8 * 3 * 2 + 2 * 11 = 70$$

but because of overlap, we need to run each application in only 46 different configurations. As an indicative comparison to existing approaches, the characterization methodology proposed in [31], needs to run each application in 128 configurations on the same board we use and if the same methodology were to be applied on larger systems unchanged, the asymptotic complexity would be $\mathcal{O}((n/m)^m)$, where m is the number of clusters in the system.

In terms of memory requirements, our profiling file size for each application is computed as follows: out of the 46 profiling runs, 24 are for performance profiling at a set frequency. Each of these produces one entry for each of the four performance metrics and for each core type. Each entry being 4 bytes, this results to $24 * 4 * 2 * 4 = 768$ bytes. The other 22 runs are used to determine the two scaling factors (one for each core type), which require and extra 2*4=8 bytes, for a total of 776. For a generic system, the number of entries produced by profiling at a set frequency is proportional to the total number of runs, resulting to the same asymptotic complexity for the required memory as for the profiling time, that is $\mathcal{O}(n)$. Frequency profiling produces just l scaling factors, one for each type of core, adding a constant factor to the memory requirements, not affecting asymptotic complexity.

3.2.2 Performance Prediction via online projection

Our runtime manager makes decisions concerning the frequency level of each cluster and the placement of the current workload on available cores. Thus, it needs to accurately predict application performance as a result of the following two actions:

1. Changing the frequency of a cluster;
2. Placing a newly-spawned application on a core.

To perform these predictions, the runtime manager combines application profiling data with online performance measurements of the applications currently running, as shown in Figure 3.1. At all times, the current values of the four performance metrics are known for each application via online performance monitoring, thus also their aggregate values for each cluster and the whole system. The following explains how predictions are performed.

Changing frequency: One of the decisions that our runtime manager has to make, is to determine cluster frequencies. To do so, it has to be able to predict applications' performance at any candidate frequency. To this end, the runtime manager uses the scaling factors of all running applications and Equation 3.1, to predict how the value of each performance metric will change for each application. As explained in the previous subsection, while Equation 3.1 refers to IPS, the four performance metrics are assumed to scale identically, as the access pattern of any given application does not change.

To carry out the above process, the runtime manager needs to perform a constant amount of calculations for each application running on the examined cluster. Assuming that each core runs a maximum of one application, this results to an asymptotic complexity of $\mathcal{O}(n/m)$, where m is the number of clusters, thus n/m is the number of cores per cluster. In case future systems maintain a similar count of cores per cluster, this cost can be considered constant.

Placing new application: When a new application spawns, the runtime has to choose a cluster with an idle core for it to execute. To guide this decision, it requires the ability to predict the effect of different placement options. This prediction is performed using our projection technique. Below we outline the steps taken to predict all four performance metrics, while Figure 3.3 illustrates the process for the DRAMaps metric specifically:

1. For each of the four performance metrics, the current value is determined, consulting the latest online monitoring results ①. These values, which are being monitored at the current operating frequency f_{op}, are then scaled to the performance profiling frequency f_c, using Equation 3.1 for each application currently running ②.

2. For each of the four performance metrics, their profiled values corresponding to the starting states are fetched. These values are stored sorted ③, to find, in logarithmic time, the value closest to the current monitored value ④. This is the value of the *resembling state*, as it was defined in the opening part of Section 3.2 and Figure 3.2. As such, it is the initial profiled state that is used for the actual projection ⑤.

3. Next, the runtime looks for the appropriate profiled final state, that is, the final state that corresponds to one extra instance of the profiled application on a core of the same type as the candidate cluster. This is the *projected state* of Figure 3.2. The difference between the values of the performance metric in the projected and

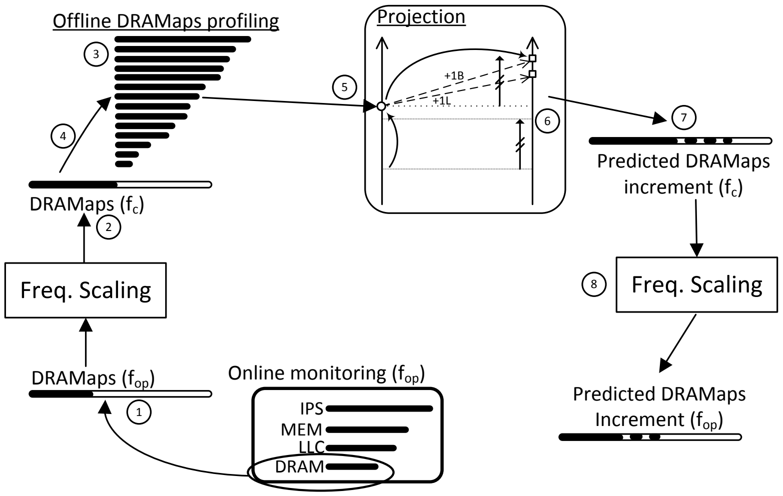


Figure 3.3: Predicting performance of a newly-spawned application using online projection. The current monitored value of DRAMaps at the operating frequency f_{op} ① is scaled to the profiling frequency f_c ②. The sorted profiled values for the new application's DRAMaps are fetched ③ and the one closest to the current is spotted ④ to be used as the "resembling state" of the projection ⑤. The resulting predicted DRAMaps increment ⑥ can be added to the system-wide aggregate ⑦. The result is scaled back to the current operating freq., f_{op} ⑧.

resembling states ⑥, is the estimation for this performance metric, if the application is placed on the candidate cluster. This is also how much the system-wide aggregate value for this performance metric is expected to increase ⑦.

4. Because this value is estimated with respect to the profiling frequency f_c, it is scaled back to the current cluster frequency, using Equation 3.1 ⑧.

The bottleneck of the projection process is finding the resembling value of each metric among the $\mathcal{O}(n)$ sorted offline profiling values. The complexity of doing so is $\mathcal{O}(log\ n)$.

Predictions involving multiple actions: Occasionally, the runtime manager has to predict the impact of a decision involving multiple incremental steps of the two types explained above (frequency scaling, application placement). To do so, it takes the following steps:

1. It performs the required prediction for the first incremental step.
2. It updates the current system status with the predicted performance values in place of the currently monitored ones.
3. It performs the required prediction for the next incremental step, based on the updated system status. The process is repeated for each step.

A good example of such a situation is the action of moving an application from one core to another. This is approached by the runtime manager as an application termination followed by a new application placement. The specific steps are as follows:

1. The system status is updated to reflect the fact that the application in question will not execute on its current core anymore. More specifically, the values of the application's performance metrics are subtracted from the cluster- and system-wide aggregates.

2. On this new system status, projection is performed as was described above, to predict the application's performance on the candidate new core.

Another example is placing a newly spawned application, at the same time increasing the frequency of the chosen cluster (e.g., to satisfy a performance requirement of the new application). In a similar manner as above, the following steps are taken:

1. Projection is performed to predict the performance metrics of the new application for placement on the candidate cluster. The system status is updated by adding the predicted performance metrics to the cluster- and system-wide aggregates, as shown in Figure 3.3.

2. On this new system status, the frequency scaling routine is applied, to predict the impact of increasing the cluster frequency on all applications, including the one that just spawned and hasn't yet started executing.

As a general rule, the runtime system always uses online monitored values, when these exist. If such values are not available, predicted values are used instead.

Being able to predict the impact of various decisions on applications' performance, allows us to use these predictions to guide power estimation. This is explained in the following section.

3.2.3 Power Prediction

Besides predicting performance, a reliable way to predict system power is needed as well. Power prediction revolves around the same performance metrics, augmented with characterization of a specific system to obtain energy costs for instructions and memory accesses. Our model considers that the total power consumption of a cluster is its idle power, incremented by the contributions of aggregate IPS and accesses to the levels of the memory hierarchy, at a particular frequency. This is expressed by Equation 3.2, that estimates the total power for one cluster, P_{cl}:

$$\begin{aligned} P_{cl} = P_{idle}(f) + \gamma * IPS * P_I(f) + L1aps * P_{L1}(f) + \\ + LLCaps * P_{LLC}(f) + DRAMaps * P_{DRAM}(f) \end{aligned} \tag{3.2}$$

In equation 3.2, $P_{idle}(f)$ is the power cost of a cluster currently not executing any application, at frequency f. IPS, L1aps, LLCaps and DRAMaps are the values for the performance metrics, either measured by online monitoring or predicted as explained in Section 3.2.2. Furthermore, $P_I(f)$, $P_{L1}(f)$, $P_{LLC}(f)$ and $P_{DRAM}(f)$ are the energy costs of one occurrence of each performance metric, i.e., one instruction executed, one L1 access etc. The energy cost of one occurrence is equal to the power cost of one occurrence per second. As the performance metrics are measured in occurrences per second, each product corresponds to the power cost incurred by one performance metric, e.g. $L1aps * P_{L1}(f)$ represents the total power cost of all L1 accesses at frequency f. Instructions are treated slightly differently, because the energy costs of different types of instructions can vary significantly, e.g. as demonstrated by Vasilakis et al. a floating point division on a big.LITTLE system has 4 to 8 times higher energy cost than an integer multiplication [85]. The γ factor in the instructions term characterizes the instruction mix of an application and covers this variability. An application instruction mix and consequently its γ factor is assumed to be constant and is determined offline for each application separately.

Characterizing a specific system in terms of power consumption consists of estimating all unknown values of Equation 3.2. To do so, we use the INA231 energy sensors of our available board [93], to measure total system power at specified intervals. Although there are finer-grain sensors on the board (i.e. power per cluster), we consider the limitation to total system power important, as more recent boards such as the ODROID XU4, often come without on-chip energy sensors, because of cost concerns. Measuring total system power is less dependent on the existence of such sensors, as it can be performed with alternative techniques, such as a sensor connected to the power outlet [94].

The easiest to determine unknown value is P_{idle}. To do so, we measure the power dissipation of the system while it executes no applications. The idle power value of each cluster can be estimated by fixing the frequencies of all other clusters to the lowest physically possible value, and varying the frequency of the examined cluster in steps of 100MHz. This is how we determine the values of P_{idle} on our available board, for each cluster and each frequency level.

To determine the values of the coefficients $P_I(f)$, $P_{L1}(f)$, $P_{LLC}(f)$ and $P_{DRAM}(f)$, we perform linear regression using measured data obtained from a mix of synthetic and NAS benchmarks [92], covering a diverse set of situations (compute- and memory-bound, integer and floating point arithmetic). As this is a step that needs to be performed only once, for a given platform, we repeat the process for each frequency of each cluster. The output of this stage is the “Platform Characterization" file in Figure 3.1, consisting of 5 floating point numbers (20 bytes) for each frequency of each cluster type.

As the training of this model only estimates platform-specific coefficients, it can be used with any application mix without re-training. Equation 3.2 can be used at runtime combined with performance prediction, to estimate the power impact of various actions (application placement, frequency scaling) or events (application termination).

The ability to predict performance and power theoretically allows our runtime manager to find the optimal configuration in terms of energy efficiency, for every occurring event. The next section describes our runtime heuristics, that make use of this model to perform *partial* search of the configuration space and produce a new configuration in acceptable time.

3.3 The Heuristics

The prediction model described in Section 3.2 allows our runtime manager to have an estimation for the impact of possible decisions on power and performance. Still, it is infeasible to evaluate every possible choice at runtime, in order to apply the best one - especially if core count of systems continues to grow as expected. This section describes the runtime heuristics we have developed to shorten this search.

The organization of our system is shown in Figure 3.1. The currently selected heuristic is triggered by an event, that is a new application spawning or a running application termination. The heuristic selects a small number of candidate solutions to evaluate, by estimating their efficiency using the prediction model. As explained in detail in Section 3.2, the model combines platform characterization and application profiling information obtained offline, with runtime performance measurements obtained from the system online, to estimate the performance and power outcome of changing a cluster's frequency and of placing a new application. Thus, all presented algorithms have available the following two routines:

- frequency_predict(c, f): Predicts the outcome of setting the frequency of cluster c to f.
- placement_predict(a, i): Predicts the outcome of placing application a on core i.

To predict the efficiency of a new configuration, the runtime manager has to define a sequence of steps of the above two types that lead from the current system configuration, to the new one.

As analyzed in Section 3.2.2, these routines have a time cost of $\mathcal{O}(n/m)$ and $\mathcal{O}(log\ n)$, respectively. In the following analysis, we consider the number of available frequency levels per cluster, r, and the number of different core types, l, as constants with respect to the total core count n.

We propose three different heuristics. They differ in the manner in which they choose candidate solutions to evaluate. A more flexible algorithm can search a larger part of the solution space and thus has a higher probability to find a more efficient solution. On the other hand, its execution time is longer, and it is possible that it scales worse than a simpler one when the system size increases.

The next three sections describe the three heuristics, followed by some notes about all of them in Section 3.3.4.

3.3.1 Heuristic 1

The first heuristic we propose is the simplest and less flexible of the three. It is described by the pseudocode of Algorithm 1. The algorithm starts by cycling through the system clusters (line 6). For each cluster, it checks whether or not there is an available core (line 8). If so, it predicts the efficiency of the system if the application is placed on this free core (line 9). The cluster that yields the best predicted efficiency and the first available core in this cluster are chosen.

Algorithm 0: Choose frequency

```
1: choose_frequency( )
2: INPUTS: Cluster index $i$, indexed list of available frequencies $f_1$ to $f_r$.
3: // $f_1$ is the highest frequency, $f_r$ the lowest.
4: OUTPUT: Selected frequency, freq_best.
5: int high = 1, low = $r$, freq_best;
6: float eff_high, eff_low, eff_best;
7: for $j = 1$ to $log(r)$ do
8:     eff_high = frequency_predict(cluster $i$, frequency $f_{high}$);
9:     eff_low = frequency_predict(cluster $i$, frequency $f_{low}$);
10:    if eff_high > eff_low then
11:        eff_best = eff_high;
12:        freq_best = $f_{high}$;
13:        low = low - (low - high)/2;
14:    else
15:        eff_best = eff_low;
16:        freq_best = $f_{low}$;
17:        high = high + (low - high)/2;
18:    end if
19: end for
20: return freq_best;
```

Algorithm 1: Heuristic 1

```
1: INPUTS: Newly-spawned application $app_{new}$.
2: OUTPUTS: A core index for placement of the application and a frequency for the core's cluster
3: //deciding core placement for the application
4: float eff_best = 0, eff_temp;
5: int placement_core = -1, placement_cluster = -1;
6: for $i = 1$ to $m$ do
7:     //$m$ is the number of clusters in the system
8:     if there is a core $j$ available in cluster $i$ then
9:         eff_temp = placement_predict(application $app_{new}$, core $j$);
10:        if eff_temp > eff_best then
11:            eff_best = eff_temp;
12:            placement_cluster = $i$;
13:            placement_core = $j$;
14:        end if
15:    end if
16: end for
17: //deciding frequency of the placement cluster
18: int freq_best;
19: freq_best = choose_frequency(placement_cluster);
20: return(placement_core, freq_best);
```

Subsequently, the system status is updated with the performance estimations and the algorithm goes on to select a frequency for the chosen cluster. To do so, it calls routine *choose_frequency()*, shown in Algorithm 0. This routine performs a logarithmic number of steps (line 7). In each step, it predicts system efficiency in two extreme frequency levels, "high" and "low" (lines 8-9). The "high" and "low" frequency boundaries are initially set to the highest and lowest available frequency respectively. After predicting their efficiency, the algorithm readjusts the value of either "high" or "low", depending on which one produced the best efficiency prediction (line 13 or 17). In this manner, it narrows downs the candidate frequency range to half the size in each step, until one frequency is chosen.

The process of choosing a cluster consists of m steps, each of which performs projection, which costs $log\ n$ time. The process of choosing a frequency consists of a constant number of calls to *frequency_predict()*, thus costs n/m time. The resulting asymptotic complexity is $\mathcal{O}(m * log\ n + n/m)$. The scaling of the number of clusters m defines which of the two terms of this sum is dominant. If the number of cores per cluster of bigger systems is the same as today's systems (e.g., four cores per cluster), m will scale linearly with n, making n/m constant and the complexity of Heuristic 1 $\mathcal{O}(m * log\ n)$. In the other extreme case, the number of clusters m will remain constant and the number of cores per cluster n/m will scale linearly with n, resulting to a complexity of $\mathcal{O}(n)$.

Note that the pseudocode describes the way Heuristic 1 deals with a new application spawning. All heuristics are also called whenever an application finishes execution. In this case, the part that determines the placement of the new application is skipped (i.e., lines 5-16 of Heuristic 1).

3.3.2 Heuristic 2

The second heuristic, described in Algorithm 2, is similar to the first, with the extra feature of estimating the impact of the newly-spawned application on the performance of other applications already executing on the chosen cluster. As applications share common resources in the system, i.e., the memory hierarchy, the performance of already running applications will be affected by the placement of a new one, thus re-estimating said performance is expected to yield more accurate results. This additional feature can be spotted in lines 9 to 11. Here, the algorithm cycles through all cores of the currently examined cluster. For each core that executes an application, it performs a projection to predict its performance in the new, more contested state and overwrites current monitoring data of these applications with the predicted values. Otherwise, just like Heuristic 1, this algorithm chooses the cluster and core with the highest predicted system efficiency and then chooses a frequency for this cluster. Just like Heuristic 1, if the calling event was an application termination, lines 4-18, corresponding to the placement decision, are skipped. Heuristic 2 performs m sets of up to n/m projections, resulting to a time cost of $\mathcal{O}(n * log\ n)$.

Algorithm 2: Heuristic 2

```
1:  INPUTS: Newly-spawned application $app_{new}$.
2:  OUTPUTS: A core index for placement of the application and a frequency for the core's cluster
3:  //deciding core placement for the application
4:  float eff_best = 0, eff_temp;
5:  int placement_core = -1, placement_cluster = -1;
6:  for $i = 1$ to $m$ do
7:      if there is a core $j$ available in cluster $i$ then
8:          eff_temp = placement_predict(application $app_{new}$, core $j$);
9:          for all cores $k$ in cluster $i$ except $j$ do
10:             eff_temp = placement_predict(application $app_k$, core $k$);
11:         end for
12:         if eff_temp > eff_best then
13:             eff_best = eff_temp;
14:             placement_cluster = $i$;
15:             placement_core = $j$;
16:         end if
17:     end if
18: end for
19: //deciding frequency of the placement cluster
20: int freq_best;
21: freq_best = choose_frequency(placement_cluster);
22: return(placement_core, freq_best);
```

3.3.3 Heuristic 3

The third heuristic, described by Algorithm 3, is the most complex of the three, as it is the only able to take corrective action, by moving already placed applications between cores. As such, it is the only heuristic that allows for revisiting and modifying previous runtime decisions. Initial decisions are made using profiled application information. Revisiting an initial decision allows the use of real, monitored performance data, in place of less accurate predicted performance, about an application.

Heuristic 3 starts by choosing a placement for the new application in the same manner as Heuristic 2 (lines 5-17). Subsequently, it chooses a currently running application to be moved to a different core. To do so, it evaluates the efficiency of each running application, based on its actual online monitored performance and its estimated power, calculated with Equation 3.2 (lines 18-28). Next, it chooses a target core for this application on a different cluster aiming to maximize efficiency (lines 29-43), again using the same, projection-based method we use to select a cluster for a new application. Lastly, the best frequencies are chosen, both for the placement cluster of the new application as well as for the target cluster of the moved application (lines 55-59). Note that in case the algorithm was called as a result of an application termination, only the new application placement part is skipped (lines 5-17). Moving applications is still possible.

The three main for-loops of Heuristic 3 have an asymptotic complexity of $\mathcal{O}(n{*}log\, n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n * log\, n)$, respectively. Thus, its overall complexity is $\mathcal{O}(n * log\, n)$, but in absolute terms it is expected to take longer than Heuristic 2, despite having the same complexity.

Algorithm 3: Heuristic 3

```
 1: INPUTS: Newly-spawned application $app_{new}$.
 2: OUTPUTS: A core index for placement of the application and a frequency for the chosen cluster. A
    (source, target) pair for moving a running application from the source core to target core and a frequency
    for the target cluster.
 3: //deciding core placement for the new application
 4: float eff_best = 0, eff_temp; int placement_core = -1, placement_cluster = -1;
 5: for $i = 1$ to $m$ do
 6:     if there is a core $j$ available in cluster $i$ then
 7:         eff_temp = placement_predict(application $app_{new}$, core $j$);
 8:         for all cores $k$ in cluster $i$ except $j$ do
 9:             eff_temp = placement_predict(application $app_k$, core $k$);
10:         end for
11:         if eff_temp > eff_best then
12:             eff_best = eff_temp;
13:             placement_cluster = $i$;
14:             placement_core = $j$;
15:         end if
16:     end if
17: end for
18: //finding current least efficient application
19: float app_eff, worst_app_eff = $\infty$; int source_core = -1;
20: for $i = 1$ to $n$ do
21:     if core $i$ currently executes an application then
22:         app_eff = estimate_app_efficiency(core $i$);
23:         if core_eff < worst_core_eff then
24:             worst_core_eff = core_eff;
25:             source_core = $i$;
26:         end if
27:     end if
28: end for
29: //deciding core placement for the moved application
30: eff_best = 0, eff_temp; int targer_core = -1, target_cluster = -1;
31: for $i = 1$ to $m$ except cluster of source_core do
32:     if there is a core $j$ available in cluster $i$ then
33:         eff_temp = placement_predict(application $app_{source\_core}$, core $j$);
34:         for all cores $k$ in cluster $i$ except $j$ do
35:             eff_temp = placement_predict(application $app_k$, core $k$);
36:         end for
37:         if eff_temp > eff_best then
38:             eff_best = eff_temp;
39:             target_cluster = $i$;
40:             target_core = $j$;
41:         end if
42:     end if
43: end for
44: //deciding frequency of the both affected clusters
45: int freq_best_1, freq_best_2;
46: freq_best_1 = choose_frequency(placement_cluster);
47: freq_best_2 = choose_frequency(target_cluster);
48: return(placement_core, freq_best_1, source_core, target_core, freq_best_2);
```

3.3.4 Dealing with Performance Requirements

Concluding our heuristics' description, we present some rules concerning specific cases, mainly dealing with applications that have a performance requirement and require priority treatment. The notes of this subsection do not change the complexity of any algorithms and were left out of the initial description to reduce clutter.

First, any system configuration that does not respect performance requirements of applications is considered invalid and cannot be chosen. This is implemented by having the various routines return an efficiency value of 0 if the IPS prediction of an application is below the required. To exemplify, in line 9 of Algorithm 1, the routine *placement_predict()* is called, which involves predicting the IPS of application app_{new}. If this predicted IPS is lower than the performance requirement of app_{new}, the routine returns an efficiency value of 0, ruling out this placement option. Another example is routine *frequency_predict()* in line 9 of Algorithm 0. If the *low* frequency does not satisfy some application's performance requirement, it is considered to have an efficiency of 0. Note that the above means that an application not currently fulfilling its requirement is a candidate to be moved to a bigger core by heuristic 3, as its estimated efficiency is also 0.

On a related note, all algorithms' decisions are partially based on predictions, which cannot be perfect. When performance requirements exist, this inaccuracy has to be outweighed by a degree of pessimism in the decision. To achieve this, a $slack$ value is defined for each algorithm. Whenever the performance of an application with a requirement is predicted, the algorithm attempts to fulfill a tighter requirement IPS_tight, instead of the real one, IPS_min, such that:

$$IPS\_tight = IPS\_min * (1 + slack)$$

The choice of $slack$ values in our experiments was guided by the average and worst-case inaccuracy of our performance model, presented in Section 3.4.1.

Furthermore, applications with a requirement are given priority over other applications when it comes to core placements. When a constrained application is placed, all cores executing a non-constrained application are considered potentially free (of course, an actually free core on the same cluster is always preferred). Thus, checks such as line 8 of Algorithm 1, take this into consideration. The application that is bumped out of a core in this case is placed on the most similar available core.

Lastly, an application which cannot be placed because there is no available core in the system, is put on hold until a core becomes available. In accordance with the rule stated in the previous paragraph, this is not true if the application has a performance requirement - in which case, a different application is paused.

3.4 Evaluation

In this section we present experimental results evaluating our energy efficiency optimization approach. Our experiments were performed on an ODROID XU3 board [91], which is built around an Exynos 5422 Processor, consisting of a cluster of four "LITTLE" Cortex-A7 cores and a cluster of four "big" Cortex-A15 cores. Each core has a private L1 data cache of 32KB, while L2 is shared per cluster and is 512KB for the little cluster and 2MB for the big. The two clusters share a 2GB LPDDR3 RAM main memory. The block diagram of the Exynos chip is shown at the left part of Figure 3.1, labelled "Hardware". We have determined the frequency range of the little cluster to be between 500MHz and 1.4GHz and that of the big cluster between 800MHz and 1.8GHz - both in steps of 100MHz. We have observed that scaling the frequency below these points does not reduce the voltage, offering diminishing power benefits. Other related works have limited the frequency range similarly [37].

We have used two sets of applications in this work. The first consists of NAS [92] and synthetic benchmarks listed in the top half of Table 3.2 and was used for calibrating the power estimation model expressed by Equation 3.2. The second set consists of NAS [92] benchmarks, the automotive MiBench suite [80] and miscellaneous benchmarks listed in the bottom part of Table 3.2 and was used to evaluate our proposed approach. We tried to have a balanced mix of compute-and memory-bound applications in both sets, to achieve both a good model calibration and a fair evaluation. When it comes to choosing which NAS benchmarks are part of our evaluation suite, we select the ones the execution time of which is of the same order of magnitude as the rest of the benchmarks (typically less than a minute on a big core), to keep experimentation time within practical boundaries (around ten minutes per run).

The application spawning for the experiments described in Sections 3.4.1 and 3.4.2 is done with an event generator tool, based on the same mathematical foundation as the one described in Section 2.4.1.2 of Chapter 2. In this case, the tool generates exponentially distributed application spawn events based on the following parameters for each application:

- A spawn rate, determining how often the application spawns. Multiple concurrent instances of an application are allowed to exist, so we run multiple random simulations of the exponential distribution in parallel, some of which will spawn an event in the allotted time.

- A workload range that determines how long the application will run, defined by a minimum and maximum number of iterations that have to complete before terminating. For each spawn event, a value from this range is chosen in a uniform random manner. The runtime manager is not aware of this number, as completion times are supposed to be unpredictable.

- A probability of the application having a performance requirement, and a range of possible performance requirements, expressed in IPS. The event generator first decides whether or not the spawned application has a performance requirement and if so, it chooses a performance value from the available range in a uniform random manner.

The runtime manager runs on a little core, unless otherwise noted.

Table 3.2: The applications used for calibration of the power prediction model (top) and for evaluation (bottom).

Calibration Applications	
FT - Class A (NAS)	BT - Class A (NAS)
MG - Class A, 2 different phases (NAS)	LU - Class A, 2 different phases (NAS)
EP - Class A (NAS)	
FP arithmetic (synthetic)	Integer arithmetic (synthetic)
Vector with random column multiplication (synthetic)	
Evaluation Applications	
CG - Class A (NAS)	IS - Class A (NAS)
SP - Class A (NAS)	Bitcount large (MiBench)
Basicmath large (MiBench)	Qsort huge (MiBench)
Susan large, all parts (MiBench)	NQueens_15 (Miscellaneous)
Linpack_2000 SP (Miscellaneous)	Matmul_512 DP (Miscellaneous)
Whetstone DP (Miscellaneous)	

3.4.1 Exhaustive Evaluation for Single Events

In this section we perform exhaustive evaluation of the prediction model and the heuristics, by comparing to an oracle model and an exhaustive search algorithm, respectively. We base this evaluation on nine representative application spawn events and examine in depth our model's and heuristic's response to them. The nine events are created as follows:

1. With our event generation tool, we construct three starting states for the system, corresponding to low (25%), medium (50%) and high (75%) core utilization.

2. We select three of the evaluation applications: Basicmath being completely compute-bound, Matmul as the most memory-bound (we set the matrix size to a high enough value to make it so) and IS as one of the intermediate applications in terms of memory boundedness.

3. We spawn each of the three applications on each of the three starting system states, thus creating nine distinct and diverse events.

For each event, we enumerate all possible configurations, in other words all combinations of new application placement, frequency of little cluster and frequency of big cluster. We use our model to predict performance, power and energy efficiency (in terms of IPS/Watt) for each possible decision (system configuration). Then, we apply all possible decisions one by one and measure the actual performance, power and efficiency. The left part of Table 3.3, titled "Model Misprediction" shows, for each event, the average inaccuracy of our model in predicting these three figures, across all possible decisions.

We observe that both parts of our model perform satisfactorily, as the average error is always less than 10%. Performance prediction is more accurate (2.5% average error) than power (6.1% average error). In some cases performance and power mispredictions add up, resulting to a higher efficiency error (e.g. event #1), while in other cases the opposite happens (e.g. event #6). Furthermore, the performance of the compute-bound application Basicmath is easier to predict (maximum average error 1.52%), as it does not depend much on contention for shared resources, thus it is more straightforward to infer from offline data. However, the performance misprediction for IS and Matmul, which are harder to predict, is not much worse (maximum error 4.68%).

Subsequently, for each event, we compare the efficiency of the following four runtime managers:

1. An exhaustive oracle predictor that uses the actual measurements to exhaustively search and select the most efficient configuration. As it always makes the optimal choice, this manager is used as a baseline.

2. Our prediction model, exhaustively evaluating all possible decisions and applying the one it thinks best. Comparing this manager to its oracle counterpart (1.) evaluates our prediction strategy, independent of the heuristics' quality.

3. Heuristic 1 guided by an oracle predictor. Comparing this manager to its exhaustive counterpart (1.) evaluates our basic heuristic independent of the quality of our prediction model.

4. Heuristic 1 guided by our prediction model. Comparing this manager to (1.) which is both oracle and exhaustive, evaluates our complete proposed approach.

The last two steps focus on only one heuristic, due to the complexity and duration of the required exhaustive experiments. However, the efficiency of the rest of our heuristics is comparatively evaluated in the next subsection. The right half of Table 3.3 shows the efficiency loss of using managers (2.) ("Model + Exhaustive"), (3.) ("Oracle Model + Heuristic") and (4.) ("Model + Heuristic"), compared to the optimal exhaustive oracle predictor (1.).

When it comes to accuracy of the final decision, we observe that the efficiency loss of exhaustively evaluating all possible configurations with the model (Model + Exhaustive), is 3%, which is lower than the average misprediction of the model (6%). This happens because misprediction is usually uniform among different configurations (i.e. either always optimistic or always pessimistic). Thus, despite the absolute error, the model consistently captures the efficiency trend, resulting to a configuration very close to the optimal one (same placement of the new application, very similar cluster frequencies). The only exception is event #5. For all three IS scenarios, the placement decision (big or little core) makes less than 10% difference in the final efficiency. Thus, in one case the misprediction was enough for the model to choose the wrong core placement, resulting to 8.0% energy efficiency loss compared to the optimal. Furthermore, when applying Heuristic 1 guided by the oracle predictor, we get an average 2.5% energy efficiency loss compared to the optimal, proving that the heuristic searches the correct part of the solution space - the efficiency loss is due to small deviations from the optimal frequencies.

Table 3.3: Exhaustive evaluation of the prediction model, our heuristic and the strategy as a whole.

			Model Misprediction			**Solution Efficiency VS Optimal**		
#	Core Util.	Application	Performance Mean (std dev)	Power Mean (std dev)	Efficiency Mean (std dev)	Model + Exhaustive	Oracle Model + Heuristic	Model + Heuristic
1	Low	Basicmath	1.52% (1.34%)	8.87% (5.24%)	9.40% (5.99%)	4.0%	4.3%	5.1%
2	Medium	Basicmath	0.95% (0.72%)	8.58% (5.59%)	9.18% (6.61%)	1.3%	1.2%	1.3%
3	High	Basicmath	0.76% (0.60%)	7.11% (4.73%)	7.50% (5.37%)	2.2%	1.4%	1.6%
4	Low	IS	1.85% (1.37%)	2.85% (1.60%)	3.80% (2.40%)	3.0%	4.5%	5.5%
5	Medium	IS	3.20% (2.38%)	3.28% (2.40%)	3.81% (2.86%)	8.0%	0.5%	3.9%
6	High	IS	4.68% (2.87%)	4.24% (3.10%)	3.18% (2.85%)	0.2%	1.9%	0.2%
7	Low	Matmul	4.52% (4.17%)	6.90% (6.52%)	5.62% (5.13%)	0.6%	5.5%	3.1%
8	Medium	Matmul	3.27% (2.93%)	7.12% (6.03%)	6.37% (5.08%)	6.7%	3.7%	4.6%
9	High	Matmul	1.92% (1.82%)	5.64% (4.15%)	5.12% (3.98%)	1.4%	2.1%	1.4%
Average (min, max)			2.52% (0.76%, 4.68%)	6.07% (2.85%, 8.87%)	6.00% (3.18%, 9.4%)	3% (0.2%, 8%)	2.5% (0.5%, 5%)	3% (0.2%, 5.5%)

This is only slightly increased to 3%, when we use the actual model in place of the oracle, reinforcing our previous point of the model misprediction being absorbed when comparing different candidate configurations.

3.4.2 Evaluation with Dynamic Scenarios

In this section we evaluate our approach as a whole, with an extensive set of unpredictable, dynamic multi-application scenarios. We have combined the evaluation applications of Table 3.2 into eight application mixes, summarized in Table 3.4. Mixes 1 to 3 consist of applications coming from the same source (NAS, MiBench automotive and miscellaneous benchmarks). Note that mix 2 is also used in similar experiments of [37]. Mixes 4 to 8 aim to create different combinations of compute- and memory-bound applications. We have sorted all applications in order of decreasing *scaling factor (SF)* - the lower the SF, the more memory-bound the application is. Based on this classification, we include the most compute-intensive applications in mix 4, the most memory-bound ones in mix 5 and the intermediate ones in mix 6 (hence called "Uniform Balanced"). Mixes 7 and 8 are also balanced, but they include applications from both extremes, instead of just the intermediate group.

For each of the eight mixes, we use our event generator to create 20 sequences of application spawns in random, exponentially distributed moments in time. Among the 20 event sequences, we vary the applications' spawn rates in such a way that the peak system utilization is *expected* to be 25% in the first sequence, to 75% in the last. We allow applications to spawn during the first two minutes of each run. Subsequently, we carry on the run until all applications have terminated. We evaluate the efficiency of the run in terms of average IPS/Watt, based on the total instructions executed across all applications, the total time that the experiment took until all applications' completion and the average power during this time.

In the experiments of this section, we evaluate all three of our heuristics. On top of this, we also evaluate an alternative version of Heuristic 3 (called "Alt. Heuristic 3"): in this version, in addition to application spawns and terminations, the algorithm is also called whenever a running application completes one iteration and updates its online monitoring data. In this manner, we give Heuristic 3 the chance to take corrective action in more regular intervals. We compare our heuristics both with each other and with existing Linux governors, namely the *powersave*, *interactive* and *ondemand* governors, as is common in related works [31, 32]. For all experiments, we keep the default idle governor active, allowing it to put cores to idle states, when they do not execute an application [95]. Generally speaking, the *powersave* governor is more energy-efficient, as it chooses the lowest available frequency, which tends to be the most efficient, especially in the range we have chosen to allow. On the other hand, for the same reason, the *powersave* governor cannot fulfill performance requirements of applications, contrary to both *interactive* and *ondemand*, that prefer higher frequencies. To provide a complete and fair evaluation versus all available governors, we perform the following two separate sets of experiments:

Table 3.4: Application mixes used for energy efficiency evaluation of the proposed heuristics.

Mix #	Description	Applications
1	NAS Benchmarks	CG, IS, SP
2	MiBench Benchmarks	Basicmath, Bitcount, Qsort, Susan
3	Miscellaneous Benchmarks	Linpack, Matmul, Whetstone, NQueens
4	Compute-bound	Basicmath, Bitcount, Whetstone, NQueens
5	Memory-bound	Matmul, CG, SP
6	Uniform balanced	Susan, Linpack, IS
7	Non-uniform balanced	Basicmath, SP, Linpack, Susan
8	Non-uniform balanced	Bitcount, CG, Linpack, IS

1. Ignoring the applications' performance requirements. This set of experiments evaluates our approach mainly versus the *powersave* governor, in the simpler scenario in which performance considerations do not exist and the only goal is maximizing energy efficiency (IPS/Watt).
2. Considering the applications' performance requirements. To fulfill the requirements, our algorithms have to choose, on average, higher frequencies than in set (1). This usually entails an efficiency penalty, making it hard to compete versus the *powersave* governor. This set of experiments aims to demonstrate how our approach is able to fulfill the performance requirements, while at the same time maintaining an efficiency advantage versus the governors that are able to achieve this (*interactive* and *ondemand*).

The efficiency results for both sets are illustrated in Figure 3.4. The top plot (a) shows results for set (1) and the bottom one (b) for set (2). For each application mix and runtime manager, the efficiency shown is the mean across the 20 runs, after each run is normalized to the value for the *powersave* governor. The bottom plot (b) also shows, for each governor and heuristic, the average performance degradation with respect to individual applications' performance requirements.

We observe the following in Figure 3.4:

Comparison to governors: We observe that, without performance requirements, our approach matches the efficiency of the powersave governor, being up to 3% better on average (using Heuristic 3) and is twice as efficient compared to the other governors. In this experiment without performance requirements, being on par with the powersave governor is especially important. When performance requirements are introduced, our approach can satisfy all of them, with Heuristic 3. Even Heuristic 1 satisfies almost all requirements, resulting to an average degradation versus target performance requirement of 0.24%. Dealing with requirements comes at an expense of 17% to 21% of energy efficiency versus the powersave governor, which however does not fulfill requirements (it is on average 23% off the performance target). Additionally, Heuristic 3 achieves 52% and 58% better efficiency than the interactive and ondemand governor respectively, while satisfying the same requirements.

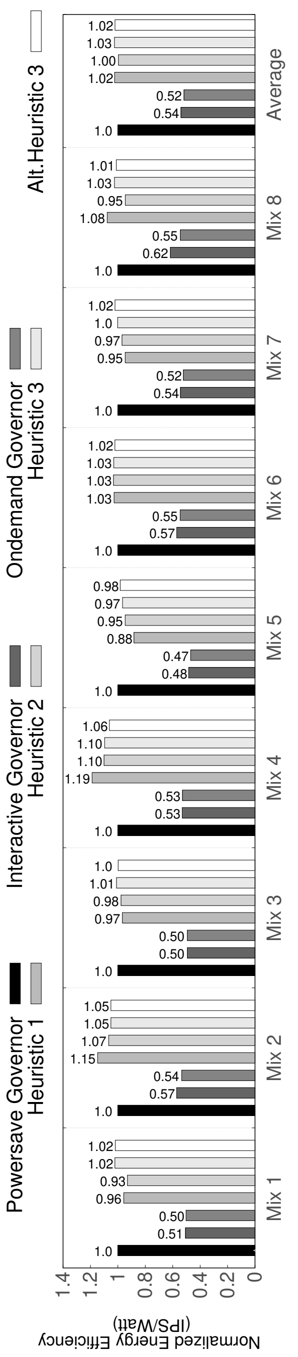
Comparison among the heuristics: The relative energy efficiency of the heuristics depends on the scenario (application mix and presence or absence of performance requirements). Without performance requirements, Heuristic 1 is better with mixes 2 and 4 that consist mostly of compute-intensive applications. Predictions for these applications are more straightforward, as there is no contention in the levels of the memory hierarchy. Thus, the simpler approach of Heuristic 1 is sufficient to detect an efficient configuration, while heuristics 2 and 3 gain nothing by their detailed prediction and/or corrective action capability. In fact, these extra steps are potential sources of misprediction, which sometimes hurts decision efficiency. The opposite trend is observed for mixes 1 and 5, consisting mostly of memory-bound applications. Performance of these applications varies more with respect to the current system status, thus the deeper prediction scheme of Heuristics 2 and 3 and the corrective action potential of Heuristic 3 benefit efficiency. Balanced mixes (3, 6, 7 and 8) show the three heuristics achieving very similar results. Still, Heuristic 3 is best in mixes 3 and 7 by 3.1% to 5.3% and is only 0.2% behind in mix 6.

Introducing performance requirements also complicates predictions, as the runtime manager has to raise the frequency enough to satisfy the requirement but not more, in order not to penalize efficiency more than needed. As a result, Heuristic 1 is relatively worse in some of the mixes (1 and 8), while it also misses more performance requirements than the other heuristics. Its average efficiency is still as good as Heuristic 3, mainly because of the outlier value it scores for mix 4. Heuristic 3 is best in 4 out of the 8 mixes, while its ability for corrective action (prioritizing performance requirements when doing so) results in all requirements being satisfied.

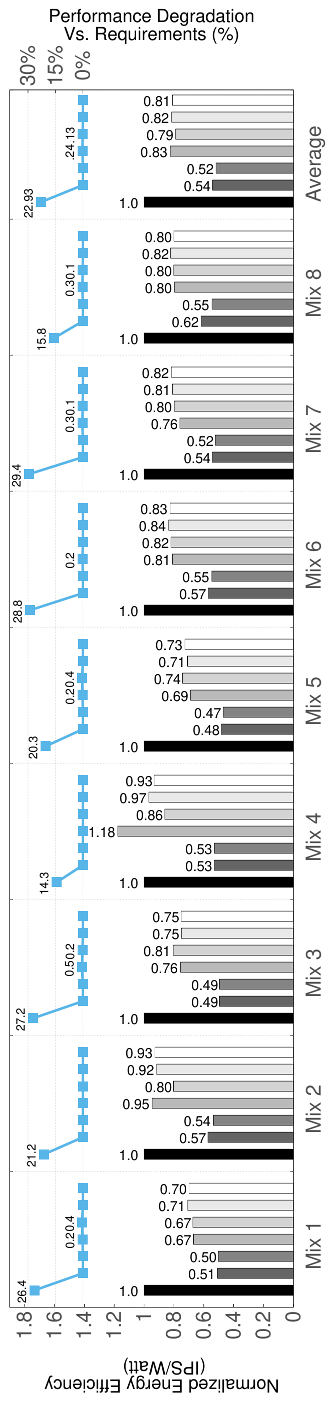
Comparing the original version of Heuristic 3 with its alternative version which is invoked more frequently, we find that it does not benefit from increasing the chances for corrective action. The reason is that, in our current implementation, application migrations can only happen between two iterations of an application. If a migration is decided, the iteration currently running, starts over on the new core. This generally penalizes Heuristic 3, but its importance is increased for the alternative version that performs more migrations. Still, we occasionally notice the alternative version of Heuristic 3 performing the same migrations as the original version, but sooner, resulting to incremental gains (mixes 5 and 7). As the original version eventually takes the same action, this benefit remains low. However, in most mixes, especially mix 4, the alternative version migrates applications more aggressively, penalizing overall efficiency because of the reason explained above.

Comparison to related works: Our results show improvement compared to related works in different ways.

The most recent approach that evaluates dynamic multi-application scenarios [37], unfortunately does not compare to the powersave governor. The most relevant comparison to this work is versus an allocator aiming to maximize energy efficiency (MAX-EE), which is described as "an optimal brute-force allocation for maximizing IPS/Watt". It has to be noted however, that this allocator occasionally achieves lower IPS/Watt than others, meaning that it cannot be optimal *with respect to the reported result.* It is, of course, possible that it is optimal in a short-term manner, with respect to individual



(a) Without Performance Requirements



(b) With Performance Requirements

Figure 3.4: Energy efficiency in terms of IPS/Watt, for all governors, heuristics and application mixes, without (a) and with (b) performance requirements. All values are normalized to the efficiency of the powersave governor. When considering performance requirements, the average performance degradation compared to the requirement is also shown (scale on the right-side vertical axis).

Table 3.5: Execution time of the three proposed heuristics for various system and cluster sizes.

	#cores (n):	8	16		32		64		128	
	#clusters (m):	2	2	4	4	8	8	16	16	32
Heuristic 1	Big, 800MHz	80us	104us	74us	113us	103us	146us	135us	193us	206us
$\mathcal{O}(m * log\ n)$	Little, 500MHz	170us	240us	197us	281us	248us	338us	330us	436us	475us
Heuristic 2	Big, 800 MHz	95us	144us	135us	170us	172us	241us	253us	383us	411us
$\mathcal{O}(n * log\ n)$	Little, 500 MHz	215us	305us	297us	383us	386us	539us	550us	857us	887us
Heuristic 3	Big, 800MHz	128us	202us	191us	241us	243us	347us	356us	567us	576us
$\mathcal{O}(n * log\ n)$	Little, 500MHz	301us	409us	407us	547us	552us	761us	772us	1210us	1229us

decisions taken throughout a run - which does not guarantee optimal energy efficiency for the whole run. Additionally, for this experiment, the various allocators being compared are guided by an oracle predictor instead of the model proposed in the SPARTA paper. Furthermore, although SPARTA is evaluated both through simulations and through runs on an ODROID XU3 board, comparison with the MAX-EE allocator is only performed in the former. Thus, on the simulated ODROID XU3, the SPARTA allocator achieves around 3% worse energy efficiency than the brute-force MAX-EE allocator, while our approach achieves 3% better than the powersave governor. Additionally, the execution time of the SPARTA allocator [37] explodes when the system size exceeds 32 cores, which, as will be shown in Section 3.4.3, is not true for our approach.

It is also important to compare with very recent works that focus on a slightly different paradigm [31, 32], as they do not evaluate for dynamic, multi-application scenarios. DyPO [31], which improves the results of [32], achieves 6% efficiency improvement over the powersave governor, which is however allowed to select frequencies as low as 200MHz. On the contrary, we limit the powersave governor to a minimum frequency of 500MHz and 800MHz for the little and big clusters, respectively, as for lower frequencies voltage is not further reduced diminishing the power benefits. According to our experiments, the powersave governor is at least 19% more energy efficient with our setup. On this setup, our approach achieves 3% improvement versus the powersave governor.

3.4.3 Execution Time of the Heuristics

In this section, we report our heuristics' execution time and how this scales with respect to system size. These results are summarized in Table 3.5. We have scaled the number of cores in the system from 8 (the size of our available board) to 128. For sizes more than eight cores, we measure execution time both for the cluster size of our available board (four cores) and double this size (eight cores).

Comparing between the algorithms, the execution time of Heuristic 1 grows slower: it takes 2.6 to 2.8 times more for a 128-core system than for an 8-core system, whereas for heuristics 2 and 3 this factors are 4.1 to 4.3 and 4.1 to 4.5 respectively. Furthermore, the execution time of Heuristic 1 grows faster for systems with more clusters of fewer cores, than for systems with fewer clusters of more cores, which is expected due to its dependence on the number of clusters, m. Indeed, for 8-core clusters, Heuristic 1 takes 1.9 times longer on a 128-core system compared to a 16-core system, whereas for 4-core clusters this factor is 2.8.

More importantly, all heuristics' execution time scales sub-linearly with system size, as it never doubles when doubling n. This shows that the theoretical upper bounds discussed in Section 3.3 are pessimistic. This result proves that all algorithms can be used on systems many times larger than the one we have experimented on, without concerns about their response time, because of their ability to perform partial search of the configuration space without sacrificing much efficiency. As an indication of how the configuration space grows with respect to system size, we have measured the execution time of a lightweight version of exhaustive search, which considers all options, but only for the new application placement and for the frequency of the selected cluster. We found the execution time of this algorithm to be at least double than our longest heuristic for the smallest system size. More importantly, it grows much faster than our heuristics when increasing the system size: when going from 8 to 128 cores, this limited exhaustive algorithm takes at least 16 more times to execute, compared to the 2.6 to 4.5 factors reported above for the heuristics. Furthermore, this limited exhaustive search, does not even consider the full configuration space. To do so, it would have to also consider all possible migrations of current applications, which would multiply execution time by a factor growing exponentially with the system size.

3.5 Limitations and Alternatives

In this section we list some limitations and alternatives of the proposed approach, in order to clarify the extent of our contributions.

As is evident by the presentation of our performance prediction strategy in Section 3.2, it is assumed that applications appearing at runtime have been profiled offline. The results of the offline profiling are used as a starting point for the projection-guided runtime prediction. Our approach is indeed partly based on this offline step, thus we focus on making it feasible, by imposing two constraints: (i) profiling of each application should be standalone, regardless of the possible application mixes that can appear at runtime and (ii) it should consist of a linear (with respect to the target system size) number of runs. The above does not mean that our runtime manager is unable to handle unknown applications, just not as accurately as profiled ones. If an application is unknown in the most extreme sense (not even one profiling run on one core type was possible to carry out offline), then at runtime it would initially have to be placed on an arbitrary core and its online performance monitoring would subsequently have to be used for any future decisions regarding this application, much like is done in [37].

A limitation, regarding mostly our experimental setup, is the assumption of at most one thread per core. As the ODROID XU3 offers one set of performance counters per core, it is challenging to monitor more than one threads running concurrently on the same core and keep the measurements independent and consistent. However, regardless of the current implementation, our approach in general is not limited in this respect. Allowing multiple threads per core would not change the complexity of the runtime heuristics, as the least busy core of each cluster (with respect to its utilization) would still be preferred for application placement. However, profiling would have to be extended, running each application for up to $c * n$ instances, instead of n, c being a constant representing the maximum number of threads per core expected in runtime situations. In this manner, the contention between threads for processor cycles would also be considered, when performing projection to predict IPS in this more challenging contention scenario. While this would not change the complexity of profiling, it would prolong it by a factor of c.

Another factor we have not considered in this work is application phase changes. While we deem this potentially beneficial to efficiency, we think that it is orthogonal as it does not fundamentally change the approach: if a phase change detection mechanism is in place, a phase change can be treated by our proposed approach, as a termination of an application, and a simultaneous spawn of another, with different performance characteristics. Additionally, each application phase (rather than each whole application) should then be offline profiled and characterized, which strengthens the importance of having a scalable profiling strategy. Phase change detection can be performed by observing the online-monitored performance measurements and detecting significant changes. DyPO [31] proposes a way to implement phase change detection.

Lastly, a limitation of our current implementation is the ability to migrate applications only at specific times. Each application consists of a basic loop that iterates a number of times decided by our event generator (see Section 3.4). Migrations can happen only between iterations of this loop - if a migration is decided, the iteration currently running has to start over on the new core. This actually punishes the efficiency of Heuristic 3 and also minimizes the effectiveness of trying to migrate applications more often: calling Heuristic 3 in regular intervals, instead of only spawns and terminations, achieves earlier migration of some applications on a more efficient core, but the original version of the heuristic eventually does the same. As our focus is on providing an efficient initial response, the corrective action capabilities of Heuristic 3 are already sufficient to remedy any bad decisions coming from model inaccuracy. The challenge of identifying the optimal migration frequency has been studied in [96].

3.6 Chapter Summary

In this chapter we propose a runtime manager for single-ISA, heterogeneous, DVFS-capable multiprocessor systems, aiming to improve energy efficiency. The manager uses a prediction model to estimate performance and power for possible configurations and a heuristic to choose candidate solutions.

Our prediction model estimates application performance using scalable profiling of applications that can run on the system and projecting this information to the current system status, as measured through online monitoring. Subsequently, it uses these performance estimations to predict system power with an analytical model based only on the specific platform's characteristics. Through exhaustive evaluation, we found our model to mispredict performance, power and energy efficiency by an average 2.5%, 6.1%

and 6% respectively.

We have designed three alternative heuristics, the execution time of which grows sublinearly with respect to the system size. Guided by an oracle model, our simpler heuristic is able to select a configuration only 2.5% less efficient than a brute-force, exhaustive approach.

Our approach as a whole (model and heuristics) is evaluated with a plethora of dynamic, multi-application scenarios. Comparing to widely-used governors, when not considering individual applications' performance requirements, our solution is 3% more energy efficient than the powersave governor and twice as energy efficient compared to the interactive and ondemand governors. Our approach is able to support applications with performance requirements at the cost of 18% lower energy efficiency versus the powersave governor, which however misses the performance targets by 23%. Furthermore, it maintains an efficiency advantage of 52% and 58% over the interactive and ondemand governors, respectively, which can satisfy all requirements.

4 Runtime Management for Load Balancing

4.1 Introduction

In both optimization problems we have tackled so far in this thesis, heterogeneity played a pivotal role in providing flexibility: different kinds of cores create decisions both when managing for graceful degradation, as a task can be moved to a slower core if the core executing it fails, and when managing for energy efficiency, as different cores provide different performance-power points. In this chapter, we shift the focus to homogeneous systems of arbitrary size, topology and memory hierarchy. One very important challenge when attempting to extract as much parallelism as possible out of these systems, is balancing the workload among the available cores.

Indeed, as the number of cores in computer systems increases and memory hierarchies deepen, exploiting the available performance potential becomes increasingly difficult. In order to achieve high performance, a common trend is to decompose applications into concurrent tasks, utilizing as many resources as possible. Programmers, compilers and run-time systems try to distribute these tasks evenly to the machine resources to keep the system's load balanced. Nevertheless, at run-time, workload imbalances occur due to unexpected fluctuations in execution time of tasks, synchronization events, system events triggering interrupts or the execution of daemons in the background. More importantly, certain workloads are by nature severely imbalanced and/or spawn new tasks dynamically, possibly in an unpredictable manner. Therefore, in order to dynamically balance the load in the system and maximize its utilization, researchers have proposed techniques for idle cores to execute the work that was originally assigned to currently busy cores. This technique is known as *Work Stealing* [39].

In *Work Stealing*, tasks are migrated between cores, to improve system utilization, but at the same time various side-effects are introduced. We identify two key factors that may affect performance:

1. the selection of a busy core to steal from (which we henceforth call *victim core*) and

2. the overhead of migrating a task and its working set to a new core.

In particular, *local* load imbalances are better mitigated by stealing between *close neighbors*, to preserve data locality and minimize migration-related overheads. Thus, certain techniques attempt to limit migrations only to neighboring cores [41]. However, *system-wide* imbalances require more drastic workload re-distribution. An efficient *Work Stealing* technique is expected to improve performance in all of the above diverse scenarios, while not penalizing it when the workload is already balanced.

For the selection of a busy victim core, it is possible to use simple techniques such as *random work stealing* [40], which is agnostic of both system load and topology information. Although on average random work stealing can benefit performance, its statistical nature means that it tends to produce inconsistent benefits among runs. A more educated guess for the selection of the victim can be performed by consulting system load information. This can improve the ratio of successful steals and increase efficiency. However, we have found that the cost of obtaining system load information may be too high, as the information has to be broadcast through the system using the same resources as the application. The same argument has also been made by other researchers, who like we do in this chapter, propose load balancing techniques that try to infer the current system load from partial information, such as [48].

In this chapter we propose a technique in which each core lazily gathers partial information about the system load distribution, as feedback from the cores queried during steal attempts. This information corresponds to a subset of the system cores, providing an incomplete and approximate view of the system load distribution. The information is stored locally in each core, therefore not introducing any bottleneck for the rest of the system. This approximate view is then used by the core, when it becomes idle, to guide the selection of the next steal victims. In addition, the steal attempts are performed considering the topology and memory hierarchy, thus favoring neighboring cores as potential victims, reducing migration cost.

The proposed technique, which we refer to as SWAS (*Stealing Work using Approximate System-load information*), is implemented on the GO:TAO task-based runtime system [97] and evaluated on a 48-core machine. In order to test the effectiveness of the work stealing technique, workloads representing different scenarios were used. In particular, we test workloads which are balanced, affected by jitter or imbalanced, as well as workloads with random and dynamic task creation. A good work stealing strategy is able to improve the performance of workloads with diverse kinds of load imbalance, while not penalizing the balanced ones.

The main contributions of the work presented in this chapter are the following:

- Introduction of a novel lightweight method to construct an *approximate system load view* by integrating load querying with work stealing attempts.
- A work stealing algorithm that, upon the occurrence of a *core idle* event, performs hierarchical victim selection guided by the aforementioned *approximate system load view*.
- Implementation of the proposed technique in the GO:TAO framework and evaluation on a real 48-core machine.

The rest of the chapter is organized as follows: In Section 4.2 we discuss other work stealing techniques and discuss their limitations with respect to SWAS. In Section 4.3 we present SWAS. In Section 4.4 we describe our experimental setup and in Section 4.5 we present the evaluation of our strategy and compare it with other approaches. Finally, Section 4.6 summarizes and concludes the chapter.

4.2 Related Work

As a fundamental technique for load balancing, work stealing has been the subject of numerous studies. Researchers have explored distinct approaches to increasing the efficiency of work stealing, by improving the decisions about which task to steal and from which victim core. We identify four different categories of work stealing strategies: locality-aware, hierarchy-aware, system load aware, and user defined.

Yoo et al. quantified the potential benefits of *oracular* locality-guided stealing [44]. Their work focuses on unstructured parallelism, i.e. parallel loops that do not have any explicit information on locality. Their analysis indicates that, on a 32-core architecture, perfect-knowledge locality-aware work stealing may potentially improve performance up to $2.4\times$ compared to random work stealing.

A large body of research focuses on extracting static information, mainly about locality. The usefulness of compile-time locality hints is explored by Guo *et al.* [41] and Acar *et al.* [45]. Chen *et al.* [46, 47] propose a profiling method that partitions application DAGs corresponding to fully strict computations with leaf-only computation into cache-friendly subtrees at runtime. The task stealer is then extended to choose cache-friendly tasks for intra-socket stealing and other tasks for inter-socket stealing. Drebes et al. [98] proposed a more general approach that removes the restriction of iterative and leaf-only fully-strict computations but requires the programming model to be extended with information on task dependencies. To handle locality, their approach implements a hierarchical work stealer in which a set of stealing attempts is done at one level of the memory hierarchy before starting to look for work in nodes belonging to the next level of the memory hierarchy. It has to be noted that these approaches require knowledge and compile-time manipulation of the applications that can run on a system.

One piece of information that is relevant in any kind of strategy is that of the underlying system hierarchy. System hierarchy affects the stealing overhead by determining the cost of migrating working sets among cores. In CRS [42] a steal request is sent to a remote system region and, while this request is in flight, local stealing is attempted. HotSLAW [43] augments the locality-aware strategy proposed in [41] with support for arbitrary hierarchies. Note that both locality- and hierarchy-aware strategies tend to be state-agnostic, meaning that no knowledge about the load of other cores or regions of the system is assumed when making a work stealing decision. As such, these approaches focus on reducing the overhead that comes from communication between cores and from coherence maintenance when a steal disrupts data locality.

For system load aware work stealing strategies, the challenge is balancing the accuracy of such information against the cost to obtain it. In Match-making [99] dedicated "matchmaker" cores collect requests for help from busy cores. Cores which become idle can visit the matchmakers to find pointers to work that can be stolen. This technique adds a considerable overhead due to dedicating part of the available system resources (both computation and communication) entirely to improving work stealing decisions. A different approach in system load estimation is to use already available information as input to a heuristic. As an example, Tzannes *et al.* [48] propose the inspection of a core's own queue, to estimate the overall state of the system: an empty local queue points to the existence of "hungry" workers who are stealing tasks. Thereby, communication overheads are avoided, at the expense of using only local information to guess system load distribution.

An alternative approach is to adapt to application characteristics via user-defined policies. Nakashima *et al.* [100] develop an API that allows the user to attach a custom work stealing algorithm. Their approach requires full knowledge of the work stealing interface. A simpler, but also more constrained option is to provide configuration parameters that allow tuning the work stealing algorithm as recently suggested by Wimmer *et al.* [101]. These approaches may achieve better results than others mentioned above but they require an increased effort by the programmer.

In this chapter we propose a technique that is aware of the locality, hierarchy, and system load information. Our work extends previous approaches by combining knowledge of system load with system hierarchy information. Furthermore we demonstrate that an approximate and incomplete system load view is enough to reap the benefits of guided stealing and we propose a lightweight method to construct and maintain it.

4.3 System Load Aware Work Stealing

In this section we describe SWAS, the proposed work stealing strategy. In general terms, a work stealing strategy determines the way in which an *idle core* decides which other core of the system (victim core) to attempt to steal work from.

SWAS is based on approximate information about system load distribution. Each core has an incomplete view of this distribution, allowing it to guess which cores are busy and, as such, good victim core candidates. This partial knowledge is henceforth called *system view*. Every core has its own system view, stored in compact data structures and obtained through lightweight operations, that add negligible performance overheads and do not burden system communication. Section 4.3.1 describes the system view, and lists all information that it contains. Subsequently, Section 4.3.2 explains how each core updates and maintains its system view. Lastly, Section 4.3.3 demonstrates how the information stored in the system view are used by an idle core to guide its selection of victim core.

SWAS comes in two different variants, which differ in what information a core uses to update its system view. In the first variant, a core updates its system view based only on information about the load of cores it visits during steal attempts. In the second variant cores are in addition allowed to occasionally consult each other's system views, to update their own. In the description that follows, whenever a step is different among the two variants, this fact is highlighted.

4.3.1 System View Data Structures

SWAS is built on the notion of a system view. In this section, we present the structures used by each core to store this view. Note that every core in the system keeps its own system view.

To clarify our description, we use a running example based on the hardware platform shown in Figure 4.1. This system consists of four multiprocessor chips of 12 cores each, resulting in a total of 48 cores (with IDs 0 to 47). Inside each chip, the 12 cores are divided into two NUMA nodes of six cores each. The cores of each NUMA node share a L3 cache. This lowest level of hierarchy defined by the NUMA nodes, is henceforth also called a *neighborhood* of cores. Thus, a neighborhood is the widest subsystem within which work stealing incurs minimal overheads. The system of Figure 4.1 consists of eight such neighborhoods, numbered 0 to 7. Without loss of generality, we focus our example on the system view of core 7 of this example system.

4.3.1.1 Bit Vectors

The system view is kept in a vector of N two-bit entries, where N is the number of cores in the system. Each entry of this vector (henceforth called *bit vector*) is indicative of the current load of one system core, which can be one of the three values: *Not busy*, *Don't know* and *Busy*. In Figure 4.1, the example system has $N = 48$ cores, thus core 7 keeps 48 two-bit entries, shown at the top of the figure. All entries in the vector are initialized to the *Don't know* value.

4.3.1.2 Regions and Busy Estimators

Each core also keeps a number of static lists corresponding to levels of hierarchy of the system. Each of these lists contains the IDs of cores which incur similar overheads for stealing work from. Hence, the first list contains the IDs of the core's neighbors, while the last list contains the IDs of cores that do not share any level of hierarchy with the local core, except the entire system. Each such list defines a *system region*. The higher the region index, the further away the cores of this region are, meaning that the overheads for stealing work from them are higher. The neighborhood of the core is thus called Region-1

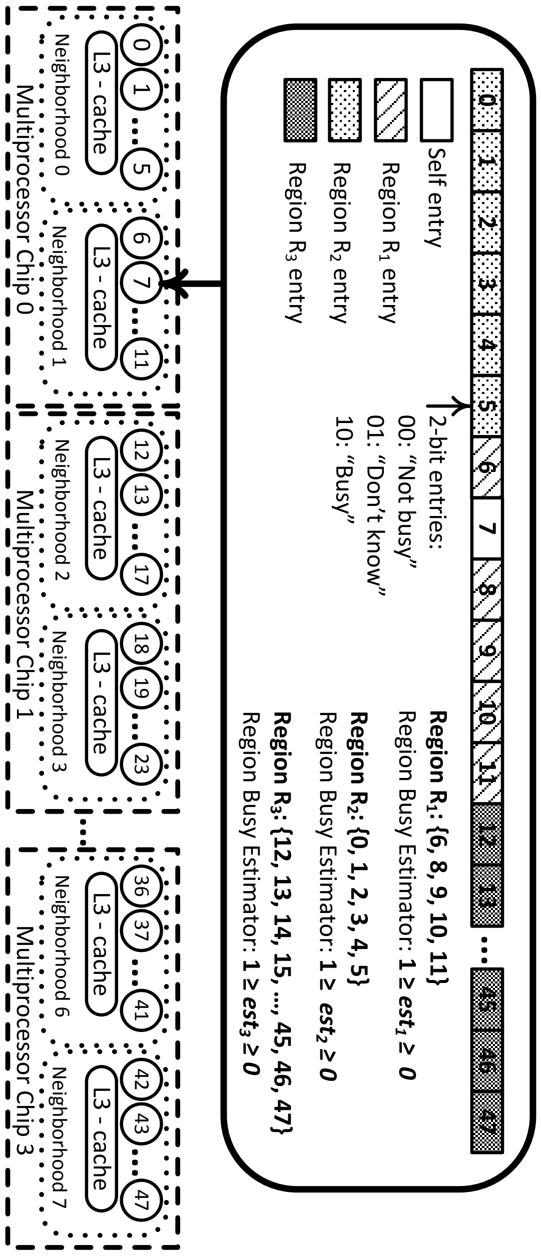


Figure 4.1: The system view that core 7 keeps for a 48-core system with multiple hierarchy levels.

(henceforth R_1) and subsequent regions, if applicable are R_2 up to R_{max}, where max is the number of levels in the physical system hierarchy.

The example system of Figure 4.1 has three levels of hierarchy: the eight neighborhoods, the four multiprocessor chips and the whole system. Thus, for core 7, the following three regions are defined, in accordance to the previous paragraph:

- Region R_1, consisting of cores 6, 8, 9, 10 and 11 which reside in the same neighborhood as core 7.
- Region R_2, consisting of cores 0 to 5 with which core 7 shares a chip, but not a neighborhood.
- Region R_3, consisting of all other cores (12 to 47), residing in a different chip than core 7. These cores are the furthest from core 7, thus it is expected that stealing from them incurs the highest overheads.

System hierarchy is specified in a configuration file that is made available to all cores at boot time. Based on this, each core defines the regions from its point of view.

For each region list, the core also keeps a non-negative variable, called a *Region Busy Estimator* (in this example, est_1, est_2 and est_3 are the estimator variables that core 7 keeps for its regions R_1, R_2 and R_3 respectively). These variables store an estimate of how loaded each region currently is, and are calculated by the contents of the bit vector as follows:

$$est_k = \frac{1}{|R_k|} \sum_{i \epsilon R_k} Load(i) \tag{4.1}$$

$Load(i)$, in turn, has the value 0, 0.5 or 1, when the bit vector entry for core i is *Not busy*, *Don't know* or *Busy* respectively. Thus, all estimators lie in the range $[0, 1]$ and direct comparisons between them are meaningful. Therefore $est_1 > est_2$ implies that region R_1 is currently busier, on average, than region R_2.

4.3.2 Updating the System View

In SWAS, *updating* and *using* one core's system view are interdependent: the system view is used to select a victim core for stealing work from, while the result of the steal attempt is used in turn to update the system view. Despite this interdependence, updating and using the system view are different processes. This section describes the process of updating the system view in SWAS, while the next (Section 4.3.3) focuses on how the system view is used to select a victim core.

Note that we focus the following description on updating the bit vectors. Any update to the bit vectors will cause the region estimator variables to be modified accordingly, as described in Section 4.3.1.

4.3.2.1 Updating Own System View

The cheapest way to update a core's system view is with information inferred directly by the core itself. In SWAS this happens during steal attempts.

Specifically, whenever an idle core accesses a victim core's queue, it updates its own system view based on the result of the steal attempt. This update also considers the system hierarchy as follows: When the victim core belongs to the idle core's region R_1, only the bit vector entry corresponding to the victim core is updated, whereas if it belongs to a region of higher index (thus, more remote than R_1), all entries corresponding to the victim core's *neighborhood* are updated.

Continuing our running example, suppose that core 7 in Figure 4.1, attempts to steal work from core 11 but does not find any. Since core 11 belongs to R_1, core 7 will update entry 11 in its bit vector, to the value *Not busy*, reflecting the lack of work to be stolen from core 11. In contrast, if core 7 attempts to steal work from core 37 and succeeds, it will update entries 36 to 41 of its own system view to the value *Busy*; this update reflects the success of the steal attempt (i.e. presence of work to be stolen), at the same time speculating the presence of work in the entire neighborhood of the victim core, as it belongs to a remote region.

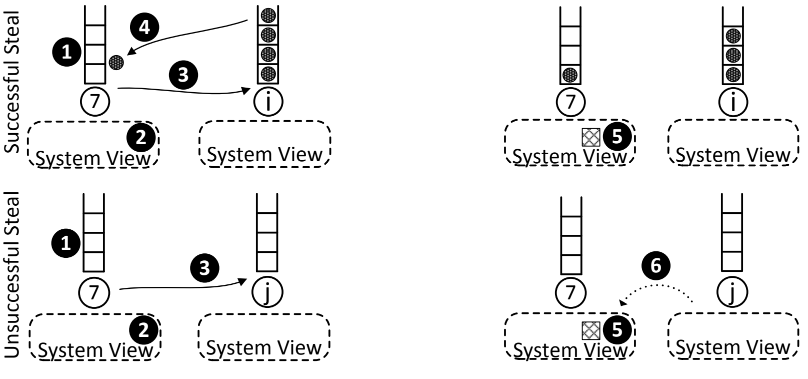
4.3.2.2 Using Other Cores' System Views

In the second SWAS variant, cores are also allowed to consult other cores' system views, in order to enrich their own.

Obtaining information in this manner happens whenever an idle core performs an unsuccessful steal attempt. At this time, the idle core reads the victim core's system view. Any entry which has the value *Don't know* in the idle core's bit vector, is set to the value of the victim core's bit vector. As such, some *Don't know* values of the idle core's system view are changed to a meaningful value (*Busy* or *Not busy*). This happens only on unsuccessful steal attempts, to increase the chances of success of the next steal attempt. When opting to also do this in the case of successful attempts, we observed a considerable performance penalty. It was thus concluded that on a successful steal attempt, it is more important that the idle core gets back to work as soon as possible, rather than spend time to enrich its system view.

4.3.2.3 Information Staleness

Information that currently is stored in a core's system view can be out-of-date if not renewed. In SWAS, we consider this staleness issue as follows: Each core keeps track of windows of time during program execution. These windows are not of a fixed duration, but are defined relative to the successful steal attempts performed by the core: a parameter m is defined, and whenever m successful steal attempts are made, one time window has elapsed. For each entry of the bit vector, an extra "update" bit is kept, which is reset at the beginning of the time window. Whenever a bit vector entry is updated the corresponding



(a) Before (b) After

Figure 4.2: The idle core and victim core (and their queues) before (a) and after (b) a successful (top) or unsuccessful (bottom) steal attempt in SWAS. Core 7 becomes idle ❶ and uses its current system view ❷ to choose a victim core to steal work from. It chooses core i (top) or j (bottom) and checks its queue ❸. If there is work to steal, one task is taken to be executed on core 7 ❹. Regardless of the steal attempt outcome, the system view of core 7 is updated to reflect said outcome ❺. In case of an unsuccessful steal, if consulting other cores' system views is allowed, this also takes place at this time ❻.

"update" bit is set. Any entries that have not been updated during an entire window, are set back to the value *Don't know* when the window ends.

4.3.3 Work Stealing Decisions

The decision making process, which is the same in both SWAS variants, is outlined in the pseudo-code of Algorithm 1. The integrated process of stealing work and updating system views is shown in Figure 4.2. The top half shows a successful steal and the bottom half an unsuccessful one. Part (a) on the left shows the idle and victim cores before the steal attempt and part (b) on the right shows the same cores after the attempt. We will use numbers ❶ to ❻ in the following description to refer to the individual steps.

When a core executes no workload and has no new tasks in its queue ❶, this signifies a *core idle* event for this core, thus it will attempt to steal work from a victim core. To choose a victim core, the idle core uses its system view ❷. First, it chooses a target region by comparing the *Region Busy Estimators* (lines 1 to 7 of Algorithm 1). In the general case, the steal attempt will be directed to the system region with the highest estimator, subject to the following constraints:

- Ties are resolved in favor of the region with the lowest index, that is, the set of cores that are closer to the idle core. This condition strengthens the impact of system hierarchy on the victim core selection process. If different regions are similarly loaded, the closest one should be chosen. Furthermore, during initialization, all bit

Algorithm 1: *Choosing the Victim Core*

```
1: //choosing target region
2: if all $est_i \leq$ threshold then
3:     Choose region $R_{max}$
4: else
5:     Choose region with higher estimator variable
6:     Break ties in favor of lower-indexed region
7: end if
8: //Creating random draw between cores of the selected region
9: int tickets = 0
10: for Each core $l$ in the selected region do
11:     Read bit vector entry for this core
12:     if entry is Busy then
13:         Add two tickets in draw for core $l$
14:         tickets $+= 2$
15:     else
16:         if entry is Don't know then
17:             Add one ticket in draw for core $l$
18:             tickets $+= 1$
19:         end if
20:     end if
21: end for
22: //choose random ticket from draw
23: select (random number $mod$ tickets)
24: //(if no tickets in draw, select random core from region $R_{max}$)
25: return core corresponding to selected ticket as victim
```

vector entries are set to *Don't know*, resulting to all estimators being initialized to the same value (0.5 according to Equation 4.1). Thus, tie resolution is important, because it determines the initial direction the system takes.

- There is an exception to the comparison rule: if all estimators are below a certain threshold, the region with the highest index (region R_{max}, corresponding to the set of remote cores) is chosen. This choice is motivated as follows: low estimators imply sparsity of workload across all regions. The priority that is otherwise given to the closest region, together with the fact that remote regions are larger and less uniform, means that the idle core's system view contains less complete information about region R_{max} than other regions. Thus, in these cases of a sparsely loaded system, remote steals are attempted in an effort to find a busy neighborhood which has not been visited recently. The threshold value was tuned for our experimental platform and set to the same value for all experiments presented in Section 4.5.

After a choice of region is made using the region busy estimators, the bit vector is used to select a particular core. To this end, the idle core creates a random draw between cores of the selected region, as follows (lines 8 to 21 of Algorithm 1):

- Each core marked as *Busy* in the bit vector is entered in the draw twice. Thereby, *Busy* cores have a higher chance to be selected, because they are more likely to provide a successful steal.

- Each core marked as *Don't know* in the bit vector is entered in the draw once. Such cores have a lower chance to provide a successful steal, but trying them enriches the system view of the idle core.

- Cores marked as *Not busy* in the bit vector do not enter the draw. These cores probably can not provide a successful steal, and there is no need to learn something about them at this point. Note that *Not busy* entries will return to the *Don't know* value, when this information becomes stale, else a *Not busy* value could never change.

The draw is performed and the winner becomes the steal victim (lines 23-25). In the very rare case that no tickets are entered in the draw, a random core from region R_{max} is selected.

The idle core then checks the queue of the selected victim core for work to be stolen ❸ and in case such exists, it steals one task to execute ❹. Regardless of the steal attempt result, the idle core also updates its system view accordingly ❺. Finally, if the SWAS variant used allows consulting other cores' system views, this also takes place at this time, in case of an unsuccessful steal ❻.

This concludes our description of SWAS. The evaluation of the resulting implementation is presented in Section 4.5. Before this, Section 4.4 describes the components of our experimental setup, namely the machine and the runtime on which experiments are run, the work stealing strategies we use as comparison points and the various workloads for which we evaluate SWAS.

4.4 Experimental Setup

In this section we outline our experimental setup, consisting of the following five elements: The experimental platform (Section 4.4.1), the GO:TAO runtime system (Section 4.4.2), the baseline work stealing strategies we implemented for comparison (Section 4.4.3), the representative workloads used to evaluate all strategies (Section 4.4.4) and some relevant experimental parameters (Section 4.4.5).

4.4.1 The Experimental Platform

The experimental platform we use is a Dell PowerEdge R815 server consisting of four AMD Opteron 6348, each with two NUMA nodes and six cores per NUMA node, for a total of 8 NUMA nodes and 48 cores. Each NUMA node contains a three-level cache hierarchy. The upper level consists of 16KB L1 data cache, one per core. A pair of cores shares an 2MB L2 cache. Finally, the six cores in a NUMA node share a 8MB L3 cache, of which 2MB store the HT Assist Probe Filter and 6MB are usable by applications. The total system therefore provides 48MB of L2 cache and 48MB of L3 cache to applications. The test system runs Ubuntu Linux 14.04. All applications have been compiled using GCC v4.8 with optimization level -O3.

4.4.2 The GO:TAO runtime

The execution environment used in our experiments is GO:TAO, a DAG-based runtime implementation targeting high parallelism, good locality and low overheads [97]. The basic unit of execution in GO:TAO is moldable tasks that can make use of multiple cores. This moldability allows the GO:TAO runtime to match hardware resources with software requirements. This feature improves cache effectiveness and can also be used to utilize idle cores [102] and to deal with composability of parallel runtimes [103]. When restricted to single-core tasks, GO:TAO is representative of other task-based runtimes, such as Cilk [104], TBB [105] or OMPSs [106]. Hence, the performance of SWAS can be directly extrapolated to such runtimes.

To support moldable tasks, GO:TAO extends the task abstraction with a task group, a set of resources (number of cores, caches) and an embedded scheduler. The resulting Task Assembly Object (TAO) is itself a parallel computation over a set of cores. This feature can be exploited by SWAS: whenever a TAO is mapped on a set of cores, these cores obtain implicitly the information of which other cores are busy executing the same TAO, thus enriching their system view.

To evaluate SWAS comprehensively and fairly, we conduct two sets of experiments:

- *Single-core TAOs*: In this configuration, GO:TAO is equivalent to traditional task-based runtimes (Cilk, TBB, etc.). We call this set of experiments **Standard-RT**. Their purpose is to showcase the performance of SWAS in a conventional setting.

- *Parallel TAOs*: In this configuration, GO:TAO assigns multiple cores to individual TAOs by matching the cache and core requirements of the TAO. We call this set of experiments **TAO-RT**. Their purpose is to highlight the added benefit SWAS can provide in the context of hierarchical runtimes.

4.4.3 Work stealing strategies

We have implemented different baseline work stealing strategies, that address the distinct relevant factors, as presented in Section 4.3:

No Work Stealing (NoWS). In this strategy the tasks are executed where they are initially assigned for execution.

Random Work Stealing (WS_Rnd). In this strategy idle cores attempt to steal work from any core in the system using a random selection algorithm [40].

Local Work Stealing (WS_Loc). In this strategy idle cores attempt to steal work only from cores belonging to the same closest memory hierarchy domain, *i.e.*, as described in Section 4.3, the cores belonging to the R_1 set. Core selection within R_1 is random. This strategy focuses on preserving data locality, like the approaches presented in [41] and [45].

Hierarchical Work Stealing (WS_Hie). In this strategy idle cores first attempt to steal work from the cores belonging to the closest memory hierarchy domain (cores in the R_1 set). If this attempt fails, then they attempt the next regions in order. Core selection within a region is random. This strategy benefits from knowledge of system hierarchy, like the ones presented in [42] and [43].

Work Stealing using Global system view (WS_Global). In this strategy, one global system view is shared by all cores, updated by all cores and used for all stealing decisions. This system view is not protected by locks, so when updated by multiple cores there might be some inconsistencies. This strategy aims to demonstrate the effect of keeping information which is more complete than SWAS, at the cost of extra coherence maintenance overhead and increased memory traffic.

SWAS based on Attempts only (SWAS_Att). This is the SWAS variant in which cores update their system view based only on steal attempt outcomes.

SWAS based also on Other cores' system views (SWAS_Others). This is the SWAS variant in which cores are allowed to consult other cores' system views on an unsuccessful steal attempt.

4.4.4 Workloads

We evaluate all strategies on a range of workloads, which constitute a good representation of some real-life scenarios. We deem that a work stealing strategy should be safe to use on a system which expects workloads with different balance characteristics and improve performance when load balancing is needed, while not hurting it during the rest of the time.

Balanced Workload: We use a perfectly partitioned Sort based on a reduction pattern with Quicksort tasks in the first level and Mergesort calls in the lower levels. The Quicksort tasks are distributed uniformly among all system cores, to achieve perfect load balance and minimize communication. The balanced Sort is an example of a program the performance of which can be hurt by work stealing. An efficient work stealing algorithm should avoid this.

Workload with Jitter: In this scenario, although the application itself is balanced, a system event triggers an interfering task such as a background daemon or the processing of an interrupt on one of the cores. We used the same implementation of Sort as above and inserted the jitter as a large, unstealable task on one of the system cores. The size of this unstealable task was determined to be roughly 50% of the baseline execution time of the balanced version. This application aims to demonstrate how different strategies are able to absorb local transient fluctuations of the workload.

Unevenly Distributed Workload: In this case, the Sort application is used and the Quicksort tasks in the upper level are distributed unevenly on the system cores. More specifically, the whole workload is assigned to one half of the system, making the other half initially idle. This can happen, for example, if the input data set is located on a subset of the available nodes. In this case, strategies are evaluated for their ability to detect and remedy the uneven placement of tasks.

Imbalanced Workload with Dynamic DAG: The Unbalanced Tree Search (UTS) is a synthetic benchmark that constructs a geometric tree such that each node has a probability to spawn a number of children nodes. We use two different flavors: each node having a 5% probability to spawn 20 child nodes (UTS_5_20) and each node having 33% probability to spawn 3 child nodes (UTS_33_3). As the nodes of the tree are created in a manner that is both imbalanced and statistical (thus unpredictable), UTS is a very interesting workload for evaluating dynamic load balancing strategies.

Stencil Computation Workload: We use a heat diffusion stencil application, which is initially perfectly partitioned among all system cores. Stencil computations have complex task dependencies, since each task in the DAG has multiple parent tasks. Due to this, a parent task that completes slightly later than others, can spawn multiple tasks on the core it executes, while other cores remain idle. We use two different flavors: 200 iterations on a 3072x3072 matrix (HEAT_3072) and 500 iterations on a 1536x1536 matrix (HEAT_1536).

4.4.5 Experimental Parameters

For all flavors of the Sort application, we use three different sizes of the input array, 2^{25}, 2^{26} and 2^{27} elements. We use the same task graph for all three input sizes, thus varying the size of the data set of each task. Also, for UTS workloads, we modified the execution time of each task in the same manner, by setting the "compute granularity" parameter of the benchmark to the values 1, 2 and 4. For each combination of workload and work stealing strategy, we execute 100 runs and calculate mean execution time and standard deviation. The results of this process are presented in Section 4.5.

4.5 Experimental Results

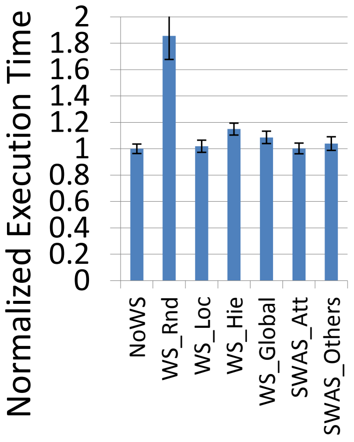
In this section we present the results of the experimental evaluation of SWAS. We begin with performance results of all strategies over all workloads for both runtimes (Section 4.5.1). Subsequently, we offer some additional insight, by presenting the number of successful steals of each strategy broken down by region of the victim (Section 4.5.2), as well as a breakdown of indicative time overheads of the different strategies (Section 4.5.3).

4.5.1 Performance Evaluation

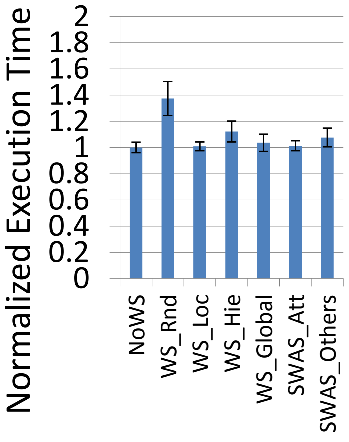
The performance results of all implemented work stealing strategies on all tested workloads for both runtimes, are measured in terms of execution time normalized to the baseline. For each workload and runtime, we visualize normalized performance of each strategy as vertical blue bars and the standard deviation as error margins at the top of the blue bars.

4.5.1.1 Balanced Workload

An evenly partitioned static workload with data locality-aware placement, such as the case of the balanced Sort, does not benefit from work stealing. Thus, we are mainly interested in not penalizing performance compared to execution without work stealing. For this purpose, the performance results shown in Figure 4.3 are normalized to NoWS, as opposed to all other workloads, for which they are normalized to random work stealing (WS_Rnd).



(a) Balanced, Standard-RT



(b) Balanced, TAO-RT

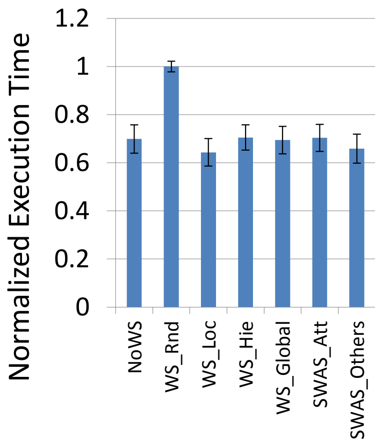
Figure 4.3: Execution time normalized to no work stealing (NoWS) and its standard deviation for different strategies in the balanced workload case, on the Standard- and TAO-RT.

Performance results for this workload are shown in Figure 4.3. The relative performance of strategies is similar among the two runtimes. Random stealing (WS_Rnd) incurs the largest penalty (85.6% for Standard-RT and 37.4% for TAO-RT). Additionally, it has the largest standard deviation between runs, making it both inefficient and inconsistent. WS_Hie has the second largest penalties (14.9% and 12.2%). WS_Global entails a penalty of 8.5% on the Standard-RT and 3.6% on TAO-RT. SWAS_Others' penalties are also relatively low, 3.8% and 7.6%. WS_Loc is the most conservative of all baseline strategies, since it does not disrupt data locality. As such, it entails a low penalty of 0.9% to 1.8%. SWAS_Att has the lowest penalties of all strategies (0.3% and 1.4%).

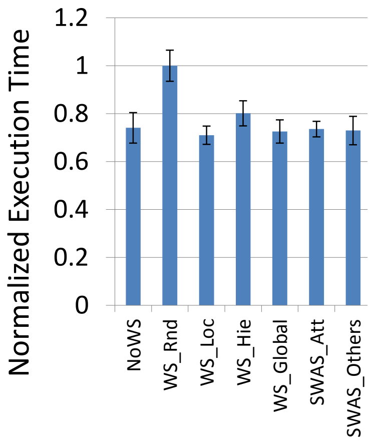
4.5.1.2 Workload with Jitter

For this as well as all subsequent workloads, we evaluate the performance of strategies compared to the random work stealing (WS_Rnd) baseline. Results for this workload are plotted in Figure 4.4.

This workload simulates a transient, local imbalance, which is best resolved by local steals. As a result, WS_Loc yields the largest improvement over WS_Rnd (35.7% and 29% on the two runtimes). Also, WS_Rnd has similar impact on performance as it did on the balanced workload (NoWS is 26%-30% faster), since it disrupts locality all over the system. Both SWAS variants are very close to WS_Loc, providing improvements of 26.4% to 34.1%

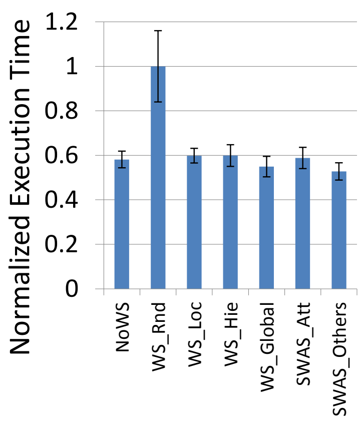


(a) Jitter, Standard-RT

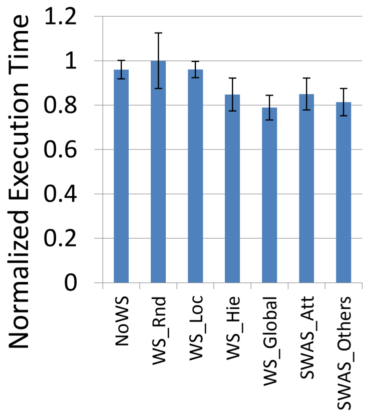


(b) Jitter, TAO-RT

Figure 4.4: Execution time normalized to random work stealing (WS_Rnd) for the balanced workload with jitter.



(a) Uneven, Standard-RT



(b) Uneven, TAO-RT

Figure 4.5: Execution time normalized to random work stealing (WS_Rnd) for the unevenly distributed workload.

4.5.1.3 Unevenly Distributed Workload

In this case, the balanced application workload is distributed on half of the system cores. The purpose of work stealing strategies is to let the idle half share the workload as efficiently as possible. Results for the uneven workload are shown in Figure 4.5.

We observe that on the standard-RT, WS_Rnd still hurts performance, as the disruption of locality it entails is more important than the workload balancing. NoWS is thus 42% more efficient. Furthermore, WS_Loc has, as expected, similar performance to NoWS, as remote steals are not allowed, so the uneven workload cannot be distributed to the whole system. SWAS_Att also yields 41% improvement. The best strategies are WS_Global and SWAS_Others (45% and 47% improvement respectively).

Results are different on TAO-RT. GO:TAO's cache-aware management of resources reduces cache misses and makes execution less sensitive to data locality [97]. Therefore, WS_Rnd now does not penalize performance as much: NoWS and WS_Loc are now only 4% better. Furthermore, the remaining four strategies are faster than NoWS: WS_Hie and SWAS_Att yield around 15%, while WS_Global and SWAS_Others are the best with around 20%.

4.5.1.4 Imbalanced Workload with Dynamic DAG

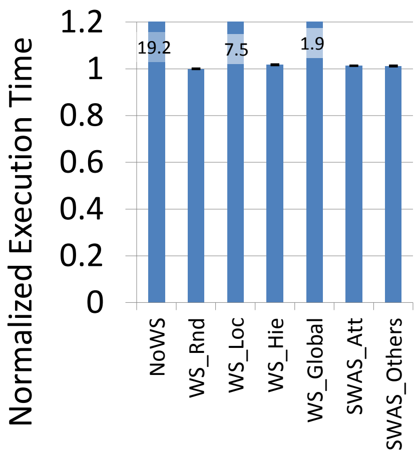
The case of UTS is important, both because it is heavily imbalanced by design and because its tasks are spawned at runtime, making a dynamic load balancing strategy necessary. Most subtrees of the dynamically constructed tree terminate very quickly, with only a low percentage of them going very deep, overloading the respective system cores. As can be observed in Figure 4.6, when work stealing is disabled, execution time is 18 to 22 times longer. Local stealing suffers from the same problem to a lesser degree, resulting in 7.0 - 7.9 times longer execution time than WS_Rnd.

Furthermore, making the system view global penalizes performance significantly, resulting in an execution time 1.9-2.7 times longer than WS_Rnd, on account of the increased memory traffic and the overhead of coherence maintenance for the global system view. This was verified by scaling up the task granularity: doubling the task size reduces the penalty of WS_Global by 25%. The finer the task granularity is, the more frequent the steal attempts become. Constant updates of the global system view by all cores result to the observed performance penalty. This effect becomes weaker as the tasks become coarser and system view updates are sparser.

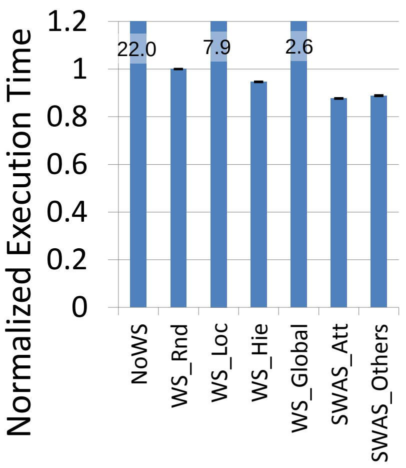
Comparing the rest of the strategies, we observe that on the Standard-RT they all achieve identical performance. On the TAO-RT, however, the two SWAS variants are significantly better, achieving 12.2% and 11.1% performance improvement over WS_Rnd for UTS_5_20 and even more than that (18.5% and 17.2%) for UTS_33_3.

4.5.1.5 Stencil Computation Workload

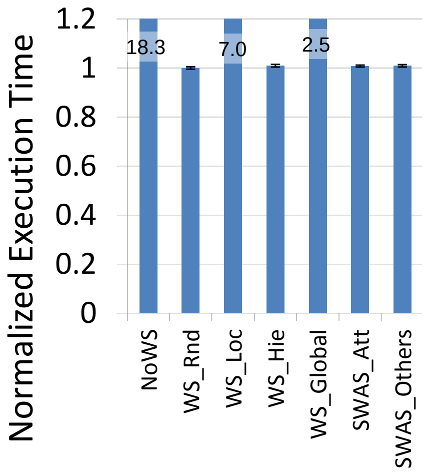
The heat diffusion stencil computation suffers from unbalancing at runtime, because of the complex nearest-neighbor dependencies in its DAG. The slowest system regions tend



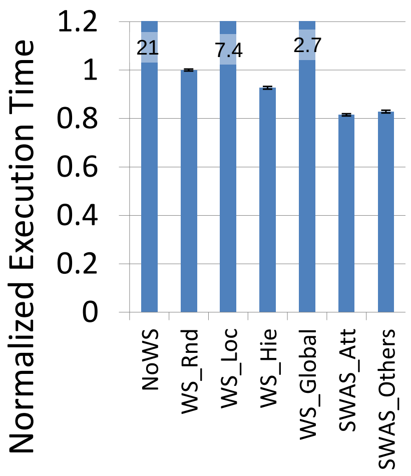
(a) UTS_5_20, Standard-RT



(b) UTS_5_20, TAO-RT

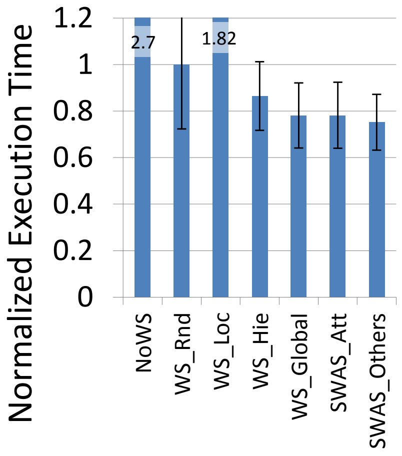


(c) UTS_33_3, Standard-RT

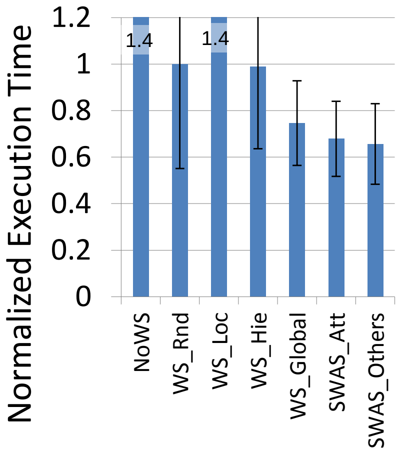


(d) UTS_33_3, TAO-RT

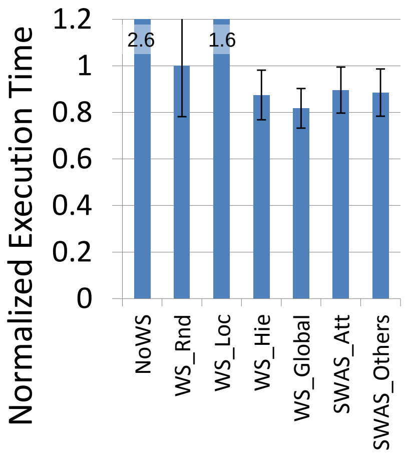
Figure 4.6: Execution time normalized to random work stealing (WS_Rnd) for both variations of the DAG-based workload (UTS).



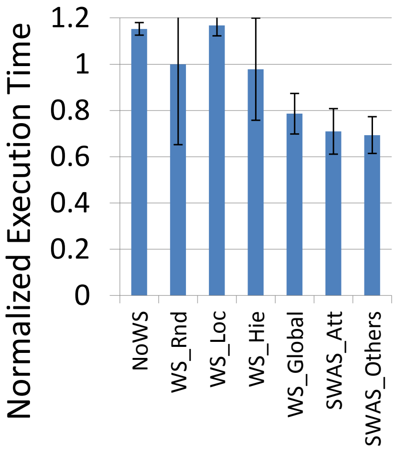
(a) HEAT_3072, standard-RT



(b) HEAT_3072, TAO-RT



(c) HEAT_1536, standard-RT



(d) HEAT_1536, TAO-RT

Figure 4.7: Execution time normalized to random work stealing (WS_Rnd) for both variations of the heat distribution stencil workload.

to pull tasks from neighboring regions from one iteration to the next, an effect that can be observed on Figure 4.7 as a slowdown of 2.5 - 2.7 times on Standard-RT and 15% to 40% on TAO-RT, compared to WS_Rnd. WS_Loc does only slightly better than that.

On the Standard-RT, WS_Hie gains 12.5% - 13.6%, but SWAS variants and WS_Global produce the highest benefits (18.3% - 21.9% for WS_Global and 10.4% - 24.8% for SWAS). On the TAO-RT, the SWAS variants do even better than other strategies (29.0%-34.4% improvement over WS_Rnd), while WS_Global only gains up to 25.5%. Furthermore, SWAS performance results also have much lower standard deviation than WS_Rnd and WS_Hie, meaning that its benefits are more consistent between runs.

To sum up all presented performance results, Table 4.1 indicates the best strategy for each workload and lists the performance penalties of every other strategy compared to the best. It can be observed that the two SWAS variants and WS_Hie are the only ones that never punish performance severely. Of these strategies, SWAS variants are often the best and otherwise very close to the best. Specifically, SWAS_Att is the best strategy for two workloads and has a maximum penalty of 11.5% and SWAS_Others is the best strategy for four workloads and its maximum penalty is 8.3%, with most of its relative penalties below 4%. This result demonstrates that SWAS is the best option for a machine without in advance knowledge of its workload.

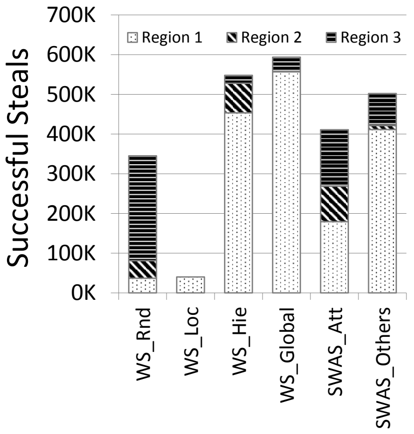
4.5.2 Successful Steals per Region

The various work stealing strategies differ in the total number of successful steals they perform, as well as the distribution of selected victims. Figure 4.8 illustrates total successful steals for all strategies, categorized by region of the victim core, for the UTS_5_20 and HEAT_1536 workloads, on both Standard-RT and TAO-RT.

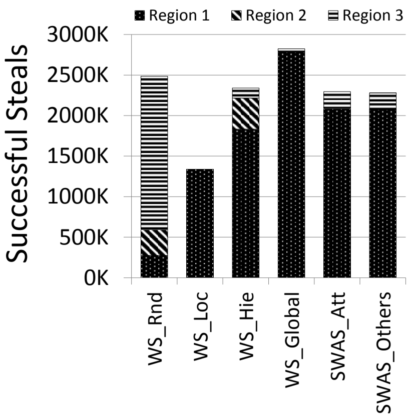
For UTS, we observe that local steals are in general much more common. This happens because every task has a chance of spawning 20 children tasks on the same core, creating local imbalances. However, the spawned children have, in turn, a probability to spawn even more work. If the potential task-spawners are not re-distributed, system-wide imbalances occur. This explains why WS_Loc has both the worst performance and the fewest steals. For the same reason, the strategies that perform better all perform a significant number of remote steals. This results to more neighborhoods of the system being active, which in turn triggers more total local steals. WS_Rnd is special in that it performs many more remote steals than local ones. In fact, the distribution of steals among the three regions, directly follows the distribution of cores, as a result of the random victim core selection. As UTS is compute-intensive and has a small working set, WS_Rnd is not severely punished for ignoring system hierarchy and application data locality - being on par with SWAS on the standard RT and 14% worse on the TAO-RT. Note that the vertical axis scales on parts (a) and (b) of Figure 4.8 are very different, thus we compare only the relative trends among different strategies, not the total steals among the two runtimes. The higher number of steals on TAO-RT is related to the small task granularity in UTS. Distributing tasks across cores, as done in TAO-RT, results in

Table 4.1: Performance penalties of all strategies compared to the best strategy for each workload.

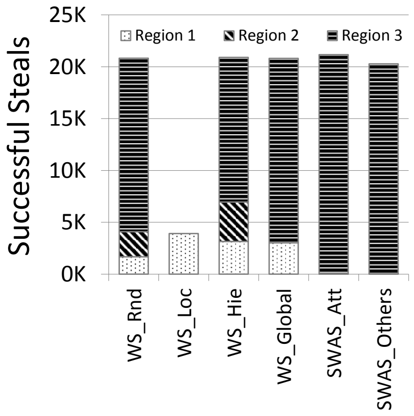
	NoWS	WS_Rnd	WS_Loc	WS_Hie	WS_Global	SWAS_Att	SWAS_Others
Balanced, Standard-RT	**BEST**	85.6%	1.8%	14.9%	8.6%	0.3%	3.9%
Jitter, Standard-RT	8.7%	55.5%	**BEST**	9.6%	8.0%	9.4%	2.4%
Uneven, Standard-RT	10.2%	89.6%	13.4%	13.6%	4.1%	11.5%	**BEST**
UTS_5_20, Standard-RT	1820%	**BEST**	653%	1.7%	93.6%	1.3%	1.2%
UTS_33_3, Standard-RT	1730%	**BEST**	602%	0.9%	147%	0.8%	0.9%
HEAT_3072, Standard-RT	256%	33.0%	142%	15.0%	3.9%	4.0%	**BEST**
HEAT_1536, Standard-RT	212%	22.4%	101%	7.0%	**BEST**	9.6%	8.3%
Balanced, TAO-RT	**BEST**	37.4%	0.9%	12.2%	3.6%	1.4%	7.7%
Jitter, TAO-RT	4.3%	40.8%	**BEST**	12.9%	2.2%	3.6%	2.7%
Uneven, TAO-RT	21.7%	26.7%	21.7%	7.5%	**BEST**	7.7%	3.1%
UTS_5_20, TAO-RT	2410%	14.0%	799%	8.0%	193%	**BEST**	1.3%
UTS_33_3, TAO-RT	2480%	22.7%	808%	13.7%	231%	**BEST**	1.7%
HEAT_3072, TAO-RT	109%	52.3%	107%	50.5%	13.7%	3.4%	**BEST**
HEAT_1536, TAO-RT	66.2%	44.2%	68.3%	41.0%	13.3%	2.4%	**BEST**
Maximum penalty to BEST	2480%	89.6%	808%	50.5%	231%	11.5%	8.3%
Average penalty to BEST	652%	37.4%	237%	14.9%	51.6%	4.0%	2.4%



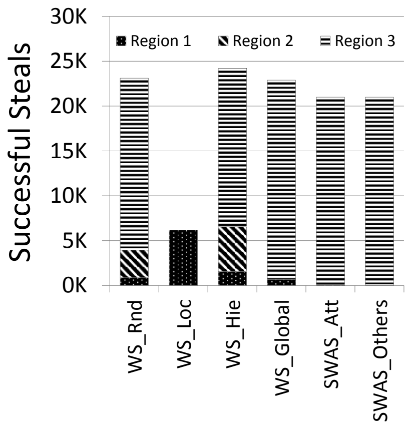
(a) UTS_5_20, standard-RT



(b) UTS_5_20, TAO-RT



(c) HEAT_1536, standard-RT



(d) HEAT_1536, TAO-RT

Figure 4.8: Number of steals by region of the victim core for UTS_5_20 (a and b) and HEAT_1536 (c and d).

Table 4.2: Average overheads of work stealing strategies per steal attempt in μsec.

	NoWS	WS_Rnd	WS_Loc	WS_Hie	WS_Global	SWAS_Att	SWAS_Others
Victim selection	N/A	0.11	0.16	0.25	3.67	0.38	0.40
System view update	N/A	N/A	N/A	N/A	2.74	0.18	0.21
Other system view consultation (after failed steal attempts only)	N/A	N/A	N/A	N/A	N/A	N/A	0.88
Stale information reset (occasional)	N/A	N/A	N/A	N/A	N/A	0.71	1.58
Total:	N/A	0.11	0.16	0.25	6.41	1.27	3.07

non-negligible scheduling overheads. As execution time increases, tasks spend more time waiting in the work queues, which increases their probability to be stolen.

For the heat diffusion stencil workload, steals from Region-3 cores dominate all strategies (except WS_Loc which cannot perform such steals). In this case, the slowest cores tend to spawn more tasks for themselves, because of the multiple output (i.e., nearest neighbor) task dependencies. Thus, from one iteration to the next, some neighborhoods absorb workload from around them. To mitigate this effect, remote steals are needed. Lacking system load information, WS_Hie always tries to steal locally first, resulting in fewer remote steals, whereas the two SWAS variants perform almost 100% remote steals. This is the reason that SWAS performs equally (on the standard-RT) or better (on the TAO-RT) than hierarchical work stealing, even though the total number of successful steals it achieves is lower.

4.5.3 Breakdown of Overheads

All work stealing strategies share some common overheads: the cost of querying the victim core for work, and in case of a successful steal, the cost of migrating the task and moving any associated data.

On top of the above, the strategies we have evaluated have different additional overheads associated with victim selection and possible maintenance of a system view. These overheads, per one steal attempt, are summarized in Table 4.2.

We observe that the performance gains of SWAS_Att are a consequence of the very low extra cost of making much more educated decisions than baseline strategies. The added overhead of the complex victim selection process and system view update is at most 3.07 μsec per attempt, including the process of resetting stale entries in the bit vectors, which happens only occasionally. Overheads of SWAS_Others are slightly higher, because the system view cannot be kept strictly private (occasionally other cores need to access it).

The global nature of the system view in WS_Global is the reason that victim selection and system view update are much slower in this case (3.67 and 2.74 μsec on average per attempt). Note that this overhead is expected to increase with system size, making WS_Global not scalable.

4.6 Chapter Summary

Work stealing strategies are expected to efficiently redistribute an imbalanced workload on systems with increasing amount of cores and deepening memory hierarchies and topologies. Furthermore, the workloads that can run on a system during different periods are often not known in advance. They can be either balanced or imbalanced, with static or dynamically generated task graphs, or any combination of the above. Additionally, they are, in all cases, prone to additional unbalancing due to external interference (jitter) or sub-optimal data and task co-location.

In this chapter we present a novel work stealing strategy to face the above challenges. The approximate information stored in the system view, which is used to guide stealing decisions, is maintained by each core using already necessary operations such as steal attempts and stored in compact data structures, enhancing the efficiency of victim core selection for a very low time cost.

By prioritizing efficient steals within a core's neighborhood, SWAS can mitigate local imbalances. At the same time, using the approximate system views, it can also detect system-wide imbalances and visit remote regions when needed. This combination allows it to achieve up to 18.5% improvement compared to a random stealing baseline for a workload with imbalanced, dynamically generated DAG and up to 34.4% for a stencil computation with nearest neighbor dependencies. Equally importantly, it does not penalize balanced workload performance and deals as well as baseline strategies with transient local imbalances (jitter) and sub-optimal data-task co-location. This combination of results, allows use of SWAS without a priori knowledge of a multiprocessor system's workload.

5
Thesis Summary and Conclusions

In this thesis we have suggested that runtime management can improve multicore systems' efficiency, by limiting the negative effects of unpredictable events. To support this claim, we have presented studies for three distinct optimization objectives: graceful degradation, energy efficiency and performance - the latter coming as a result of load balancing. For optimizing each of the above objectives, we detect existing flexibility and expose it to a runtime manager, which in turn exploits it to improve system efficiency. While this flexibility provides optimization potential, it also complicates the relevant decisions. Thus we show, in each study, that absolute optimality is not feasible at runtime, as a result of (i) the size of the solution space and (ii) the unavailability of complete and precise information with respect to critical parameters of the optimization.

This chapter summarizes and concludes the thesis. Section 5.1 provides a summary of chapters 2, 3 and 4, that contain the three studies. Section 5.2 outlines the main contributions of each study. Lastly, Section 5.3 proposes some possible extensions to the presented work.

5.1 Summary

The first study, which is presented in Chapter 2, is on graceful degradation of fault-prone multicores with adaptability coming from reconfigurable hardware, heterogeneity and workload flexibility. We start Chapter 2 by detecting the aforementioned adaptability, suggesting a high level model for such an adaptive system and, based on this, define its configuration space. We go on to quantify *system efficiency*, based on the performance, energy and functionality of the system in a given configuration. Subsequently, we propose different algorithms that react to *component failure* events, attempting to optimize system efficiency: custom heuristics based on incremental and precomputed solutions, as well as simulated annealing and a genetic algorithm. We compare the different algorithms in terms of solution quality and execution time, through both simulations and emulations on the Intel *Single-Chip Cloud* machine [77], with two different workloads (with and without task dependencies). We conclude the chapter with a methodology for characterizing degradable hardware components, in terms of performance and energy efficiency of their degraded configurations. The characterization process provides our high level adaptive system model with annotations required by the runtime manager. Additionally, it also allows design space exploration of mixed-grain reconfigurable processor arrays with respect to the reconfiguration granularity and the fault density.

Chapter 3 contains our study on runtime optimizations for energy efficiency. We focus on single-ISA heterogeneous systems that offer adaptability in the form of different kinds of cores (faster or more power efficient) as well as Dynamic Voltage and Frequency Scaling (DVFS). We utilize this adaptability, to react to *application spawn* and *application termination* events, with the purpose of maximizing energy efficiency in terms of IPS/Watt. We first describe our prediction model, that the runtime manager uses to evaluate potential configurations before applying them, in order to select the best one. The model predicts applications' performance based on profiling information of each standalone application, as well as a method to project this information to the dynamic, multi-application current state of the system. Additionally, it predicts system power using a linear power model based on the predicted performance numbers, together with a set of static, platform-specific parameters. We go on to propose three different heuristics that use the prediction model to perform partial search of the configuration space and choose the best configuration as a response to every event. Subsequently, the three algorithms are evaluated, both exhaustively for a small number of representative events and through a large number of dynamic, unpredictable, multi-application scenarios. We compare our algorithms to the *powersave, interactive* and *ondemand* governors and also explore the scaling of their execution time with respect to the number of cores in the system.

Lastly, in Chapter 4 we study load balancing by means of *Work Stealing*, with the objective of optimizing performance. In this case, we perform runtime management in a distributed manner, as each core becoming idle has to choose a victim core to steal work from, with incomplete and approximate information of the current workload distribution over the system resources. We begin the description of our approach by defining the system view: a hierarchical structure that every core maintains, denoting which other cores or regions of the system are more likely to be loaded with work to be stolen. Subsequently, we outline the process the core follows when it becomes idle, to select a victim core to steal work from, using the system view, while also enriching the information in it. Our approach is evaluated with diverse workloads: perfectly balanced, jittery, unevenly distributed, dynamically generated imbalanced DAGs and a stencil application with complex nearest-neighbor dependencies. We compare the performance of our approach with existing techniques: random, hierarchical and local work stealing, as well as stealing guided by complete system load information. We conclude the chapter by laying out the time overheads incurred by each strategy.

The following section summarizes the contributions of this thesis for the three aforementioned parts.

5.2 Contributions

This thesis studies three different runtime optimization problems. This section summarizes the main contributions and findings of each study.

5.2.1 Runtime Management for Graceful Degradation

The first part of the thesis deals with runtime management of adaptable systems for graceful degradation. The main contributions of this study are the following:

- **Formulating adaptive systems' graceful degradation as an optimization problem.**

  We proposed a high level model of an adaptive multicore, based on mutually exclusive versions of its hardware components and its workload tasks. The versions are annotated with performance, energy and functionality numbers that allow us to quantify system efficiency and use it as an objective function for our optimization algorithms.

- **Proposing alternative algorithms for optimizing system efficiency, utilizing incremental and partial precomputed solutions.**

  We proposed multiple runtime algorithms that are triggered by *component failure* events and choose a new system configuration optimizing system efficiency. We developed custom fast heuristics, as well as versions of simulated annealing and genetic algorithm, that are tailored to the particular optimization problem.

- **Characterizing degradable hardware components.**

  We developed a methodology for characterizing, in terms of performance and energy efficiency, the degraded versions of hardware components that facilitate fault isolation by means of reconfiguration. This process produces the necessary annotations our high level system description needs. Additionally, for the case of processor arrays with mixed-grain reconfiguration, it resulted to a design space exploration dictating the optimal reconfiguration granularity for any given fault density.

Our results show that the custom heuristics are one to two orders of magnitude faster than simulated annealing and genetic algorithm, while maintaining similar system efficiency in case the reconfiguration cost is unconstrained. Specifically, they provide better efficiency for at least the first 20% of system lifetime and are at most 13% worse towards the end, compared to simulated annealing and genetic algorithm. Performing the same evaluation under reconfiguration cost constraints, the custom heuristics are often found unable to satisfy them. As custom heuristics are much faster, we propose a combination of algorithms for this constrained case: run a fast heuristic first and back it up with a slower genetic algorithm if it fails. The above strategy is also backed up by another finding: our best algorithm always provides efficiency less than 15% worse, compared to an exhaustive, optimal solver.

Focusing on the design space exploration for mixed-grain reconfigurable processor arrays, we make the following observations for different fault densities: Non-reconfigurable arrays offer more cumulative performance for fault densities of up to four faults in the area of nine non-reconfigurable processors. For higher densities (five or more faults in the same area) a coarse-grain array with reconfiguration granularity of eight substitutable units per processor offers 10% better performance. Lastly, for even higher densities of at least 13 faults in the same area, a mixed-grain reconfigurable array with granularity of 16 substitutable units per processor is found to be 8% faster than other design points.

5.2.2 Runtime Management for Energy Efficiency

In the second part of the thesis, we focus on designing a runtime manager for optimizing energy efficiency on heterogeneous, DVFS-capable multiprocessors. The main contributions of this part are the following.

- **Predicting performance with scalable profiling and runtime projection.**

  We proposed a feasible and accurate performance prediction strategy. We built the strategy on two components: First, a scalable process for profiling individual applications and extracting their performance characteristics with a number of runs linear with respect to the system size. Second, a runtime operation, that projects the offline-acquired, static, application-specific measurements to the dynamic, multi-application status of the system, monitored online.

- **Predicting power with a model that does not require retraining.**

  We calibrated a power prediction model that uses as inputs the performance predictions produced as described in the previous point, and a set of static, system-specific power parameters. As application-specific parameters are not involved, the model can be used for any application mix without offline or online retraining.

- **Developing heuristics for runtime management.**

  We developed three different runtime heuristics that use the aforementioned models and online monitoring, to decide application placements and cluster frequencies whenever an *application spawn* or *application termination* event happens. The heuristics perform limited search of the configuration space and choose the configuration that is expected to maximize energy efficiency while satisfying performance requirements of individual applications.

Through exhaustive evaluation of a small number of representative events, our prediction model is found to mispredict performance, power and energy efficiency by an average 2.5%, 6.1% and 6% respectively. Our approach as a whole (model and heuristics) is evaluated with a plethora of dynamic, multi-application scenarios. Without considering performance requirements of individual applications, it proves to be 3% more energy efficient than the *powersave* governor and twice as energy efficient compared to the *interactive* and *ondemand* governors. Our approach is also able to support performance requirements of individual applications, at the cost of 18% lower energy efficiency versus the *powersave* governor, which however misses the performance targets by 23%. Furthermore, our approach maintains an efficiency advantage of 52% and 58% over the governors that satisfy said requirements (*interactive* and *ondemand* respectively). Lastly, exploring our heuristics' execution time scaling with respect to system size, we find that they can be used on systems many times bigger than the one used for the experiments: all algorithms' execution time grows in a slower-than linear manner and is at most $1.23msec$ on a system of 128 cores.

5.2.3 Runtime Management for Load Balancing

The third part of the thesis is dedicated to optimizing *Work Stealing* decisions, with the goal of achieving dynamic load balancing and in this manner improve the performance of multiprocessors. The main contributions of this part are as follows:

- **Obtaining system load information with negligible overheads.**

  We introduced a novel, lightweight method to construct an approximate system load view. By integrating load querying with steal attempts, we allow individual cores to have incomplete information about which system cores or regions are likely to be loaded and, as such, good targets for stealing work from.

- **Proposing a work stealing algorithm.**

  We developed an algorithm that performs victim core selection whenever a core becomes idle. The algorithm uses the aforementioned approximate system load view and also considers memory hierarchy and system topology, to increase the probability of a beneficial steal.

- **Evaluating for diverse workloads.**

  We evaluated our strategy using workloads featuring different characteristics with respect to load balancing: balanced, jittery and unevenly distributed workloads, dynamically generated DAGs, and a workload with complex task dependencies that create imbalance.

Our experimental evaluation shows that our proposed strategy achieves good all-around performance for all examined workloads. Among the 12 unbalanced workloads we examine, we achieve better performance than random, hierarchical and local work stealing for six of them, being at most 8.3% slower among the rest. However, the alternative strategies that achieve better performance in one or more of the remaining six cases, incur a penalty of at least 89% to some other workload. Furthermore, our strategy incurs negligible overheads (1.4%) compared to no work stealing, when used on a perfectly balanced workload, as opposed to at least 37% and 12% penalty respectively for random and hierarchical stealing. The last two points prove that our proposed work stealing approach can be used safely on a multiprocessor, without a priori knowledge of the types of workloads that it will run.

5.3 Proposed Research Directions

In this thesis we showed how runtime management can help multicore systems maintain their efficiency when unpredictable events happen. In this final section we propose, for each of the three individual studies, the most interesting, in our opinion, direction towards further improving it:

Runtime Management for Graceful Degradation: Our runtime algorithms for graceful degradation of adaptive multicores focus on maximizing system efficiency for its current status, i.e., the set of permanent faults that have happened on it so far. However, it would be interesting to make more long-term decisions for a combined lifetime-efficiency metric, considering aging effects and how our decisions affect them. This work would require augmenting our event generator (see Section 2.4.1.2) to work online and estimate the effect of the current workload mapping on the wearout of components. Then, our algorithms, instead of just optimizing for system efficiency at the present time, would have to also consider how each configuration would burden the system components in the future: a configuration that is very efficient on the current system status, might affect the aging of some healthy hardware components so seriously that faults appear on them in the future, resulting to an abrupt efficiency loss. The algorithms should then be aware of specifications and requirements, such as minimum required system lifetime, minimum efficiency, etc. Such an extension would combine the contributions of our work with works that focus mostly on lifetime extension without considering all sources of flexibility that we do, such as [20–22].

Runtime Management for Energy Efficiency: One aspect that is not considered in the current implementation of our strategy is that applications can change their behaviour in time, when they transition between different *phases*. If a mechanism to detect these phase changes is implemented, each application phase can be treated independently, as a separate entity in the runtime. More specifically, each phase would, in this case, be profiled separately, to extract its particular characteristics when it comes to resource requirements. Then, in runtime, whenever the application transitions to a different phase, it would be evaluated anew, possibly resulting to different mapping and frequency decisions, adapting to the characteristics of each phase. Furthermore, this would allow the runtime manager to deal with multithreaded applications efficiently. Multithreaded applications consist of phases that can benefit in varying degrees from parallelism: sequential phases should be mapped on a single processor, while parallelizable phases can benefit from being mapped on multiple processors. Thus, when dealing with multithreaded applications, a phase detection mechanism would allow the runtime manager to make more interesting decisions, considering the potential parallelization benefit as well as the current availability of idle processors in the system, to optimize efficiency. Phase detection has been implemented in works such as [31].

Runtime Management for Load Balancing: In the experimental evaluation of our work stealing strategy, we have observed that *WS_Global: Work stealing using global system view* produces some interesting results (see Sections 4.4.3 and 4.5.1). Although there are workloads for which it incurs unacceptable performance penalties (being up to 2.7 times slower than the baseline), there are also workloads for which it is up to 9.6% better than our proposed approach. In fact, WS_Global is punished severely by finer task granularities, resulting to more dense steal attempts in time and making a bottleneck of the global system view. We expect these penalties to become worse for systems bigger than the 48 core system we used in these experiments. However, this trend suggests that our proposed approach could benefit from a more strongly hierarchical implementation. Within neighborhoods of cores, one version of the system view could be kept, similar to the global one but confined to a small part of the system. Thus, local steals can benefit from the complete information, while at the same time avoiding the creation of a bottleneck for the whole system. Then, the precision and detail of the system view information could get gradually lower, as we move to more remote regions. Lastly, different neighborhoods could occasionally share, with each other, their more complete knowledge of the local region, possibly dedicating a chunk of the time of one core per neighborhood to this purpose. The above has to be approached by means of design space exploration, considering the system size, the depth of the system's hierarchy and the task granularity, to find optimal design points for each combination of the above.

The directions proposed in this section are an indicative, non-exhaustive list of interesting topics. All three studies can benefit by deeper investigation.

Bibliography

[1] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, 2005.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[3] Phillip Krueger and Miron Livny. *Load balancing, load sharing and performance in distributed systems*. 1987.

[4] M.L. Fair et al. Reliability, availability, and serviceability (ras) of the ibm eserver z990. *IBM Jour. of Research & Development*, 48(3-4):519–534, 2004.

[5] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop advanced architecture. In *DSN*, 2005.

[6] C. LaFrieda et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *DSN*, pages 317–326, 2007.

[7] M. Imai, T. Nagai, and T. Nanny. Pair and swap: An approach to graceful degradation for dependable chip multiprocessors. In *Int. Conf. on Depend. Syst. & Networks WS (DSN-W)*, pages 119 –124, 2010.

[8] J.L. Weston, M. Imai, T. Nagai, and T. Nanya. An efficient decision unit for the pair and swap methodology within chip multiprocessors. *IEEE Pacific Rim Int. Symp. on Dependable Computing*, 0:62–69, 2010.

[9] Bogdan F. Romanescu and Daniel J. Sorin. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *PACT*, pages 43–51, 2008.

[10] S. Gupta, Shuguang Feng, A. Ansari, and S. Mahlke. Stagenet: A reconfigurable fabric for constructing dependable cmps. *IEEE Trans. on Computers*, 60(1):5–19, 2011.

[11] Andrea Pellegrini, Joseph L. Greathouse, and Valeria Bertacco. Viper: virtual pipelines for enhanced reliability. In *ISCA '12*, pages 344–355, 2012.

[12] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. Resilient high-performance processors with spare ribs. *IEEE Micro*, 33(4):26–34, 2013.

[13] I. Hong, M. Potkonjak, and R. Karri. Heterogeneous bisr-approach using system level synthesis flexibility. In *ASP-DAC*, pages 289–294, 1998.

[14] Nicholas Weaver, J. H. Kelm, and M. I. Frank. EmÖčode: Masking hard faults in complex functional units. In *DSN*, pages 458–467. IEEE, 2009.

[15] A. Benso, S. Chiusano, and P. Prinetto. A self-repairing execution unit for microprogrammed processors. *Micro, IEEE*, 21(5):16 –22, 2001.

[16] Y. Nakamura and K. Hiraki. Highly fault-tolerant fpga processor by degrading strategy. In *Pacific Rim Int. Symp. on Dependable Computing*, pages 75 – 78, 2002.

[17] R. Rodrigues and S. Kundu. On graceful degradation of microprocessors in presence of faults via resource banking. In *IEEE Int. On-Line Testing Symp. (IOLTS)*, pages 61 –66, 2011.

[18] R. Rodrigues and S. Kundu. On graceful degradation of chip multiprocessors in presence of faults via flexible pooling of critical execution units. *On-Line Testing Symp., IEEE Int.*, pages 67–72, 2011.

[19] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese. Microprocessor fault-tolerance via on-the-fly partial reconfiguration. In *IEEE Europ. Test Symposium (ETS)*, pages 201 –206, 2010.

[20] S. Feng, S. Gupta, A. Ansari, and S. A. Mahlke. Maestro: Orchestrating lifetime reliability in chip multiprocessors. In *HiPEAC Conf.*, pages 186–200, 2010.

[21] Alessandro Baldassari, Cristiana Bolchini, and Antonio Miele. A dynamic reliability management framework for heterogeneous multicore systems. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2017, Cambridge, United Kingdom, October 23-25, 2017*, pages 1–6, 2017.

[22] Cristiana Bolchini, Luca Cassano, and Antonio Miele. Lifetime-aware load distribution policies in multi-core systems: An in-depth analysis. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 804–809, 2016.

[23] B. Nahar and B. H. Meyer. Rotr: Rotational redundant task mapping for fail-operational mpsocs. In *IEEE DFTS*, 2015.

[24] P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin, and M. Sami. System adaptivity and fault-tolerance in noc-based mpsocs: The madness project approach. In *2012 15th Euromicro Conference on Digital System Design*, pages 517–524, Sept 2012.

[25] Cristiana Bolchini, Matteo Carminati, Tulika Mitra, and Thannirmalai Somu Muthukaruppan. Combined on-line lifetime-energy optimization for asymmetric multicores. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2016, Storrs, CT, USA, September 19-20, 2016*, pages 35–40, 2016.

[26] Marcus Hähnel and Hermann Härtig. Heterogeneity by the numbers: A study of the odroid xu+e big. little platform. In *Proceedings of the 6th USENIX Conference on Power-Aware Computing and Systems*, HotPower'14, pages 3–3, Berkeley, CA, USA, 2014. USENIX Association.

[27] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Eighteenth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, 2013.

[29] A. Pathania, S. Pagani, M. Shafique, and J. Henkel. Power management for mobile games on asymmetric multi-cores. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 243–248, July 2015.

[30] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Feb 2013.

[31] Ujjwal Gupta, Chetan Arvind Patil, Ganapati Bhat, Prabhat Mishra, and Umit Y. Ogras. Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 16(5s):123:1–123:20, September 2017.

[32] Ali Aalsaud, Rishad Shafik, Ashur Rafiev, Fie Xia, Sheng Yang, and Alex Yakovlev. Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 368–373, New York, NY, USA, 2016. ACM.

[33] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM Trans. Archit. Code Optim.*, 13(4):43:1–43:25, December 2016.

[34] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack amp; cap: Adaptive dvfs and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, Dec 2011.

[35] E. Del Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini. Workload-aware power optimization strategy for asymmetric multiprocessors. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, pages 531–534, San Jose, CA, USA, 2016. EDA Consortium.

[36] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 54–61, Oct 2013.

[37] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. Sparta: Runtime task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 27:1–27:10, New York, NY, USA, 2016. ACM.

[38] S. Sarma, T. Muck, L. A. D. Bathen, N. Dutt, and A. Nicolau. Smartbalance: A sensing-driven linux load balancer for energy efficiency of heterogeneous mpsocs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[39] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Tran. on Parallel and Distributed Systems*, 2(3), July 1991.

[40] Robert D. Blumofe et al. Scheduling multithreaded computations by work stealing. *J. of the ACM*, 46(5):720–748, 1999.

[41] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *IPDPS*, pages 1–12, April 2010.

[42] A. De Boelelaan et al. Adaptive load-balancing for divide-and-conquer grid applications. *J. of Supercomputing*, 2004.

[43] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *PGAS*, Oct 2011.

[44] Richard M. Yoo et al. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *SPAA*, 2013.

[45] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00*, pages 1–12, 2000.

[46] Quan Chen, Minyi Guo, and Haibing Guan. Laws: Locality-aware work-stealing for multi-socket multi-core architectures. In *ICS '14*, pages 3–12, 2014.

[47] Quan Chen, Minyi Guo, and Zhiyi Huang. Cats: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *ICS*, pages 163–172, New York, NY, USA, 2012. ACM.

[48] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3):10:1–10:51, September 2014.

[49] S. Tzilis and I. Sourdis. A runtime manager for gracefully degrading socs. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 216–221, Oct 2014.

[50] Stavros Tzilis, Ioannis Sourdis, Vasileios Vasilikos, Dimitrios Rodopoulos, and Dimitrios Soudris. Runtime management of adaptive mpsocs for graceful degradation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 5:1–5:10, New York, NY, USA, 2016. ACM.

[51] A. Malek, S. Tzilis, D.A. Khan, I. Sourdis, G. Smaragdos, and C. Strydis. A probabilistic analysis of resilient reconfigurable designs. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 141–146, Oct 2014.

[52] G. Smaragdos, D.A. Khan, I. Sourdis, C. Strydis, A. Malek, and S. Tzilis. A dependable coarse-grain reconfigurable multicore array. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 141–150, May 2014.

[53] D. Theodoropoulos, D. Pnevmatikatos, S. Tzilis, and I. Sourdis. The desyre runtime support for fault-tolerant embedded mpsocs. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 197–204, Aug 2014.

[54] A. Malek, S. Tzilis, D. A. Khan, I. Sourdis, G. Smaragdos, and C. Strydis. Reducing the performance overhead of resilient cmps with substitutable resources. In *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 191–196, Oct 2015.

[55] I. Sourdis, D. Khan, A. Malek, S. Tzilis, G. Smaragdos, and C. Strydis. Resilient cmps with mixed-grain reconfigurability. *Micro, IEEE*, 2015. in print.

[56] Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. Energy-efficient runtime management of heterogeneous multicores using online projection. *ACM Trans. Archit. Code Optim.*, 15(4):63:1–63:26, January 2019.

[57] S. Tzilis, M. Pericàs, P. Trancoso, and I. Sourdis. Swas: Stealing work using approximate system-load information. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 309–318, Aug 2017.

[58] V. Vasilikos, G. Smaragdos, C. Strydis, and I. Sourdis. Heuristic search for adaptive, defect-tolerant multiprocessor arrays. *ACM Transactions on Embedded Computing Systems*.

[59] S. Gupta, Shuguang Feng, A. Ansari, J. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *MICRO-41*, pages 141–151, nov. 2008.

[60] S.T. Chakradhar and A. Raghunathan. Best-effort computing: Re-thinking parallel software and hardware. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 865–870, June 2010.

[61] D. Mohapatra, V.K. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[62] Titos Saridakis. Design patterns for graceful degradation. *Transactions on Pattern Languages of Programming*, 1:67–93, 2009.

[63] J. Srinivasan, S.V. Adve, Pradip Bose, and J.A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *ISCA '05*, pages 520 – 531, 2005.

[64] Premkishore Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings 21st International Conference on Computer Design*, pages 481–488, Oct 2003.

[65] R. Noji, S. Fujie, Y. Yoshikawa, H. Ichihara, and T. Inoue. An fpga-based fail-soft system with adaptive reconfiguration. In *IEEE Int. On-Line Testing Symp.*, pages 127 –132, 2010.

[66] M. Glaß, M. Lukasiewycz, C. Haubelt, and J. Teich. Incorporating graceful degradation into embedded system design. DATE '09, pages 320–323, 2009.

[67] A. Miele. A software framework for dynamic self-repair in embedded socs exploiting reconfigurable devices. In *IEEE Int. Conf. onAutomation Quality and Testing Robotics (AQTR)*, volume 2, pages 1 –6, 2010.

[68] A. Das and A. Kumar. Fault-aware task re-mapping for throughput constrained multimedia applications on noc-based mpsocs. In *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 149–155, Oct 2012.

[69] Onur Derin, Deniz Kabakci, and Leandro Fiorin. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 129–136, New York, NY, USA, 2011. ACM.

[70] I. Sourdis, C. Strydis, A. Armato, C.S. Bouganis, B. Falsafi, G.N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D. Pnevmatikatos, D.K. Pradhan, G. Rauwerda, R.M. Seepers, R.A. Shafik, K. Sunesen, D. Theodoropoulos, S. Tzilis, and M. Vavouras. Desyre: On-demand system reliability. *Microprocessors and Microsystems*, 37(8, Part C):981 – 1001, 2013. Special Issue on European Projects in Embedded System Design: {EPESD2012}.

[71] Alirad Malek, Ioannis Sourdis, Stavros Tzilis, Yifan He, and Gerard Rauwerda. Rqnoc: A resilient quality-of-service network-on-chip with service redirection. *ACM Trans. Embed. Comput. Syst.*, 15(2):28:1–28:25, February 2016.

[72] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *DAC*, pages 174:1–174:9, 2013.

[73] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[74] Amit Sinha and Anantha P. Chandrakasan. Jouletrack: A web based tool for software energy profiling. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 220–225, New York, NY, USA, 2001. ACM.

[75] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[76] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[77] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. Scc: A flexible architecture for many-core platform research. *Computing in Science Engineering*, 13(6):79–83, Nov 2011.

[78] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[79] Christian P. Robert and George Casella. *Introducing Monte Carlo methods with R*. Use R! Springer, New York, London, 2010.

[80] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4*, pages 3–14, 2001.

[81] http://www.autosar.org/.

[82] http://www.iso.org/iso/catalogue_detail?csnumber=43464.

[83] Samsung. Samsung Exynos. `http://www.samsung.com/semiconductor/minisite/exynos/`. Online; Accessed: 2018-09-12.

[84] Qualcomm. Qualcomm snapdragon 808 processor. `https://www.qualcomm.com/products/snapdragon/processors/808`. Online; Accessed: 2018-09-12.

[85] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. H. Katevenis. Modeling energy-performance tradeoffs in arm big.little architectures. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sept 2017.

[86] Arunachalam Annamalai, Rance Rodrigues, Israel Koren, and Sandip Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 63–72, Piscataway, NJ, USA, 2013.

[87] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pages 38–43, Aug 2007.

[88] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000.

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

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

[91] Samsung. ODROID XU3. `https://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127&tab_idx=2`. Online; Accessed: 2018-09-12.

[92] NASA. NAS Benchmarks. `https://www.nas.nasa.gov/publications/npb.html`. Online; Accessed: 2018-09-12.

[93] Texas Instruments. INA231 sensor. `http://www.ti.com/lit/ds/symlink/ina231.pdf`. Online; Accessed: 2018-09-12.

[94] Bhavishya Goel. *Measurement, Modeling, and Characterization for Energy-efficient Computing*. Chalmers University of Technology and Göteborg University, 2016.

[95] V. Pallipadi, S. Li, and A. Belay. cpuidle - do nothing, efficiently... In *Proceedings of the Linux Symposium*, Vol. 2, pages 119–125, 2016.

[96] Ilias Vougioukas, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. Nucleus: Finding the sharing limit of heterogeneous cores. *ACM Trans. Embed. Comput. Syst.*, 16(5s):152:1–152:16, September 2017.

[97] Miquel Pericàs. POSTER: ξ-TAO: A Cache-centric Execution Model and Runtime for Deep Parallel Multicore Topologies. In *PACT '16*, pages 429–431, 2016.

[98] Andi Drebes et al. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3):30:1–30:25, August 2014.

[99] Hrushit Parikh, Vinit Deodhar, Ada Gavrilovska, and Santosh Pande. Efficient distributed workstealing via matchmaking. In *PPoPP*, pages 37:1–37:2, 2016.

[100] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and implementation of a customizable work stealing scheduler. In *ROSS*, June 2013.

[101] Martin Wimmer et al. Work-stealing with configurable scheduling strategies. In *18th ACM SIGPLAN Symp. on PPoPP*, pages 315–316, 2013.

[102] Alina Sbirlea, Kunal Agrawal, and Vivek Sarkar. Elastic tasks: Unifying task parallelism and spmd parallelism with an adaptive runtime. In *Euro-Par: Parallel Processing*, volume 9233, pages 491–503. Springer, 2015.

[103] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *PLDI '10*, 2010.

[104] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *SIGPLAN*, June 1998.

[105] Threading building blocks (intel tbb).

[106] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *IEEE Int. Conf. on Cluster Computing*, pages 142–151, Sept 2008.