Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Exploiting Concurrency and Heterogeneity for Energy-efficient Computing: An Actor-based Approach

par

Yaroslav Hayduk

Thèse

présentée à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, Directeur de thèse
Université de Neuchâtel, Suisse

Dr. Anita Sobe, Co-Directeur de thèse
Accenture, Suisse

Prof. Romain Rouvoy
Université des Sciences et Technologies
de Lille 1 / Inria, France

Prof. Peter Kropf
Université de Neuchâtel, Suisse

Dr. Osman Ünsal
Barcelona Supercomputing Center, Espãna

Soutenue le 25 janvier 2017

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Yaroslav HAYDUK

Titre:

"Exploiting Concurrency and Heterogeneity for Energy-efficient Computing: An Actor-based Approach"

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Prof. Peter Kropf, Université de Neuchâtel, Suisse
- Prof. Romain Rouvoy, Université de Lille, France
- Dr Osman Sabri Ünsal , Barcelona Supercomputing Center, Barcelone, Espagne
- Dr Anita Sobe, Accenture, Suisse

Neuchâtel, le 7 février 2017

Le Doyen, Prof. R. Bshary

ACKNOWLEDGMENTS

This thesis is dedicated to my immediate family — my mother Nina, my father Bohdan, and my sister Annie, who have given me unconditional love and support throughout the process of my studies. The work I have accomplished would not have been possible without each one of them. My family gave me a great deal of encouragement. Whenever I needed it, they were there for me. My family is what keeps me strong at moments when I feel like giving in to life's challenges. I am very fortunate to have been raised in such a family.

Among the people to be thanked from the University of Neuchâtel, first and foremost, I would like to thank my adviser, Professor Pascal Felber, for an eye-opening opportunity to join the Computer Science Department. He gave me motivation, a lot of support for my ideas and encouragement throughout this research. Pascal was always willing to engage in insightful conversations whenever I stumbled upon a problem. He was also a constant source of support during the course of the ParaDIME project.

My co-adviser Dr. Anita Sobe, with whom we worked diligently together despite our different work patterns, clearly deserves much praise. Anita is (mostly) a very patient reader who has put up with my habit of missing focusing on the context first. Anita is also highly organized, gets to the University early, and works in a deliberate and well-planned style. The work for the ParaDIME project and the irregular working hours often kept me up well into the early morning often conflicted with Anita's regular working schedule. I'm grateful that Anita was kind enough to accommodate my late-night work schedule. She has also dedicated many hours of her time assisting me with the final draft of this thesis, for which I am very grateful and deeply indebted.

Special thanks go to Dr. Patrick Marlier, who's **nix* knowledge and willingness to be there when you need help I really admire. Patrick graciously provided his invaluable assistance and advice. Without it, it would be rather impossible to handle all the ScalaSTM intricacies before the deadlines that we had to meet. I would also like to extend my gratitude to Dr. Derin Harmanci for his ideas and writing help of the initial "actor" paper. I was lucky to be teamed with someone who could transform the general ideas that we would sketch on the board, in the process of brainstorming, into a scientific text worthy to be accepted at the highest quality academic venues. I want to thank Maria Carpen-Amarie and Raphaël Barazzutti who have given me numerous advice and support.

Throughout my years of study, the University of Neuchâtel has cultivated and nourished me. The delight of being at the University of Neuchâtel surrounded by people of quality and wisdom could be considered a highlight of this thesis.

RÉSUMÉ

En vue d'améliorer l'efficacité énergétique, les fournisseurs de cloud computing ont commencé à examiner des méthodes radicales pour réduire le niveau de consommation énergétique. Le problème de l'optimisation de l'efficacité énergétique doit être considéré tant au niveau matériel qu'au niveau logiciel. De nombreux perfectionnements ont été réalisés pour réduire la consommation d'énergie au niveau matériel. Bien que ceux-ci soient efficaces, leur utilisation individuelle ne garantit pas encore une diminution considérable de la dissipation énergétique. Ainsi, un objectif majeur de cette thèse est de proposer une structure logicielle qui s'adapte de différentes façons afin de répondre aux exigences de performance logicielle et de consommation énergétique. Afin d'élaborer un tel système, je me concentre en premier lieu sur les modèles de transmission de messages, et en particulier sur le modèle acteur.
Le modèle acteur est probablement trop conservateur dans sa manière de traiter la concurrence. En effet, j'ai découvert un certain nombre de problèmes concernant la programmation concurrente avec les paramètres par défaut du modèle acteur, qui sont: 1) un retard de mise en attente des messages lors d'actions coordonnées, 2) un traitement séquentiel des messages, 3) des problèmes de performance dans le traitement concurrent de messages en cas de forte concurrence, et enfin 4) l'incapacité du modèle acteur à utiliser harmonieusement les ressources d'un processeur graphique.
J'utilise la mémoire transactionnelle pour améliorer le processus de transmission de messages du modèle acteur, et je propose également une structure de DSL, c'est-à-dire un langage spécifique à ce domaine, pour aborder une prise en charge du processeur graphique. En répondant aux problèmes identifiés ici, je montre qu'on peut améliorer significativement la performance, l'efficacité énergétique, ainsi que la simplicité de programmation dans le modèle acteur.

Mots-clés: modèle acteur, mémoire transactionnelle, systèmes distribués, architecture, cloud computing, processeur graphique.

ABSTRACT

To accommodate energy efficiency, cloud providers started looking into radical ways of reducing the energy consumption. Energy-efficient optimizations should be addressed at both software and hardware levels of the datacenter. There have been numerous improvements in reducing the energy consumption on the hardware level. While they are efficient, however, their usage alone does not warrant significant decrease of energy dissipation. I argue that software-based methods for energy efficiency have not received as much attention as hardware-based methods. As such, in this thesis, an important target is to provide a software framework that adapts itself in many different aspects in order to satisfy application performance and energy consumption requirements. For developing such a framework, I primarily concentrate on message passing models and, in particular, on the actor model.
The actor model is arguably too conservative in its default concurrent settings. Specifically, I have identified a number of issues with the default concurrency settings of the actor model, which are: (1) message queuing delay during coordinated actions, (2) sequential message processing, (3) performance problems for concurrent message processing during high contention, and (4) the inability of the actor model to seamlessly exploit GPU resources. I use transactional memory for optimizing actor model's message passing process, as well as propose DSL support for introducing GPU support. By addressing the identified problems I show that we can significantly improve performance, energy efficiency and programmability in the actor model.

Keywords: actor model, Transactional Memory, distributed systems, architecture, cloud computing, GPU.

Contents

List of Figures **xv**

List of Tables **xviii**

1 Introduction **1**

1.1 Context 1

1.2 Motivation 3

1.3 Contributions 6

1.4 Organization of the Thesis 7

2 Background **9**

2.1 Energy Efficiency 9

2.2 Parallel (Multi-core) and Heterogeneous Systems 10

2.2.1 Amdahl's and Gustafson's Laws 10

2.2.2 Programming Multi-core Systems 11

2.3 GPU Computing 12

2.4 Programming Models and Actors 13

2.4.1 Actors as the Main Entities in the Actor Model 14

2.4.2 Asynchronous Message Passing 14

2.4.3 The Akka Framework 15

2.5 The ParaDIME Stack 17

2.5.1 Efficient Message Passing 17

2.5.2 Multi-data Center Scheduler 18

2.5.3 Operation Below Safe V_{dd} 18

2.5.4 Approximate Computing 18

3 Related Work **21**
3.1 Concurrency and Speculation . . . 21
3.1.1 Alternative Models and Frameworks . . . 21
3.1.2 Actor Model Enhancements . . . 22
3.2 Dynamic Scheduling . . . 24
3.2.1 Contention Management . . . 24
3.2.2 Power-Aware Techniques . . . 25
3.3 Heterogeneity . . . 26
3.4 Summary . . . 27

4 Exploiting Concurrency and Speculation in the Actor Model **29**
4.1 Introduction . . . 29
4.2 Problem Statement . . . 30
4.3 Message Processing Model . . . 31
4.4 Implementation . . . 32
4.4.1 Concurrent Processing of Messages . . . 33
4.4.2 Non-blocking Coordinated Transactions . . . 33
4.5 Evaluation . . . 35
4.5.1 Read-dominated Workload . . . 35
4.5.2 Write-dominated Workload . . . 39
4.5.3 Non-blocking Concurrent Processing . . . 42
4.5.4 Comparison to Habanero-Scala . . . 43
4.5.5 Discussion . . . 44
4.6 Summary . . . 45

5 Dynamic Scheduling for Message Processing **47**
5.1 Introduction . . . 47
5.2 Concurrent Message Processing . . . 48
5.3 Dynamic Concurrent Message Processing . . . 49
5.3.1 STM Message Processing . . . 49

5.3.2 Read-only Message Processing . . . 51
5.4 Evaluation . . . 52
5.4.1 List Benchmark . . . 53
5.4.2 Simulation of the Hydraulic Subsurface . . . 55
5.5 Summary . . . 59

6 Exploiting Heterogeneity for Energy Efficiency **61**
6.1 Introduction . . . 61
6.2 Improving Energy Efficiency . . . 63
6.2.1 Reducing Power Consumption . . . 63
6.2.2 Reducing Execution Time . . . 63
6.3 Enabling Heterogeneous Actors . . . 64
6.3.1 JNI . . . 65
6.3.2 RabbitMQ . . . 65
6.3.3 DSL . . . 66
6.4 Resource Load Balancing with Heterogeneous Actors . . . 67
6.5 Experimental Setup . . . 68
6.5.1 Parallel Implementation with Actors . . . 69
6.5.2 Heterogeneous Implementation with JNI . . . 70
6.5.3 Heterogeneous Implementation with RabbitMQ . . . 71
6.5.4 Heterogeneous Implementation Using a DSL . . . 71
6.5.5 Heterogeneous Work Balancing Implementation . . . 71
6.5.6 System Configuration . . . 72
6.6 Results and Discussion . . . 72
6.6.1 Reducing Power Consumption . . . 72
6.6.2 Reducing Execution Time . . . 74
6.6.3 Resource Load Balancing with Heterogeneous Actors . . . 75
6.7 Summary . . . 76

7 Conclusion **77**

7.1 Summary of Contributions 77

7.2 Future Directions 78

7.2.1 Concurrent and Speculative Message Processing 78

7.2.2 Dynamic Message Processing 79

7.2.3 Heterogeneous Actor Model 79

A Publications **81**

References **83**

List of Figures

1.1 Monthly datacenter costs breakdown. 2
1.2 Activity distribution of Google clusters. 2

2.1 Actor model: Communication through messages. 14
2.2 The ParaDIME stack overview. 17

4.1 Sequential processing and its effect on execution time. 30
4.2 Optimized message processing and its effect on execution time. 31
4.3 Implementation of concurrent message processing. 33
4.4 Implementation of non-blocking coordinated processing. 34
4.5 Concurrent sorted integer linked-list benchmark using actors. 35
4.6 Execution time for sequential, concurrent, and non-blocking message processing on a read-dominated workload. 36
4.7 Sequential processing of the coordinated sum operation. 37
4.8 Concurrent processing of the coordinated sum operation. 37
4.9 Execution time for sequential, concurrent, and non-blocking message processing strategies on a read-dominated workload. 38
4.10 Execution time for sequential, concurrent, and non-blocking message processing on a write-dominated workload. 40
4.11 Execution time for sequential, concurrent, and non-blocking message processing on a write-dominated workload for 1, 8, and 16 list actors. 41
4.12 Execution time for sequential, concurrent, non-blocking, and combined message processing. 42
4.13 Execution time comparison with Habanero-Scala. 43

5.1 Concurrent message processing in a read-dominated workload within a shared linked list. 50
5.2 The different handling of STM messages and read-only messages. 51
5.3 Throughput and rollback count for the write-dominated workload: Static thread allocation. 52
5.4 Throughput and rollback count for the write-dominated workload: Dynamic thread allocation. 53
5.5 Throughput and rollback count for the mixed workload: Static thread allocation. 55
5.6 Throughput and rollback count for the mixed workload: Dynamic thread allocation. 56
5.7 Matching an SG point with the same pattern in the TI. 57
5.8 Throughput and rollbacks of STM messages and read-only messages over time: Static thread allocation. 58
5.9 Throughput and rollbacks of STM messages and read-only messages over time: Dynamic thread allocation. 58

6.1 A typical data-parallel algorithm: Implementation strategies. 64
6.2 Heterogeneous actors using JNI. 65
6.3 Heterogeneous actors using RabbitMQ. 65
6.4 Heterogeneous actor using DSLs from the programmer's view. 66
6.5 BalancingPool Router in Akka. 66
6.6 Comparison of the power consumption, execution time, and energy consumption using different governors. 73
6.7 Programmability of different heterogeneous implementations. 75
6.8 Execution time of the profiling phase and of the remaining workload for a mixed CPU/GPU execution. 75

List of Algorithms

1 The baseline sequential k-means logic. 69

2 K-means iteration actor logic. 70

3 K-means worker actor algorithm. 70

List of Tables

6.1 Hardware characteristics of the experimental environment. 68

Chapter 1

Introduction

1.1 Context

Today's data centers provide the compute infrastructure for solving a plethora of problems in diverse fields ranging from the support of simple website hosting to extracting complex relationships in big data. The growing popularity of cloud computing has, however, increased the scale of data centers, which results in significant increase in energy consumption. Cloud computing as a whole started consuming more energy than countries like Germany or India [29]. To deal with the extensive energy requirements, companies are now required to invest significantly not only in the compute infrastructure itself, but also in the cooling equipment, which is often over-provisioned to support peak utilization [60]. This investment can be especially significant for start-up companies that require bootstrapping an in-house data center (e.g., email servers providing strict guarantees on data security) given limited capital resources. Figure 1.1 demonstrates the monthly costs breakdown for an average datacenter [50], which suggests that costs related to power are rather significant as compared to the overall costs. Indeed, the costs related to power are the fastest growing operational costs in a modern data center [60]. Also, to meet the performance needs of their applications, developers started to use extreme-scale systems such as teradevices which in turn increases the power consumption of the platform. Considering the outlined issues, it is not surprising that power efficiency is gaining increased attention in academia as well as in industry. When talking about power efficiency, we also talk about energy efficiency when we want to take time into account ($E = P \cdot T$); these terms, however, tend to be used interchangeably.

In order to overcome the energy efficiency challenges, we need novel computing paradigms that address energy efficiency. One of the promising solutions is to incorporate parallel distributed methodologies at different abstraction levels, as advocated by the ParaDIME[1] project, which provided the framework for the results presented in this thesis. ParaDIME's objective is to attack the energy efficiency challenges by radical software-hardware techniques that are driven by future circuit and device characteristics on one side, and by a programming model based on message passing on the other side [90].

In ParaDIME we claim that, up until recently, the cloud computing community was exclusively focused on performance optimizations. For example, at the scale of the data center, to ensure uninterrupted uptime and low response latency, servers are using, among others, multiple processors, have multiple hard drives for data background replication, as well as redundant power supplies. Servers are also commonly left idle, and during the time of being idle they are also consuming a substantial amount of power. A recent report [115] suggests that in the US, up to 30 percent of servers are left idle as well as that the majority of

[1] "Parallel Distributed Infrastructure for Minimization of Energy" (`www.paradime-project.eu`).

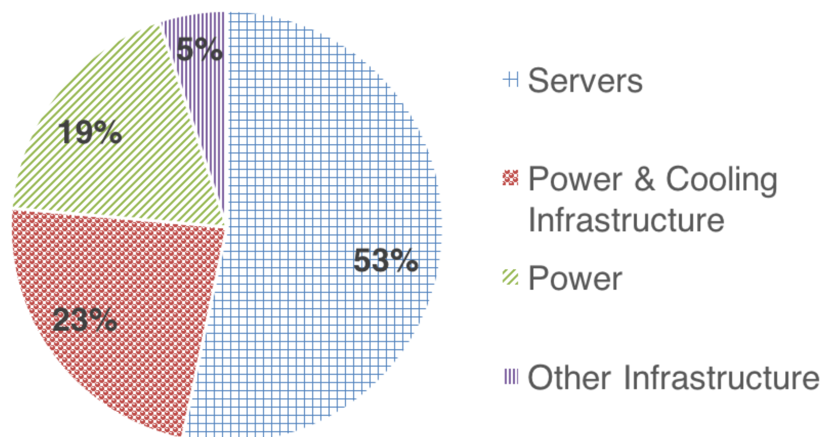

Figure 1.1: Monthly datacenter costs breakdown [50].

the servers are running at 10 to 15 percent of their capacity. The conclusions of the report are also echoed in the recent data about the average CPU utilization of a 20,000 server Google cluster during a 3-month period [60]. The cluster continuously supports a variety of online services as well as executes mixed workloads. From Figure 1.2 we can see that servers spend most of the time working in the 10-50% CPU utilization range. All these measures combined can significantly hinder energy efficiency.

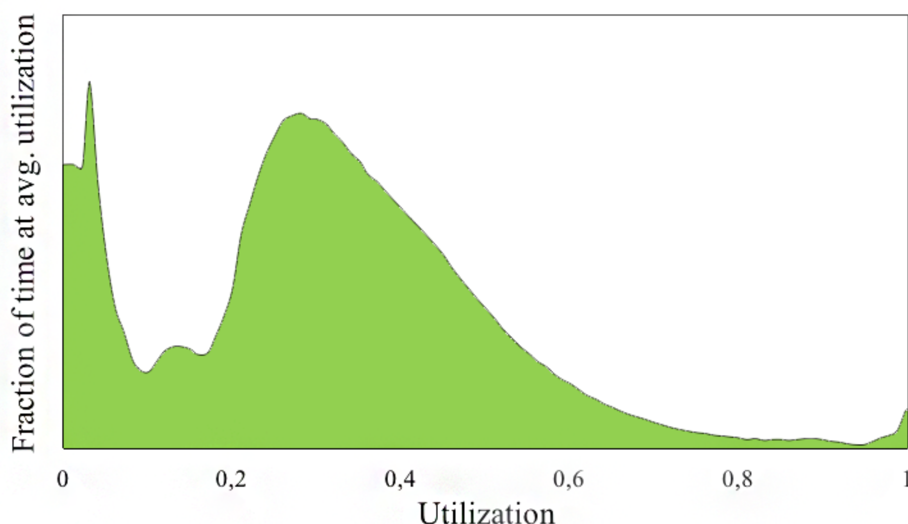

Figure 1.2: Average activity distribution of a sample of two Google clusters [60].

At the scale of CPUs, power was successfully traded for performance. Power, however, has become the main limiter of performance scaling beyond the 65nm generation CPUs [17]. Looking at the design of modern CPUs, we can see that CPU manufacturers have hit a so called **Power Wall**, which postulates that power consumption increases non-linearly with the increase of the clock rate. With the increase of CPU speeds into the 3-4 GHz range, they required a prohibitive amount of electrical power. Also, increasing clock rates any further

would exceed the power density that could be dealt with by air cooling alone, thus leading to power-inefficient computations [85].

To continue scaling performance proportionally, chip manufacturers embraced the move to multi-core CPUs, packing multiple cores into a single chip. With the increased core counts we also expect technology scaling to face the problem of "dark silicon" [37], implying that the new CPUs will have more transistors than can be simultaneously powered. This happens because the switching power per transistor has not been scaling linearly with the reduction of the transistor area, increasing power density. Admittedly, the problem of "dark silicon" in the near future will lead us to use devices with completely new characteristics [90].

We can also see that not only the modern CPUs have to comprise multiple cores, but these cores have to be low powered as well. This is, however, not the case for some of the recent multi-core CPUs. For example, the X5647 Xeon CPU (4 cores, running at 2.93 GHz) requires 130 watts while the L5618 Xeon CPU (4 cores running at 2.13 GHz) consumes only 40 watts. Both CPUs have similar features, although we see that, while the X5647 Xeon CPU requires 225 percent more power, it yields only 37 percent more performance.

For single-core CPUs, Moore's Law was allowing the same program to seamlessly scale given powerful CPU, meaning that it was relatively easy to speed up serial single threaded code. In the multi-core world, the performance improvements that can be obtained are highly limited by the ability of new algorithms to take advantage of the available resources. Programming applications which consider multiple cores is a tedious task, as the developer has to ensure that all the cores (and not just one) are doing useful work. Also, as the cores communicate via shared memory, adequate synchronization mechanisms have to be in place in order to avoid concurrency hazards. In particular, locks are commonly used to guarantee mutual exclusion. The concept of mutual exclusion postulates, that given an application that works with multiple threads, only one thread would be able to update a certain memory location in isolation. When a thread successfully acquires a lock, other threads that want to acquire the same lock need to wait, until that lock is released. As such, gains from parallelism are highly limited by the fractions of the code that can be executed on multiple cores in parallel (and not serially). To increase scalability, programmers need to also carefully consider which sections of code need to be locked as well as select the optimal locking granularity, which can be a tedious task. If performed incorrectly, locking can also decrease energy efficiency, as idle cores would be consuming power without performing useful computations. These challenges will be even more significant when moving to 100-core or 1000-core CPU chips. Given high core counts, few applications would have sufficient parallelism levels to be able to exploit the resources. In this context, the structures used to implement shared memory do not scale well and might need to be abandoned.

1.2 Motivation

In the previous section, I argued that not only performance but energy efficiency needs to be considered in the context of effective data center operation. To accommodate energy efficiency, cloud providers started looking into radical ways of reducing the energy consumption. As stated, the core premise of ParaDIME is that energy-efficient optimizations should be addressed at both software and hardware levels of the datacenter.

Measures on the hardware level are well understood and are wide spread. A few examples are:

- circuit and architecture operations below safe voltage limits;

– dynamic voltage & frequency scaling improvements;
– clock- and power-gating;
– the use of specialized energy-aware computing accelerators;
– the utilization of future device characteristics for energy savings;
– power-aware message passing at the hardware level.

Undoubtedly, as can be seen there have been numerous improvements in reducing the energy consumption on the hardware level. While the outlined efforts are efficient, however, they alone do not warrant significant decrease of power dissipation [26, 87]. I argue that software-based methods for energy efficiency have not received as much attention as hardware-based methods. Indeed, as shown by initial case studies conducted by the Cluster Green Software project,[2] more intelligent use of software can lead to drastic energy savings, ranging between 30 percent and 90 percent.

Motivated by the aforementioned issues, in this thesis, an important target is to provide a software framework that adapts itself in many different aspects in order to satisfy application performance and energy consumption requirements. To this end, for developing such a framework I primarily concentrate on message passing models. Given that ParaDIME's main focus is to target future devices, which includes CPUs with a growing number of cores, I chose not to use shared memory programming. Shared memory programming might be impractical for ParaDIME, since the typical primitives used in shared programming do not easily scale for 100-core or 1000-core CPU chips. On the other hand, the main characteristic of message passing models is that they avoid common hazards associated with shared memory concurrent programming, such as livelocks, deadlocks and race conditions.

In the course of my research, I use the actor model [58], a message-passing based model, which has been shown to be effective for high-uptime distributed systems [77]. The actor model is inherently concurrent and is widely used for implementing parallel, distributed, and mobile systems (e.g., Apache Spark [120], Akka [66], RobotS [89]).

The characteristics of the actor model are particularly attractive and contribute to their popularity. Nonetheless, the actor model is arguably too conservative in its default concurrent settings. Specifically, I have identified the following issues:

1. The processing of a message can cause further communication spawning multiple actors, forming a distributed transaction, which we call *a coordinated transaction*. During coordinated actions, actors that have completed executing are required to be blocked i.e., participating actors cannot process new messages until the coordinated transaction commits. Each time an actor is blocked, all the messages queued in the mailbox are inherently blocked since they need to wait for the current coordinated transaction to terminate, which leads to message queuing delay. For queued messages that do not have dependency on the variables in the current coordinated transaction execution, waiting for the currently running coordination transaction to finish results in wasted time.
2. The actor model was developed to support the execution of programs on thousands of independent processors, each having its own encapsulated private memory. One of the fundamental principles of the actor model is that it guarantees sequential message processing. This requirement is a deliberate choice to avoid synchronization hazards and guarantee data consistency. In earlier distributed systems where each physically

[2]`http://www.clustergreensoftware.nl/`

separate computer could perform a single task at a time, this sequential processing of messages was inevitable. However, I argue that today with the multi-core architectures each computer is capable of performing multi-tasking. Hence, sequential processing limits message processing throughput and hence scalability, as it is perfectly possible to process multiple messages in a single actor on a multi-core machine as long as the messages processed concurrently do not have any dependencies.

3. With the new capacity to process multiple messages concurrently in a single actor, in cases of high contention, the performance of parallel processing can drop close to or even below the performance of sequential processing.

4. Along with the safety and scalability advantages of the actor model, one additional advantage is in that the model can seamlessly exploit all the cores of the CPU. Seamless exploitation of CPU resources in the context of the actor model means that the developer is required to provide the actor code and submit it to the actor framework for execution. It is the job of the framework to handle the way the code has to execute. Similarly, developers targeting to exploit GPU resources are required to explicitly interact with the GPU — define GPU code, copy the application-specific data to GPU memory, start the code execution on the GPU, wait for the GPU to complete its work and, finally, copy the results back to CPU memory. These challenges are exacerbated by the fact that GPU programming is distinctly different then CPU programming. As some tasks, as parts of a program, might be better targeted at a CPU, while other parts are data parallelizable and run more efficiently on GPUs, it might be necessary to provide two versions of the same application (e.g., CUDA/Scala).

   Similar steps have to be taken in order to exploit GPU resources from the actor model. As such, programmers do not get the opportunities to seamlessly exploit GPU resources to cater application's performance and energy consumption requirements. I claim that the actor model is lacking programmable solutions in the form of sufficient software support for heterogeneous programming.

Given the described problems, the two main questions that I explore in this thesis are:

1. Can we optimize the message passing processes in the actor model using transactional memory, a technique that has been shown to be effective for concurrent programming?
2. Given that the actor model can seamlessly use CPU resources, can we enhance it such that it could also seamlessly exploit GPU resources, in order to satisfy application performance and energy consumption requirements?

Although message processing effectiveness can be considered to be disconnected to energy consumption, I am interested to observe whether it may turn out to be an important aspect to consider for achieving energy efficiency. For example, while processing messages in a single actor in an optimal way, the framework can also be generating more wasted work, thus making the system less energy-efficient than it could be.

To reduce energy consumption, one can either radically (1) reduce the power consumption — usually at the cost of execution time — or (2) reduce the execution time — usually at the cost of power consumption ($E = P \cdot T$). Therefore, the reasoning behind using GPU computing in the context of the actor model is that it might be a good a fit for reducing the execution time for certain types of applications/workloads (e.g., data-parallel workloads).

1.3 Contributions

In the direction of developing a framework that adapts itself in many different aspects in order to satisfy application performance and energy consumption requirements I retrofitted the actor model in several ways. In particular, I have addressed the problems with the default concurrency settings of the actor model, as identified in Section 1.2. In what follows, I briefly state the problems that I have addressed and outline the solutions as proposed in this thesis.

Sequential message processing

Sequential processing of messages unnecessarily limits message processing throughput and thus limits scalability. To lift this limitation, I employed transactional memory [55], a technique that has been used for writing concurrent code, to tailor the message processing process in the actor model. Transactional memory has thus far been used for writing scalable shared memory concurrent code as well as provides safety guarantees for concurrent applications. The approach is designed to operate with message passing and shared memory, and can take advantage of parallelism available on multi-core systems. Specifically, transactional memory is used here to safely process multiple messages in a single actor without breaking the actor semantics. The processing of each message is wrapped in a transaction executed atomically and in isolation, but concurrently with other messages. This allows us (1) to scale while keeping the dependability guarantees ensured by sequential message processing, and (2) to further increase robustness of the actor model against threats due to the rollback ability that comes for free with transactional processing of messages. We validated the design within the Scala programming language and the Akka framework. We show that the overhead of using transactions is hidden by the improved message processing throughput, thus leading to an overall performance gain.

Message queuing delay during coordinated actions

During coordinated actions spawning multiple actors, actors that have completed executing might be blocked, leading to message queuing delay. I argue that the blocking of actors, i.e., the coordination delay, might be not necessary in some cases and can be hidden. Therefore, I claim that speculation, with the help of transactional memory, can significantly improve actors's message processing performance. I show that if speculatively processed messages are independent, they can commit immediately, i.e., be processed without needing to wait for the coordinated transaction to commit. If there are conflicts between the state accessed by the coordinated transaction and the speculatively processed message (e.g., a write-write conflict), the processing of the speculatively processed message rolls back and only then blocks its processing and the processing of all other messages until the coordinated transaction commits.

Performance problems for concurrent message processing during high contention

As was shown above, concurrent message processing can be implemented with the help of transactional memory, ensuring sequential processing, when required. This approach is advantageous in low contention phases, however, does not scale for high contention phases. Didona et al. [33] demonstrated that multi-thread applications have an optimal level of concurrency (i.e., an optimal number of threads executing during application execution). The optimal level of concurrency can change over time and can be adjusted to get the best overall performance. The work by Didona et al. provides a good basis for addressing performance problems during high contention phases. It adapts one aspect, the concurrency (the number of threads) used for the application, in order to provide high performance (high transaction

throughput). In the direction of this work, in the actor model I dynamically adapt the number of concurrent message processing in a single actor according to contention levels. Second, I extract read-only message processing from the transactional context. And third, I exploit the fact that the two types of messages do not interfere and occupy the existing threads with both types of messages according to the current contention level. As a result, this approach occupies all threads with work. We demonstrate that we can substantially reduce the execution time of high-contention workloads in a micro-benchmark as well as in a real-world application.

The inability of the actor model to seamlessly exploit GPU resources
Modern CPUs, along with regular CPU cores, have an integrated GPU. HPC data centers have also started to offer GPU acceleration for a portion of their compute infrastructure. As noted, significant effort needs to be invested for exploiting GPU resources in the actor model. I use the optimized actor model, as discussed in the previous contributions, as a basis for efficient and simple programming for heterogeneous computing. Specifically, I target seamless development of data parallelizable iterative applications. I do so by introducing heterogeneous actors, which can be programmed domain-specific languages (DSLs) that provide automatic code generation for both CPU and GPU code. With heterogeneous actors, programmers get the capacity to decide on a fine-grained level whether a task, encapsulated in an actor, should execute on a CPU or on the GPU. The heterogeneous actor model also features support for workload-aware techniques. With our proposed system we are able to increase programmability as well as save 40-80% of energy in comparison to CPU-only applications.

As noted earlier, to reduce the energy consumption one can either radically reduce the running time or the power consumption, or both at the same time. As such, I use *running time* and, when possible, *power*, and resulting *energy consumption* reductions as the key indicators for evaluating the contributions of this work.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows: In Chapter 2, I present the background on energy efficiency, multicore and GPU programming, the actor model and the ParaDIME project. In Chapter 3, I present related work. In Chapter 4, I show how the delay associated with coordinated actions can be hidden as well as propose improvements to sequential message processing. The dynamic adaptation of the number of messages processed in a given actor is presented in Chapter 5. In Chapter 6, I introduce the heterogeneous actor model. Conclusions and future work are presented in Chapter 7.

Chapter 2

Background

In this chapter, I present the background work which we use for conducting research in this thesis. Since we target saving energy, I cover the basics of energy efficiency and elaborate on why it is important and how it can be achieved. In addition, given that I target retrofitting programming models, I cover the basics of programming multi-core systems. I also discuss how GPU resources could be used in the context of programming heterogeneous systems. After, I discuss the benefits of using message passing models for programming complex systems. I then advance to discussing the ParaDIME stack and elaborate on how it combines various techniques for energy efficiency.

2.1 Energy Efficiency

Energy efficiency is a way of managing the growth in energy consumption. For computing, energy efficiency targets the maximize computations per kilowatt-hour. Over the past half century the steps taken to increase the performance of computers inherently lead to the increase of their energy efficiency. According to Koomey et al [71], the computations per kilowatt-hour have doubled every 1.5 years.

The reduction of transistor size that targeted speed improvements also led to implicitly reducing per-transistor power requirements [16]. Power consumption is, however, driven not only by the transistor size alone, but also by other server components, such as disk drives, memory and network equipment. As such, rather focusing on specific components in the system, efforts targeted at energy efficiency should be holistic, i.e., focusing on a compute systems as a whole.

The growing popularity of cloud computing has greatly increased the scale of data centers resulting in significant increase in power dissipation. Hence, the increasing power and energy consumption of computing nodes that are appearing on the market today might hinder the energy efficiency and performance efficiency of data centers comprising these computing nodes. The major factor which inhibits energy efficiency for cloud computing is the underutilisation of resources. In a typical setting, servers are fully utilized only 5 to 15% of the time [67] and are not energy-proportional. Energy-proportional here means that the power required by a computing system is directly proportional to the work done. A standard commodity server is typically not energy-proportional. A typical server can draw from 40 to 120 W at 0% utilization. An energy-proportional server would draw 0 W at 0% utilization, meaning that the power drawn would increase linearly with utilization. A server with close to 0% utilization can be switched off, while the work is taken over by the remaining servers [70].

Energy efficiency also targets to reduce the costs associated with datacenter cooling. There exists a metric for rating the data center energy efficiency called the **power usage**

effectiveness (PUE) [18]. PUE reflects the ratio of total amount of energy used by a computer data center facility to the energy delivered to computing equipment. For the typical server the PUE is 2.0, which, stated simply, means that for every watt of power consumed by the computing equipment, another watt is used for cooling as well as powering other infrastructure equipment. Recent estimates suggest that a power-inefficient server at average utilization over the period of four years could cost more in electricity than the original price of the server [51].

To deal with extensive heat, companies such as Cloud and Heat (C&H) [41] started using alternative approaches for heat management. C&H's business model is based on a distributed infrastructure for cloud services. The energy-efficient aspect of this model is to co-locate the data-centers with facilities where the heat waste of server components can be used. That is possible in private homes as well as in offices or hotels. Running jobs on infrastructure that was especially built to dissipate heat of computers into warm water storage tanks means already a tremendous increase of the energy efficiency of the data-center. Depending on the chosen product the PUE reaches a minimum of 1.05 [35]. That means the relative energy that is needed by peripheral devices, such as pumps and cooling equipment, to operate the data-center is close to zero.

Lastly, energy efficiency is less important if sufficient cheap and emission-free energy sources are available. Because energy is a growing cost factor for data center operators, reducing the overall consumption in turn reduces the overall operating expenditures [114]. Coupled with penalties for carbon emissions the urge to cut energy consumption is even stronger. If, however, a cheap and "green" energy source is available, the overall consumption may suddenly be secondary [15]. When data centers have access to alternative energy sources, such as solar and coal, the question of where to process a task is then also dependent on where energy is cheap, plentiful, and green.

2.2 Parallel (Multi-core) and Heterogeneous Systems

Traditionally to increase the speed of processors, CPU manufactures have been increasing the CPU clock rate [101]. This process, however, came to a halt at the time when the power requirements of new, higher clocked CPUs became too prohibitive [106]. Consequently other design ideas had to be incorporated in order to increase performance without increasing the CPU's clock rate. Unlike old designs, new chips started to house multiple processing units (cores) as well as integrated GPU's. While significant challenges need to be dealt with when programming symmetric multicore systems, the move from symmetric to asymmetric multicore designs presents further challenges for programmability and seamless scalability.

2.2.1 Amdahl's and Gustafson's Laws

Adding multiple processors does not bring much gain for applications that require significant parts to run sequentially, i.e., on a single processor [9]. Commonly other cores have to stall, waiting for that one core to finish. Indeed, Amdahl's law postulates [7], that the scalability of any algorithm is limited by the parts of it that have to run sequentially. Formally, Amdahl's law can be expressed as follows. Assume that $T(n)$ is the time, which is required for the application to be executed on n cores. Further assume that a program P has sequential — β, $0 \leq \beta < 1$, and parallel — $1 - \beta$ parts.

Then the speedup $S(P)$ can be expressed as

$$S(P) = \frac{n}{\beta * n + (1 - \beta)}. \tag{2.1}$$

Gustafson's Law [48], however postulates, that the more resources a given system has, the larger the problems it can solve given the same time. It can be expressed as

$$S(P) = n - (n - 1)\beta \tag{2.2}$$

Although Amdahl's Law and Gustafson's law are both used to predict speed-ups, there is a fundamental difference between them. Amdahl's law states that we cannot run a particular problem faster given a predefined problem size. On the other hand, Gustafson's law fixes the total execution time, stating that we can solve problems bigger in size using faster hardware, for example. As such, Gustafson's law works very well for algorithms having to process large problems, such as huge dataset processing tasks.

2.2.2 Programming Multi-core Systems

In multi-core programming a thread is an execution context, which contains an independent set of values for the processor registers that are needed for executing a stream of instructions [101]. The set of values includes the *Instruction Pointer* as well as the *Stack Pointer*. This allows threads to be executing different code in different parts of the program. Threads are scheduled according to a predefined scheduling algorithm. Executing many threads simultaneously is akin to running a number of different programs concurrently, although threads have a number of compelling benefits. The main benefit is in that threads running in one process share the same data space (shared memory), which allows them to communicate through shared memory easier than if they were separate processes. Threads have a number of internal running states, e.g., they can be temporary put to sleep or pre-empted. The thread may be put to sleep manually by the programmer. Pre-emption is guided by the OS, which dynamically matches threads to cores, pausing threads when their turn is up and possibly starting them on different cores next time they are scheduled [30, 79].

Since all threads have access to some shared memory, their activity needs to be coordinated, so as to not corrupt the available data. To do so, programming environments support the concept of a **critical section**, which provides the capacity to limit the execution of a particular section of code by one thread at the same time. Special programming constructs — locks — are used to guarantee mutual exclusion. Locks guarantee that actions inside the critical sections, which they protect, are done indivisibly or atomically. Below a rudimentary code snippet that features how to use locks with POSIX Threads is presented.

```
pthread_mutex_lock(&lock);
 MakePizza();
pthread_mutex_unlock(&lock);
```

Given two threads in the system, if the first thread is executing the **MakePizza()** function and the second thread tries to enter the critical section by calling **pthread_mutex_lock()** at the same time, that call will block the second thread before the first thread releases the lock by calling **pthread_mutex_unlock()** [110].

2.2.2.1 Transactional Memory: An Alternative to Locks

Although locks are effective in achieving mutual exclusion, they can be too restrictive in some contexts, as, by definition, the code which is protected by the lock is execution serially instead of running in parallel. Programmers need to carefully consider where to insert the locking code so as not to degrade their codes's scalability. Locks also may suffer from (1) races, due to forgotten locks, (2) deadlocks, in the case when locks were acquired in the wrong order, and (3) problematic error recovery, for which the invariants and lock releases need to be provided in exception handlers. Further, lock-based code does not compose [52].

Transactional memory(TM) [55], has been proposed to improve programmability and scalability of multi-threaded code. TM offers an optimistic view on parallelization, assuming that contention will not occur when executing the critical section code by more then one thread concurrently.

In TM a **transaction** is a sequence of memory operations that must be executed atomically or not be executed at all from the perspective of other threads. At commit time a validation step must detect if other threads have written to the memory which was used by the transaction, or, vice versa, if other threads have used the memory written by the transaction. When a transaction finishes, it either exports all the changes made as part of the transaction to memory committing the changes to memory, or throws away the changes and aborts the transaction. To deal with aborts transactions may decide to roll back and re-execute or fall-back to using locks for protecting the critical section. Similarly to locks, critical sections in TM are guarded by TM-specific constructs. Below a rudimentary example of how to use transactions is presented.

```
atomic {
 ReadPizzaCount();
}
```

The promise of transactions is that the user could simply write sequential code inside the **atomic** block, with TM guaranteeing its save execution in multiple threads.

2.3 GPU Computing

Both, the CPUs and the GPUs have their unique features and strengths. The CPU excels at tackling tasks-parallel as well as irregular problems, having unpredictable memory access patterns [44]. The GPU computing, on the other hand, embraces the Single Instruction, Multiple Threads (SIMT) programming model, which allows obtaining exceptional performance for data-parallel applications. SIMT advances over the older Single Instruction, Multiple Data (SIMD) programming model in that the programmers no longer need to write code, in which every thread follows the same execution path [98].

The main difference between GPUs and CPUs is that the former is optimized for throughput and the latter is optimized for latency [74]. Also, GPUs contain substantially more computing cores than the latter. As an example, CPU's normally contain 2-8 cores capable of executing 4-16 hardware threads with hyper-threading enabled, with each of the cores featuring out-of-order execution and branch prediction, among other advanced features. GPUs are composed of, on average, 16 Streaming Multiprocessors (SMs) units, each of which containing 48 resident warps[1], with each warp capable of executing 32 threads. This setup

[1] A set of threads that execute the same instruction

allows for 24,576 threads to be active at the same time, since all SMs are always active, and each SM can have many active threads at the same time.

Despite having an abundance of threads in availability, GPU threads operate in a more restrictive manner than CPU threads [20]. Recall, that CPU threads can execute different code in different parts of the program. This is, however, not the case of GPUs. Inside an SM threads run in lockstep, executing the same pice of code (potentially on different portions of data) at the same time. This behavior can be problematic for code having branch instructions, as the threads, for which the branch condition does not hold, are required to stall. Another major difference between the CPU and the GPU lies in how the cache memory is managed. The CPU automatically manages which data have to stay in the multiple levels of cache through a cache coherency protocol managed in hardware. In such a design cache memory is not meant to be not used by the programmer directly. Conversely, the GPU gives the programmer direct control over what data is to be stored in the cache as well as when that data are to be evicted.

When executing a hybrid CPU-GPU application, the CPU offloads the compute-intensive part of the application to the GPU, while running the CPU-specific parts of the code along with the GPU. Once the computation on the GPU has completed, the CPU copies the results from the GPU to the CPU memory. Programmers also need to take into account that the memory copy time can be significant for some applications, and it needs to be amortized by the gains of executing on the GPU.

While offloading work to the GPU and waiting for the results to be available on the CPU is the most common way of executing GPU programs, it is rather wasteful with respect to the CPU resources. The CPU has to stay idle until the results are available, which leads to potential wastage of energy. As such, when programming heterogeneous systems one needs to carefully schedule work such that the idle time of either of the processing units is kept to a minimum.

2.4 Programming Models and Actors

The paradigm shift from centralized and serial to distributed and concurrent computing, introduced the need for models that fulfill the inherently concurrent nature of such systems [107]. One example of a model, which recently regained researcher's attention, is the actor model. Concurrency issues in thread-based systems are explicitly handled by the application layer what appears to be complex and error prone. The actor model, in comparison, implicitly avoids possibly arising concurrency issues such as race conditions, deadlocks, starvation or liveness. Its strict semantics introduce strong guarantees. Created by Carl Hewitt et al. [58] in the early seventies, the actor model was further formalized and theoretically refined by Grief [45], Hewitt et al. [57] and Agha [2].

In contrast to thread-based environments, concurrency in the actor model is achieved by using asynchronous message passing. Figure 2.1 illustrates the communication mechanism of the actor model.

Beside academic interests, the actor model meanwhile was implemented in a wide variety of languages (e.g., Erlang, Scala, AmbientTalk) and integrated with frameworks, such as the Akka[2] framework.

[2]`http://akka.io`

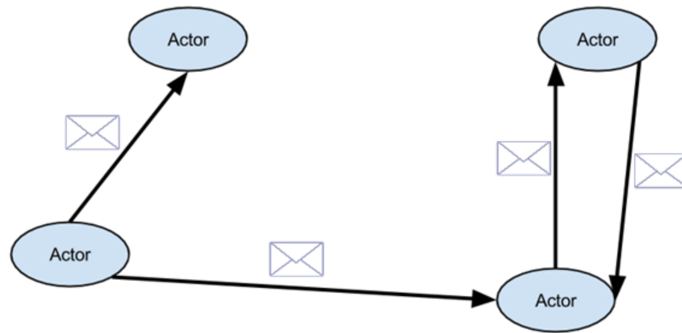

Figure 2.1: Actor model: Communication through messages.

2.4.1 Actors as the Main Entities in the Actor Model

Actors as the main entities of the actor model follow the *'everything is an actor'* credo, similar to the *'everything is an object'* paradigm [82]. They have a restricted set of fundamental capabilities:

- send a finite amount of messages to other actors
- receive a finite amount of messages from other actors
- spawn new actors
- change their internal state (become another actor)

Actor model's most important characteristics are also its strongest limitations: strong encapsulation and locality. An actor can be stateful but the actor model removes the feasibility to access another actor's internal state. Although an actor can expose information about its state by appropriate messages, there exists no shared mutable state. In general, actors can be viewed as black-boxes, having references to other actor instances only. This concept of strong encapsulation inherently creates mutual exclusion without synchronization and locking. As a consequence, an actor is constrained to send messages to already known addresses from incoming messages or spawned actors only, which is referred to as locality [57].

2.4.2 Asynchronous Message Passing

Communication between actors is handled by asynchronous message passing [58]. A message consists of an actor's address (or identifier), the return address and the data (sometimes referred to as *message payload*). Depending on the actor model implementation, an actor determines appropriate behavior based on the type of the incoming message.

Actors store received messages in a unique storage facility known as a mailbox. The actor model ensures fairness by providing weak message delivery guarantees, meaning that every message gets eventually delivered. In Object-Oriented Programming the invocation of a method on an object can lock the invoking entity. The emission of messages in the actor model happens asynchronously and does not block the sender. The non blocking communication enables the actors to work in parallel and avoids the locking of system resources [68] (e.g. a system stack).

An actor is not required to process messages in any particular order (what Hewitt et al. refer to as *particular order of events* [57]). The absence of constraints on the order of message delivery corresponds to non-determinism, an essential property of parallel computing.

Nevertheless, an actor is only permitted to process one message at a time, a decision that was made upon the introduction of stateful actors [118]. Allowing two messages or more messages to be processed in parallel leads to concurrent access to the internal state of the actors. While this limitation guarantees the absence of deadlocks and data races, it puts a serious limitation on task level parallelism. Also, the absence of shared mutable state requires libraries implementing the actor model to support call-by-value semantics [68].

2.4.3 The Akka Framework

The Akka framework provides interfaces in Scala and Java, claiming in its official documentation that it offers many advantages over its immediate competitors [66]. Another advantage of actors implemented in Akka is its capacity to easily integrate with existing projects. In Akka the execution of an actor is in part performed by using a Java Fork/Join thread pool introduced in Java 7, implying that to scale Akka's actors use threads for their execution. Erlang on the other hand does not use threads for executing actors. The actors in Erlang are lightweight processes (LWP) which are executed in Erlang's Virtual Machine (VM). Unlike Java threads which are mapped to operating system's threads, LWP are implemented inside Erlang's VM. The syntax which use Akka to manipulate actors is borrowed from Erlang.

As Akka is based on Scala, the programmers can make use of all of the rich features available in Scala. For example, Scala empowers Akka to provide advanced pattern matching techniques, which supports using regular expressions to match incoming messages with their specific processing logic. The next code shows an Akka program which instantiates an *actor A* and sends a message to it. When the actor receives a message, it prints "Hello World".

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloWorldActor extends Actor {
 def receive = {
   case _ => println("Hello world")
 }
}

object Main extends App {
 val system = ActorSystem("HelloWorldSystem")
 val helloworldActor = system.actorOf(Props[HelloWorldActor])
 helloworldActor ! "sayhello"
}
```

We first import the necessary libraries. Then we create a class, defining an actor as an object, calling a set of methods that perform the initial bootstrapping of the system. There are two main blocks the class *HelloWorldActor* and the *Main* object. The *HelloWorldActor* class declaration is the actor's definition, which contains the receive method that is similar to the one used in the Erlang's code. The receive method handles incoming messages, and in this case it prints "Hello world" to the console when the message has been received.

The Main object creates an *ActorSystem* object, and finally sends a message to the created actor. The *ActorSystem* object is used for interacting with the Akka framework. The syntax for sending a message is similar to that of Erlang — the messages are sent using the "!" operator.

2.4.3.1 Akka: Mailboxes

The messages received from other actors are stored in actor mailboxes [5]. Since many messages can be received at the same time, Akka mailboxes are backed internally by a thread-safe data structure. Akka provides different implementations of actor mailboxes out of the box, each of which uses a different variant of a concurrent queue from the standard *java.util.concurrent* library.[3]

The implemented mailboxes differ in the way the queue management is performed. Akka mailboxes support bounded queues and blocking queues. The semantics of the blocking queue are as follows: The enqueue operation is blocked until the queue has space for the element to be inserted. Similarly, the dequeue operation will be blocked until there is an element available to be dequeued. The semantics of the bounded queue are as follows: the queue is first initialized with an initial capacity, which cannot be subsequently changed. Once the queue becomes full, meaning that it contains the same number of elements as its capacity, and any attempts to enqueue additional elements will block the actor [47].

Akka uses a blocking unbounded queue by default. Similarly to Erlang, in Akka you can wrap and send any object as a message. Also, Akka does not have the capacity to enforce immutability and relies on the developer to support the immutability of data transmitted in a message. In the actor model it is important to use immutable objects when exchanging messages because it prevents the state of the actor from being modified by external threads [66]. Scala supports reference immutability by prefixing the variable definition with a *val* (e.g., *val myvalue=11*). The user has to be aware that an immutable variable in Scala/Akka can refer to a mutable object, implying that the object's state can be corrupted by other threads in the future.

Akka provides a number of mechanisms for exchanging messages between actors, which include the default "!" operator, also available in Erlang. This operator allows to send an asynchronously message to an actor without waiting for a response from the receiver. Akka also offers a concurrency mechanism called a *Future*. That mechanism allows an *actor A* to send a message to another *actor B* and enforce *actor A* to wait until *actor B* has sent a response. The response from *actor B* is encapsulated in a *Future* object, which contain methods for checking whether the response has been received already, as well as contains blocking methods for waiting for the response to be available. More specifically, a *Future* is a mechanism which is used to retrieve the result of some operation that is performed concurrently by a different process, at a later time [47]. The next code demonstrates how the *Future* mechanism is used.

```
implicit val timeout = Timeout(5 seconds)
val future = actor ? msg
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

The message is sent using the "?" (line 2) method which returns a *Future* object. To obtain the result, the *Await.result* method is used, which blocks the actor while the response is not available. The actor will continue its execution until it receives the result or the defined timeout (line 1) expires.

[3] http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

2.5 The ParaDIME Stack

In this section, we give a brief overview the Parallel Distributed Infrastructure for Minimization of Energy (ParaDIME) architecture [84], in the context of which this work was carried out. Figure 2.2 presents the ParaDIME stack, which combines several energy minimization methodologies. ParaDIME is based on three main mechanisms: (1) message passing, (2) operation below safe V_{dd}, and (3) approximate computing. The described mechanisms can yield significant savings if they are combined, especially because a single mechanism might only get efficient in non-realistic settings. For instance, aggressive voltage scaling increases the error probability dramatically, leading to high costs for error detection. Parallelization brings important overheads because more processors have to be powered, but they can be limited when used in combination with smart scheduling and voltage scaling.

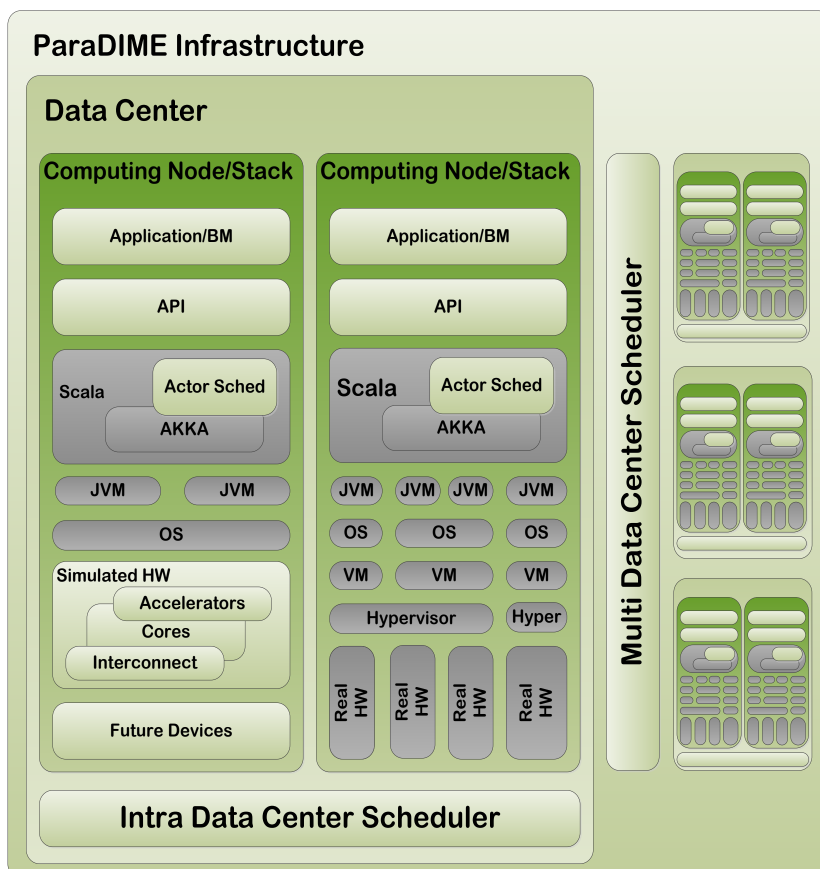

Figure 2.2: The ParaDIME stack overview [84].

2.5.1 Efficient Message Passing

Programming environments for current multi-core architectures largely rely on a shared-memory model. This approach is, however, only effective up to a certain number of cores, typically 8–16 cores on 1–4 CPU sockets. Beyond this threshold, contention on shared resources (memory, bus, caches) sometimes degenerate to that of sequential execution as accesses to shared data end up being serialized. In such settings, we need a more appropriate programming model with lower synchronization overheads, such as message-passing, which can scale better on large numbers of processors [100, 113].

The additional advantage of message passing is that different processors involved in a computation are decoupled. This can be used to increase the utilization of the involved machines, which reduces the cost of computation and leads to higher efficiency at the data-center level. It also provides the support interacting with the hardware to provide information that can be used to apply aggressive energy saving techniques.

As such, ParaDIME opted to use the Akka framework, a library with actor model support. Akka runs on top of Scala, a general-purpose language, that runs on top of the JVM and combines functional and object-oriented programming patterns.

2.5.2 Multi-data Center Scheduler

In the context of ParaDIME, energy-efficient processors have been examined as key components of the future computing node. The ParaDIME stack also focused on the energy demands of the data centers in the context of the developed programming model and, specifically, concentrated at the individual and multiple data center environments. At the multiple data center level, a multi-data center scheduler has been developed that permits scheduling computations on one out of many data centers available. To enable green computing, the scheduler takes into account the CO2 emissions of each data center, looking for the best possible match between data center workloads and heating/cooling necessities [1].

At the level of individual data centers, the intra data center scheduler distributes the required resources onto the available servers. The created scheduler incorporates a number of independently created technologies that were created for the ParaDIME stack to account workload variability. The scheduler can be activated “just in time”, reducing the time that is required to reactivate a sleeping virtual machine and the time and network traffic to migrate virtual machines. In addition, to enable energy-efficient storage, techniques have been proposed to swiftly turn off idle disks [70].

2.5.3 Operation Below Safe V_{dd}

Due to the recent issues impacting device scaling as we approach the end-of-the-CMOS-roadmap, safe operation margins have been increasing. In particular, there is a substantial “tax” in the case of guard-bands for supply voltage [11]. Reducing the supply voltage of a circuit (Vdd) is a well-known technique for making trade-offs between performance and power [31]. ParaDIME, however, claims that aggressive voltage scaling is only possible by operating the CPU outside of the manufacturer’s specified safety margin. As such, ParaDIME investigates techniques to lower the Vdd below the safe limits that results in an overall reduction in energy consumption. For example, redundancy and full replication techniques can be used to detect and correct errors when running below safe V_{dd} [116].

2.5.4 Approximate Computing

Studies have shown that between 50% and 70% of the values in the processor are narrow [36, 42], depending on the architecture, the application, and the number of bits needed to consider a value narrow. This means that the processor does not need to load, store, or compute the upper bits of an integer value if it has been identified as a narrow value. Under circumstances when maximum precision is not required a similar approach can be used for floating point values. In this case the least significant bits of the mantissa are not relevant. One of the most interesting points is that errors occurring in these bits are not relevant. This means that

we can use approximate computing to reduce the error detection overhead when lowering voltages. Successful combination with voltage reduction, but without any additional error handling, has been shown by Sampson et al. [95].

As such, ParaDIME defines a number of profiles for interacting with the hardware related to lowering the Vdd. For this, low-level annotations are used for indicating data types with reduced precision, and safe regions for operations below safe Vdd. For low-level annotations ParaDIME uses Scala annotation macros, which allow to define annotations and with them linked macros that will be executed at compile time [91].

Chapter 3

Related Work

There exists a broad body of literature which covers techniques for supporting energy efficiency. Much work exists also in the area of improving the message passing models. Most of the techniques for improving the message passing models, however, do not consider energy efficiency. In the next sections, I present the most relevant work that has been conducted in the area of optimizing the actor model, as well as in the area of support heterogeneous execution of tasks.

3.1 Concurrency and Speculation

The actor model introduces several desirable characteristics, with one example being its abstraction of concurrency and strong guarantees for concurrent systems. In essence, the model consists of two concepts: the actor entities and asynchronous message passing. Actors are self-contained, strongly encapsulating entities. They communicate via asynchronous message passing under strong semantics as formalized by Agha [3], Clinger [28] and Grief [45]. In addition, to guarantee data consistency actors are permitted to process only one message at a time. The advantages provided by the model introduce a trade-off between data-race-freedom and efficiency caused by the sequential message processing inside an actor [118].

3.1.1 Alternative Models and Frameworks

There exist other models, which base their execution on message passing. The active objects model is an example of such a model. The active objects model decouples method execution from method invocation in order to simplify synchronizing access to the object. Although being similar to the actor model, the active objects model is fundamentally different [93] in that it works with acknowledgments (similar to TCP acknowledgments), which can degrade performance [73].

Existing actor frameworks such as surveyed by Karmani et al. [68] differ regarding the way they handle parallelism in general. There have been attempts to use data partitioning [32], for example, to achieve the desired task level parallelism without violating the actor model. This strategy enables the delegation of data-coupled tasks to new actors. An actor therefore is responsible for creating data chunks of incoming or contained data and for provisioning them together with the necessary parameters to the spawned actors upon initialization. Using this method an actor can implement a *divide-and-conquer* strategy where spawned actors compute partial results on a chunk of data.

Since new programming languages, such as Scala, Clojure, and F# provide multiple concurrency abstractions, in their work Swalens et al. [108] investigate the feasibility of

combining these abstractions for managing concurrency. Specifically, the authors considered the following abstractions: atomics, STM, futures and promises, as well as alternative approaches for communicating threads. Since the combinations of different concurrency abstractions can introduce deadlock and/or new livelocks, as well as introduce race conditions by causing unexpected interleaving, the main question that the authors targeted to answer was whether different concurrency mechanisms were composable. All considered mechanisms were looked at in the context of developing a modern email application. While this research looks into ways of *explicitly* interacting with multiple concurrency mechanisms, we target to *implicitly* use one concurrency mechanism (i.e., transactional memory) in the context of another (i.e., the actor model). As such, we use both of these mechanisms in a controlled manner. In our setting the user is not responsible for integrating these mechanisms, which reduces the possibility of introducing bugs. Moreover, when running an application written in the actor model in our environment, the user might not be aware that transactional memory (TM) is used under the hood, since we fully preserve the semantics of the actor model.

Noël [83] created XSTM, a new extensible programming model which, as claimed by its author, combines all the benefits of shared memory programming, supported by transaction memory, and of the actor model. The transaction memory programming model is used in this research because it provides atomicity, isolation, and rollback capabilities within transactional code regions [53]. In the model, the programmer simply has to demarcate the blocks of instructions that must execute atomically and the TM performs all the necessary bookkeeping to ensure that the target code is executed in a transaction, i.e., the consistency of data accessed within the block is not affected by concurrent accesses. In the resulting model shared memory programming is used to avoid copying memory between components and the actor model is used for offering safety and composability. In XSTM, all the shared data are immutable. This feature is very similar to the requirements of actor model, which dictate that all the messages exchanged by actors need to be immutable. Data can be only mutated by the 'owning' thread, which is also similar to the actor model, in which data can only be modified in the actor's local state. This design decision allows no synchronization to take place during data modification. XSTM also works similarly to source code version control systems — at transaction start the system takes a snapshot of the data, performs the required work using that data, and then merges the results with the repository. As such, XSTM reassembles a multi-version STM and not a message passing system. Despite the similarities between this research and our research (i.e., both use concepts borrowed from the actor model and transactional memory), our research takes the actor model as a base and optimizes it using transactional memory, and this research creates a new transaction memory programming model, borrowing concepts from the actor model for its efficient execution.

3.1.2 Actor Model Enhancements

Since the actor model does not permit the parent actor and the spawned actors to have access to the same data, the parent actor has to create a copy of that data before passing it to the child. In this case actor model semantics can be fully preserved, but partitioning and copying of the data are expensive, as shown by Scholliers et al. [97], cause overheads. Furthermore, executing global actions on the distributed data chunks is accompanied with a high degree of overhead associated with controlling and coordination. Examples of global actions are search operations over several actors or data manipulations, that require other actors to update their part of the data. Scholliers et al. also highlight the resulting coupling between data and computation. The authors also propose parallel actor monitors (PAM) to

support concurrent processing by scheduling multiple messages in actor queues. Using PAM, the programmer must understand the concurrency patterns within the application and define application-specific schedulers. This may prove particularly challenging for applications where concurrency patterns vary during execution. In contrast, our approach (see Section 4.3), which automatically enables the concurrent processing of messages, removes any programmer intervention. Further, we do not break the original actor semantics at any time, while using an inappropriate scheduler with PAM can cause inconsistencies.

The Habanero-Scala framework [65] introduces parallelism by mixing the actor model with a fork-join design (**async-finish** model). Actors can start concurrent sub-tasks (**async** blocks) for the handling of a single message. Since the processing of a message terminates only when all sub-tasks are finished, this approach introduces a significant processing bottleneck. To alleviate this restriction and improve scalability, Habanero also allows a message to be processed in parallel in a single actor. To ensure that the actor's state is not accessed concurrently, a **pause and resume** model that works similar to **wait and notify** is used. While processing a message, the actor can spawn external sub-tasks it must then pause to avoid intermediate modification to its state. When the sub-tasks finish with changing the state, the actor resumes its operation and can process further messages. While this approach avoids concurrent access to an actor's state, it must be used carefully as it provides no protection against synchronization hazards such as data races and deadlocks. Although Habanero-Scala enhances the utilization of multi-core resources, it does not provide the support of executing multiple messages in parallel, which could further enhance resources utilization.

Azadbakht et al. [10] proposed an approach for processing actor messages with multiple threads. In each multi-threaded actor (MAC) there is a list of active objects which can all access the shared actor queue in order to process messages from that queue concurrently. While in our research we use transaction memory to protect the shared state of an actor, MACs use *locks* on method signatures to protect shared state. As such, even in a scenario where the majority of operations are reads this approach cannot scale, since regardless of the operation type we need to obtain a lock. On the contrary, TM provides us with more opportunities for scalability by trying to execute *all* operations concurrently and roll back only when a conflict occurs. In addition, the MAC framework attempts to specify the locks used in the system such that they enforce temporal properties between different method invocations. As was elaborated earlier, the actor model does not enforce any particular order on message processing, and the attempt to enforce message order might hinder scalability. In our research TM was used precisely because the actor model does not guarantee any order of message processing, which allows to roll back and restart message processing, without being forced to stop all other message processing. In summary, while this research also attempts to process actor messages concurrently, it uses locks (and not TM) for guaranteeing correctness.

To the best of our knowledge, the coordinated processing of messages is only supported by the Akka Framework [49] for realizing actors, as integrated in the Scala language. Coordinated processing in Akka is supported by transactional memory implementation in Scala [43] We use TM not only in the context of coordinated message processing, but also for providing the support for concurrent message processing as proposed in this thesis. TM provides built-in support for check-pointing and rollback, which are instrumental for controlling the concurrent message processing process.

3.2 Dynamic Scheduling

To the best of our knowledge, none of the approaches thus far provide different levels of concurrency in the context of message passing models. However, in the context of concurrent programming using threads, a number of works have emerged for improving system throughput with the help of dynamic scheduling.

3.2.1 Contention Management

To optimally use the resources and to improve performance, researchers proposed several mechanisms to match the level of concurrency with the current workload. For example, contention managers [56] have been proposed for enhancing performance by eagerly detecting contention and handing it according to predefined rules. The proposed **DSTM** framework supports the *early release* facility, which permits to reduce conflicts. Once a piece of code calls the *release()* method, the transaction, in which the method was called, will not conflict with other transactions accessing the same object. *Early release* could be only called for objects accessed in `READ` mode. If an object is promoted from the `READ` mode to the `WRITE` mode, all calls to *release()* are silently ignored. This feature is particularly dangerous because, if it is used incorrectly, it can potentially violate the linearizability of the releasing transaction. Despite the warning, the authors provide only one example of the correct usage of this function in the context of list traversal. They do not elaborate on other contexts, in which the usage of *early release* would be not appropriate. The *early release* facility is also, however, only partly realized in the Scala STM (based on CCSTM [22]) that we use for providing transactional support.

Yoo et al. [119] claim that when the workload to be executed does not have inherent parallelism, starting too many TM transactions could negatively affect performance. As a base for increasing performance, they use the concept of contention managers. Vanilla contention managers were identified to be limited, however, in that they could only take action at the time when a conflict is detected. They also get called frequently when contention is high. As such, the developed adaptive transaction scheduling (ATS) technique selectively schedules transactions that tend to abort frequently. It also varies the number of threads executing transactional code based on contention feedback. The schedule module is only accessed when the transaction is started in a high contention scenario. ATS, however, depends on a single centralized queue of transaction. Since all threads access the queue, it might induce more contention in scenarios where many threads are used.

Thread Reinforcer [88] is a system developed for determining dynamically the appropriate number of threads for a multithreaded application. To understand the speedup behavior of all studies application the authors monitor all Operating System level factors that could potentially explain the variation of the speedups with the change of thread count. Specifically, the following Operating System factors are considered: `LOCK` (lock contention), `VCX_RATE` (voluntary context switch rate) `MIGR_RATE` (thread migration rate) and `CPU_UTIL` (processor utilization). The authors also considered two categories of application execution: one, where the thread count is set below CPU core count, the another, where thread count is set above CPU core count. For both of these cases, `LOCK` and `CPU_UTIL` are considered to be important parameters. The `VCX_RATE` parameter is only important for the case when thread count is set below CPU core count. On the other hand, `MIGR_RATE` is important for the case when thread count is set above CPU core count. The Thread Reinforcer Framework framework is executed in two steps. First, the application is executed multiple times using

small problems or using small datasets. Based on the observations obtained at runtime, the framework determines the appropriate number of threads. In the second step the application is executed again with the desired workload using the obtained number of threads. This research, on the other hand, proposes a system that is capable to monitor and adjust the applications execution **online** and, as such, vary the number of threads dynamically during the application execution.

Didona et al. [33] propose to dynamically adjust the level of concurrency according to the number of TM commits and aborts. The optimal number of threads is found with the help of two phases: (1) a measurement phase and (2) a decision phase. In the first phase, the application is profiled with a fixed number of threads, in the second phase, a hill-climbing approach is applied to increase and decrease the number of threads according to the given workload, maximizing the transaction's throughput based on successful commits. While the basic idea is interesting, the variation of the thread count based on the throughput might be disadvantageous for transactions with different granularity. In addition, the idle threads are not used for potentially useful operations.

Ansari et al. [8] also target the level of concurrency to enhance application efficiency. They do so by monitoring and adapting the concurrency level based on the predefined threshold that limits the transaction abort rate. In our approach, we not only consider the transaction abort rate, but also the commit rate to drive performance adaptation.

Finally, the idea behind automatic adaptation of the number of messages processed in a single actor corresponds to elastic scaling of the number of compute nodes that the distributed system is deployed onto. It is also similar to selecting the optimal number of nodes to be deployed in a replicated database [121]

3.2.2 Power-Aware Techniques

Petoumenos et al. [86] investigated all state-of-the art techniques for capping power consumption and systematically examines the advantages and disadvantages of each studied mechanism under similar hardware and software settings to draw conclusions about each method's effectiveness. This work considered the following approaches — DVFS, DFS, RAPL, thread packing, forced idleness, NOP insertion, and compiler-based power control. The authors highlight that while many of these approaches work effectively in the context, for which they have been developed, their joint usage has not yet been considered. The presented findings show that RAPL and DVFS should be used when the selected power limit is high, otherwise (i.e., when the power limit is low) it is best to use forced idleness and thread packing. On the other hand, Clock throttling and DFS techniques should be avoided, although they are being extensively used in literature. Clock throttling and DFS were also found unsuitable to be used in practice. The combination of RAPL with thread packing and forced idleness yielded the best results when used together in a coordinated way. Although the main goal of the research on power capping was to investigate methods to *limit* the power consumption, it provided a number of insight on how power power consumption could be not only limited but *altered* in the context of our research.

Suleman et al. [105] proposed a number of techniques for controlling the number of threads during the runtime of an application not only for increasing the performance but also for reducing the energy consumption. The authors note that simply setting the number of threads to the number of available CPU cores might not suffice, since contention on shared data might also increase, in turn worsening energy efficiency and power consumption. To that end, Feedback-Driven Threading (FDT) was proposed to dynamically control the number of

threads. FDT works by sampling a small number of parallel kernels during application to estimate at which thread count the performance of these kernels saturates. Specifically, FDT considers two parameters — data-synchronization and bus bandwidth to drive the thread prediction. Data-synchronization is important, since when we increase the number of threads, the time spent in the critical section tends to dominate the overall execution time. Off-chip bus bandwidth is also an important factor, since when it saturates, no further improvement in performance can take place, and, as such, increasing the thread count further can only harm performance and energy efficiency. FDT was then used as a base for powering Synchronization-Aware threading (SAT). SAT is capable to use the amount of data-synchronization (i.e., time spent in critical sections) as the main metric for controlling the optimal number of threads. In the same vein, and also based on FDT, Bandwidth-Aware Threading (BAT) was proposed for predicting the minimal number of threads such that they can saturate the off-chip bus. This method was mainly thought to be used with bandwidth-limited applications. The authors demonstrated that the combination of the aforementioned techniques can yield drastic energy savings as well as significant performance benefits. The usage of these methods may be, however, associated with non-trivial runtime overhead. On the other hand, the method to vary the thread count, as proposed by our research, assumes using only one extra thread to drive its execution, which would periodically gather TM rollback/commit statistics.

3.3 Heterogeneity

In general, research on hybrid computing rarely considers energy efficiency. Researchers focus more on performance improvements (e.g., [117]) or develop power estimation models [61, 69]. The trend of using graphical processing units (GPUs) for scientific programming became popular as there is a potential for significant performance improvement over executing only on a CPU because of the higher core counts. With the radical reduction of execution time GPUs can, in turn, reduce the total energy consumption, providing means for energy-efficient programming [64, 92]. Nevertheless, if the gains in execution time of GPU implementations are not high enough, the energy consumption might increase as compared to a CPU-only implementation [92].

The PEACH framework [40], for example, combines performance and power metrics to guide the scheduling on both CPU and GPU, but it focuses on defining a theoretical model rather than a practical implementation capable of working with real-world applications. Researchers working on SEEP [62] aim at helping programmers to produce energy-aware software. Their approach considers continuous energy monitoring of specific code paths helping to identify energy-hungry code. They mainly target, however, embedded systems capable of executing a single task. On the programming language level the authors of [39] divide a program into phases for which specific CPU frequencies are assigned. This approach does not only necessitate fine-grained monitoring of energy and execution time, but also requires that a program exclusively occupies a single core of a CPU. Hsu et al. [63] take a step further to create an enhanced compiler model that allows for dynamic frequency scaling. Li et al. [75] propose a hybrid OpenMP/MPI programming model for power-aware programming. The authors use this model to steer the level of parallelism as well as the current frequency of a CPU.

The SPRAT [109] environment can automatically select the proper execution processor (either CPU or GPU) at runtime for energy efficiency. Migration is, however, quite expensive as the current state of the application must be saved when moving from one processor to

another. This goal is hard to achieve as hybrid programming is still a matter of properly managing two types of processors. There are a number of approaches for scheduling work between the CPU and the GPU. They can be broadly divided into performance/cost models (e.g., HEFT [111]), offline training [76], as well as work stealing [38]. A performance/cost model requires determining the approximate runtimes and data transfer times beforehand for each processing unit. For developers this requirement is hard to meet. Offline training alone limits the scope of the system's expertise to input parameters or datasets seen during training. In contrast, work-stealing schedulers typically do not require having a priori runtime knowledge for driving application execution. As a result, we decided to experiment with work-stealing for efficient workload balancing.

Lastly, Becchi at al. [13] devised an efficient method for mapping running tasks to different types of processors. The authors experimented with policies for performing the task-processor mapping and provided an overview of the advantages and the limitations of all considered policies. The general idea was that, since the runtime behavior of a program varies during program execution, it might be of benefit to map a thread to a different (and potentially more efficient) processor for obtaining performance benefits. First, the authors started with a static policy, which assigned threads to processors according to a simple heuristic. The threads in this policy do not migrate during program execution. This policy was considered as a baseline. Then, a number of more advanced, i.e, **dynamic**, policies were proposed. The round robin policy dictates that the all jobs are rotated among available EV6 (i.e., more powerful) cores. Despite the dynamic nature of this policy, for driving the thread-core assignment it does not use the run-time information, such as instructions per cycle(IPC)). As such, the authors propose to use the novel IPC-driven dynamic assignment scheme, in which the threads' characteristics are being considered for the whole duration of the program execution to drive the thread to core mapping. Succinctly, this scheme tries to map threads to the cores, which would be able to fully exploit the architecture of the processor. This implies, that if a thread cannot fully exploit all the architectural benefits provided by the faster core, that thread would be moved to a slower core. Despite the dynamic nature of the proposed schemes, this research assumes using same kinds of processing units (albeit with different performance characteristics) and does not foresee exploiting GPU resources. As such, it is unclear if it is feasible to use the proposed approaches in the context of scheduling tasks between CPU and GPU resources. For example, it might be costly to move memory between the CPU and the GPU. It is also unclear how these costs could be calculated. Moreover, since this research is only considering using CPU resources and, as such, it does not look at the problem of the seamless definition of tasks, which could run on heterogeneous processors.

3.4 Summary

Only a few research studies consider in detail the problem of message processing in the actor model. Most of the proposed solutions targeting to enhance the message processing throughput are not practical, since they bring substantial overheads and hinder programmability [97]. In contrast to other work, the orientation of my work is in that it targets to optimize the message passing process by employing techniques and ideas from shared memory programming as well as strategies for facilitating seamless GPU programming.

While shared memory and GPU programming techniques and strategies initially were not necessarily designed to be applied in the context of message passing models, my goal is to demonstrate that they can be effectively applied in the actor model. Specifically, I use

transactional memory, a technique embraced by the shared memory programming community, to process messages concurrently. I then consider a dynamic approach for processing messages in transactions, inspired by the work of Didona et al. [33].

Lastly, I use the Delite [104] framework, a tool for simplifying GPU programming, to provide seamless messages processing facilitates in the actor model. In contrast to the key objectives of Delite, which are to enhance performance and programmability, I aim to demonstrate that my contributions bring enhanced energy efficiency, being one of the main targets of this thesis. As such, the additional orientation of my work lies not only in pursuing performance benefits, but considers energy efficiency and programability as first class citizens in the context of message passing models.

Chapter 4

Exploiting Concurrency and Speculation in the Actor Model

4.1 Introduction

The actor model, initially proposed by Hewitt [58], is a successful message-passing approach that has been integrated into popular frameworks [68]. The actor model introduces desirable properties such as encapsulation, fair scheduling, location transparency, and data consistency to the programmer. It also perfectly unifies concurrent and object-oriented programming. To simplify reasoning about concurrent systems, actors provide macro-step semantics [4], which dramatically reduce the number of possible computations and interleavings that would otherwise be required to be considered. While the data consistency property of the actor model is important for preserving application safety, it is too conservative in concurrent settings as it enforces sequential processing of messages, which limits throughput and hence scalability.

In this chapter, we address this limitation by proposing a mechanism to boost the performance of the actor model while being faithful to its semantics [68]. The key idea is to apply speculation, as provided by transactional memory (TM), to handle messages concurrently as if they were processed sequentially. The semantics of transactions and the macro-step semantics of actors are very similar. Multiple steps/operations are performed in a transaction and they need to appear as a single step. Coincidentally, that is exactly what the macro-step semantics dictate in the actor context. Also, in the actor model there is no restriction on the order of messages to be treated. The same goes for transactions — as long as there are no inconsistencies, transactions can be committed in any order. As a result, TM transactions provide the means of facilitating the macro-step semantics of the actor message treatment under concurrency. As such, in cases where these macro-step semantics might be violated, we rely on the rollback capabilities of TM to undo the operations potentially leading to inconsistencies.

We see a high potential for improvement in scenarios where actors maintain state that is read or manipulated by other actors via message passing. With sequential processing, access to the state will be suboptimal when operations do not conflict (e.g., modifications to disjoint parts of the state, multiple read operations). TM can guarantee safe concurrent access in most of these cases and can handle conflicting situations by aborting and restarting transactions.

Speculation can also significantly improve performance when the processing of a message causes further communication. Any coordination between actors requires a distributed transaction, which we call *coordinated transaction*. We combine coordinated transactions and

TM to concurrently process messages instead of blocking the actors while waiting for other transactions to commit.

We have implemented our approach in the Scala programming language and the integrated Akka framework [49]. Since this implementation cover changes to the Akka framework only, the developer is not affected at all. We evaluate our approach using a distributed linked list benchmark already used with other concurrent message processing solutions [65]. We show that concurrent message processing and non-blocking coordinated processing can considerably reduce the execution time for both read-dominated and write-dominated workloads.

The rest of the chapter is organized as follows. We first discuss the limits of the sequential message processing in Section 4.2 and then propose improvements to sequential message processing in Section 4.3. We describe our implementation in Section 4.4 and present evaluation results in Section 4.5. We finally present the summary in Section 4.6.

4.2 Problem Statement

Despite its inherent concurrency, the actor model requires sequential message processing. While this requirement is a deliberate choice to avoid synchronization hazards, it unnecessarily limits the performance (i.e., throughput) of individual actors, in particular when they execute on multi- or many-core computers.

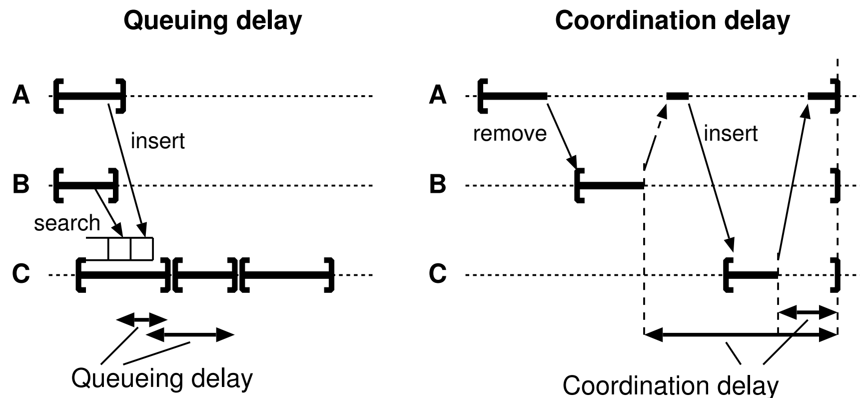

Figure 4.1: Sequential processing and its effect on execution time.

We elaborate the problem of sequential processing with the help of the examples depicted in Figure 4.1. They involve three actors (A, B and C) performing operations as illustrated on their respective time lines (horizontal dashed line). The transfer of a message between actors is indicated by an arrow and its processing is indicated by thick solid lines, where we explicitly mark the beginning and end of processing with brackets. In our examples, the actors are responsible for maintaining a distributed linked list of ordered integers. Actor B stores the first part of the list and actor C the second part, while actor A acts as a client and performs operations on the list (e.g., `search`, `insert`, `remove`).

Sequential processing in the actor model limits performance in two ways:

Queuing delay: In Figure 4.1, a common communication we depict the delay that is introduced upon arrival of multiple messages on a single actor. If actor A and B send messages to actor C, which is busy, both messages are stored in a queue. The queuing delay is the time a message has to wait at a given actor from arrival to the start of its processing.

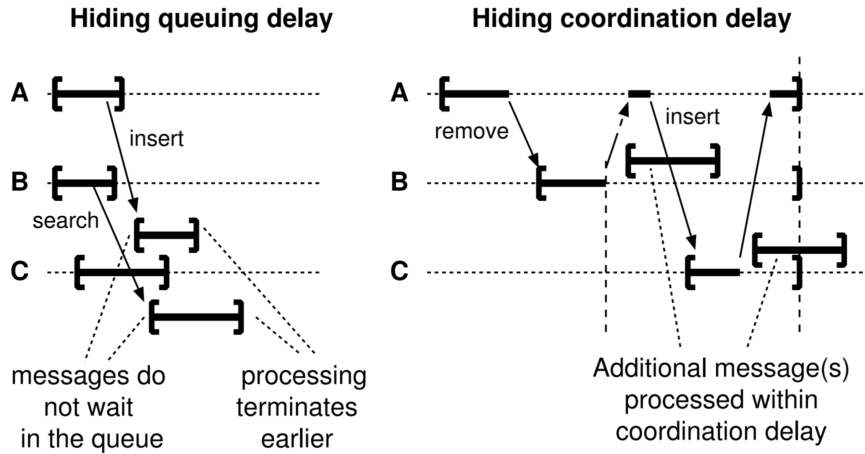

Figure 4.2: Concurrent and non-blocking coordinated processing and their effect on execution time.

Coordination delay: In Figure 4.2, we depict a common communication pattern. Consider that actor A wants to move the value x from the list of actor B to the list of actor C. For doing so, it sends messages `remove(x)` to actor B and `insert(x)` to actor C. To fulfill the macro-step semantics in actor A, the list operations have to be part of a coordinated transaction, which commits when all three actors successfully finish their task. The coordinated commit protocol defines a barrier on which actors B and C block until they can resume processing other messages. Hence, the coordination delay describes the time actors have to wait after finishing their own tasks until the distributed transaction commits.

4.3 Message Processing Model

Queuing delays are inherent to the structure of the actor model and its sequential processing operation; their reduction may become particularly important for actors that receive many messages. Further, upon coordination delay, actors block and thus cannot perform any useful work. We claim that these delays are unnecessarily long and can be significantly reduced by thoughtful changes to the message processing of actors.

Our main idea is based on the observation that we can guarantee atomicity and isolation if we encapsulate the handling of messages inside transactions. Thanks to the rollback and restart capability of transactions, several messages can be processed concurrently, even if they access the same state. We call this approach *concurrent message processing.* Additionally, we exploit the characteristics of transactions to avoid blocking actors while waiting for a coordinated commit (*non-blocking coordinated processing*).

Our concurrent message processing approach is analogous to the parallelization of instruction executions in VLIW (very long instruction word) processors. While the parallelization in VLIW processors provides instruction-level parallelism and increases application performance, our approach aims at providing a parallelization at the level of messages with the same purposes. For concurrent executions VLIW processors propose multiple execution blocks while with our approach we propose to use different threads (each of which will run in a separate processor core) to process multiple messages at the same time. A notable difference between these approaches is, however, that we perform message processing transactionally, so we do not need to know the dependencies of concurrently executed messages in advance

(i.e., the dependency issues can be resolved at runtime) whereas VLIW processors need to resolve these dependencies at compile-time to generate multi-instruction words.

With sequential message processing, the order in which messages are fetched from the mailbox determines the order of their processing. However, with concurrent execution, the processing time of messages can vary and this can result in a different processing order than in the sequential case.

In general, this is not a problem since the actor model does not impose any specific execution order. However, many applications assume FIFO order, messages originating from the same sender are processed in the same order as initiated by the sender. There can also be some other constraints in the order of messages due to the coordination of different actors as elaborated by Agha at al. [5]. Such ordering requirements have to be taken into account while determining an execution order for concurrent message processing.

To explain the principle of our two optimizations, consider the same example as in Figure 4.1 and Figure 4.2 with actor A performing operations on a list stored on actors B and C. Figure 4.2 illustrates how the delays caused by sequential processing can be reduced.
Queuing delay: By processing several messages concurrently on a single actor, we can reduce the queuing delay as shown in Figure 4.1. Therefore, if A and B send a message to C, which is currently busy, the messages do not have to wait. If the transactions do not conflict and can immediately commit, the queuing delay is avoided.
Coordination delay: Actor A wants to move a value from the list of actor B to the list of actor C, which requires both an insertion and a removal action. This is typically achieved by using a coordinated transaction. To ensure consistency, however, participating actors cannot process new messages until the coordinated transaction commits. By treating messages speculatively, one can avoid blocking the actors and allow concurrent execution of non-conflicting transactions (e.g., as in actors B and C in Figure 4.2), therefore hiding the coordination delay.

4.4 Implementation

In general, the implications of our approaches at the programming level are minimal. The user program can be written as before. Message processing is then wrapped into transactions implicitly. The effort required to do that depends on the compiler support given for TM. There exists compilers or compile-time frameworks in different programming languages (e.g., ScalaSTM API [21, 22] for Scala, Deuce [72] for Java, gcc-TM [96] and DTMC [27] for C/C++) allowing the developer to enclose the message processing code within a transactional block (generally called an atomic block) in order to make the message processing code transactional. Otherwise the programmer needs to take other necessary steps to make the code transactional (such as mapping objects used within the message processing body to objects controlled by the TM, or to map accesses to objects to corresponding method calls specific to TM).

To hide the delays as explained before, we extend Scala version 2.10.2. Specifically, we concentrated on two parts of Scala: the Akka framework version 2.10.0 and the Scala-STM library version 0.7. Akka provides a clean and efficient implementation of the actor model for the JVM. Scala-STM supports transactional memory in Scala and, while it adds some overhead for checkpointing and concurrency control, it is particularly non-invasive and well integrated in the language. In the following we describe the specific changes we made to Akka and Scala-STM to realize the proposed optimizations.

4.4.1 Concurrent Processing of Messages

The concurrent message processing only involves changes of the message handling provided by the Akka framework. Specifically, we changed the behavior of the actor's mailbox. In the original Akka implementation a dispatcher is responsible for ensuring that the same mailbox is not scheduled for processing messages more than once at a given time. Another particularity of Akka is that every actor has one mailbox with two queues: the first one stores user messages, i.e., messages received from other actors, while the second one is used for maintaining system messages specific to Akka, which control lifecycle operations (i.e., start, stop, resume). Once a user message is scheduled, the dispatcher checks first if there are any system messages. Then, all existing system messages are treated before the user message. The same is done after the processing of the user message. As system messages are rare, actors spend most of their time processing user messages.

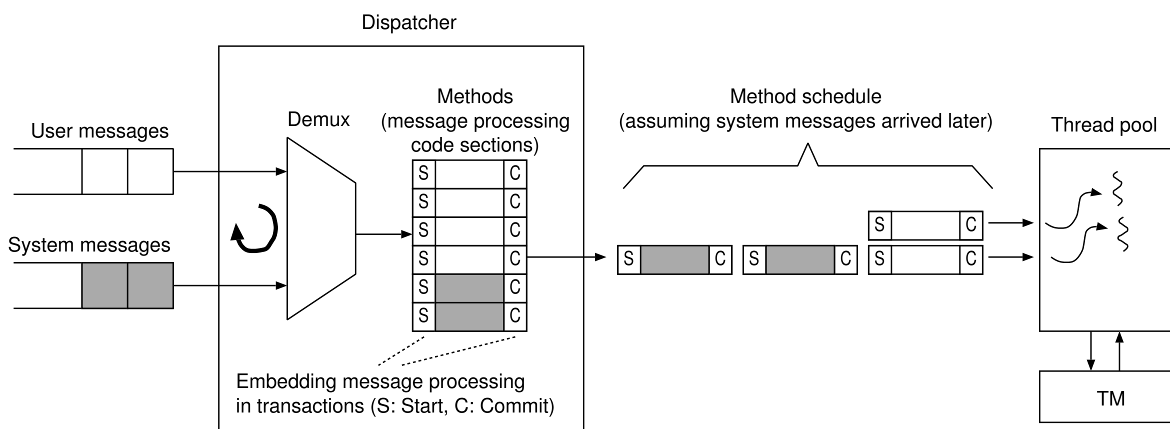

Figure 4.3: Implementation of concurrent message processing.

To facilitate concurrent message processing, we reimplemented the mailbox and message treatment as shown in Figure 4.3. System messages are handled as in the sequential case, before and after user messages, but instead of processing user messages one at a time, we process them concurrently in batches. Each user message from a batch is submitted to a thread-pool for execution. The actual message processing is performed concurrently inside a transaction, as indicated by the start (S) and commit (C) events in the figure. For the transactional handling of messages we use the default Scala-STM. If the concurrent operations do not conflict, we can hide the queuing delay as illustrated in Figure 4.2.

4.4.2 Non-blocking Coordinated Transactions

The non-blocking coordinated transaction alters the commit behavior of the Scala-STM. By default, a coordinated transaction is blocking (see Figure 4.1). All actors participating to a coordinated transaction must reach a commit barrier before any other message can be processed.

Consider the case of a transaction that executes a block of code corresponding to the processing of a message. After the transactional code is executed, the STM makes an attempt to commit the changes, possibly rolling back and trying again upon failure. In the process of a commit, several steps are performed: (1) locks for the variables accessed in the transaction are obtained; and (2) if the transaction belongs to a coordinated transaction, an external *decider* is consulted. The coordinated transaction's commit barrier blocks as long as some of

its transactions are still executing. Once they have all successfully completed, the commit barrier is unlocked and control is returned to the caller. After the external decider returns, the transaction does final sanity checks and flushes outstanding writes to main memory.

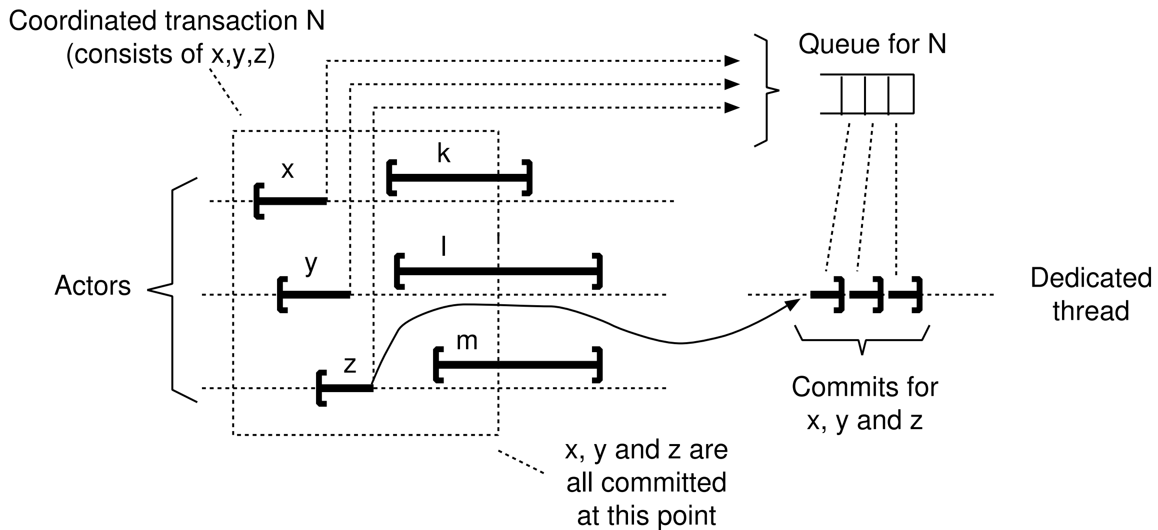

Figure 4.4: Sketch of the implementation of non-blocking coordinated processing.

In our implementation, illustrated in Figure 4.4, we perform the following operations instead of blocking the thread when waiting for other parties to arrive at the barrier. We first save the current transaction descriptor in a queue (*queue for N*). Then, we return from the atomic block immediately, bypassing any additional logic associated with the commit operation. By doing so, we do not fully commit the transaction; we instead suspend its commit at the point where it would normally block.

While an actor is unblocked, it can handle other messages concurrently, hiding the coordination delay as illustrated on transactions *k*, *l*, and *m* in Figure 4.4. If concurrent messages (i.e., the suspended/pre-committed coordinated transaction and the speculatively processed message) are independent, they can commit immediately. If there is a read-write or a write-write conflict, i.e., the speculatively processed message conflicts with the suspended/pre-committed coordinated transaction, we delay the commit of the transaction executing the speculatively processed message until the coordinated transaction completes.

In a system comprising multiple actors, it is likely that several coordinated transactions execute concurrently. Each coordinated transaction uses its own queue to store its suspended pre-committed transactions (`N` corresponds to the identifier of the coordinated transaction in the figure). Hence, we do not mix pre-committed transactions belonging to different coordinated transactions. To resume the commit, a dedicated thread is notified when all the parties belonging to the same coordinated transaction have completed their work. That thread then resumes the commit by firing the appropriate ScalaSTM callbacks.

In summary, after we pre-commit the coordinated transaction, it becomes irrevocable with respect to the transactions executing speculative messages, meaning that it still executes as a transaction but is not allowed to rollback, i.e., if another speculatively processed transaction conflicts with the pre-committed transaction (due to access to same data) the speculatively executed transaction would have to wait until the pre-commited coordinated transaction has been committed. The pre-committed transaction can be rolled back only in the case when one of the parties participating in the coordinated transaction rolls back. In this case all parties participating in a coordinated transaction need to also abort, rolling

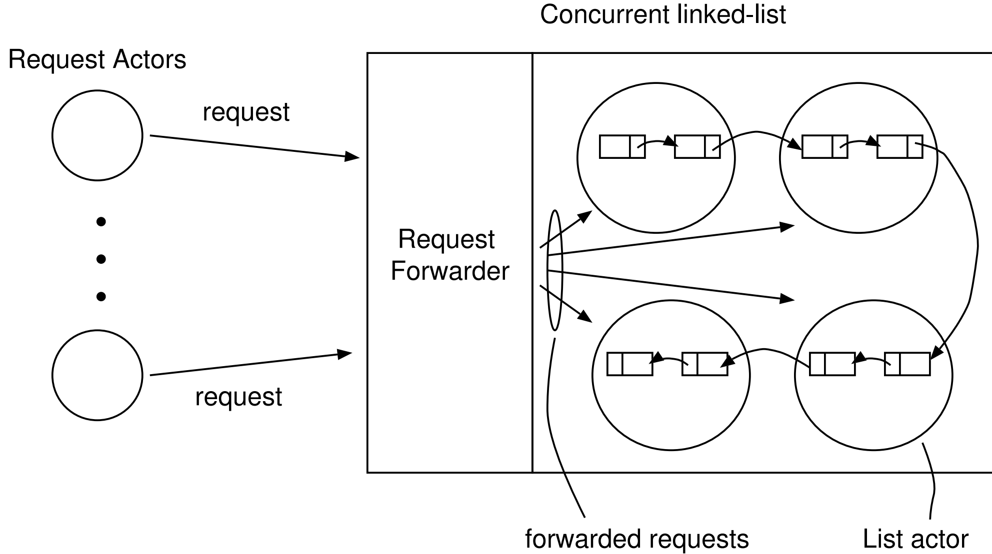

Figure 4.5: Concurrent sorted integer linked-list benchmark using actors.

back their transactions. By speculatively processing messages instead of blocking we allow for more concurrency since we permit non-conflicting transactions to run in parallel. Out approach also ensures that the speculatively executed transaction cannot induce the rollback of a pre-committed coordinated transaction.

4.5 Evaluation

4.5.1 Read-dominated Workload

Our optimizations are expected to be most useful in applications where state is shared among distributed actors. Hence, to evaluate our approach, we use a benchmark application provided by Imam and Sarkar [65] that implements a stateful distributed sorted integer linked-list. The architecture considers two types of actors: *request* and *list* actors. Request actors only send requests such as `lookup`, `insert`, `remove`, and `sum`. List actors are responsible for handling a range of values (buckets) of a distributed linked list. In a list with l actors, where each actor can store at most n elements representing consecutive integer values, the ith list actor is responsible for elements in the $[(i-1) \cdot n, (i \cdot n) - 1]$ range, e.g., in a list with 4 actors and 8 entries (Figure 4.5, each actor is responsible for two values.

A request forwarder matches the responsible list actors with the incoming requests. We extend this benchmark to evaluate different facets of our proposed optimizations. We evaluate different workloads, different numbers of actors holding elements of the list, etc. For the `sum` operation, each actor holds a variable that represents the current sum of all its list elements, called `partial_sum`, which is updated upon insertion and removal. When computing the sum of the whole list, we only accumulate the partial sum of each list actor without the need of traversing all list elements. While the lookup, insert, and remove operations execute on a single list actor, the sum operation needs to traverse all the list actors in order to return the partial sums. Hence, the sum operation involves multiple list actors. The original benchmark did not initially ensure atomicity of the `sum` operation; we therefore changed the implementation so that the computation of the sum is performed within a coordinated transaction.

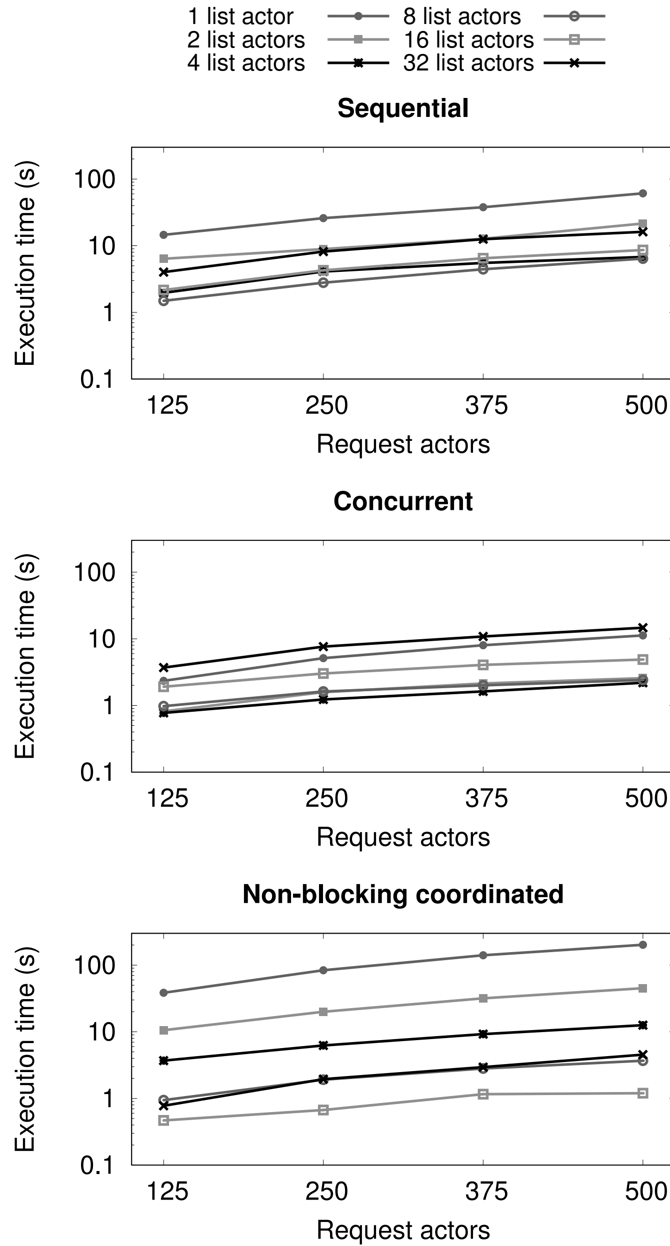

Figure 4.6: Execution time for sequential, concurrent, and non-blocking message processing on a read-dominated workload.

In the sequential case, the sum operation proceeds as depicted in Figure 4.7. In our example there are two request actors (R1 and R2) and two list actors (L1 and L2). Actor R2 sends a `sum` message to the list actor L1 and forwards its partial sum to L2. Actor L2 adds its own sum and responds R2 with the result. The coordinated transaction comprises the operations done on L1 and L2, thus L1 can commit when L2 finishes its work.

In the sequential case, if R1 sends an `insert(x)` request to L1 concurrently to the execution of the sum operation, the processing of this message can only start after the coordinated sum transaction is done (left side of Figure 4.7 and Figure 4.8). The same applies if R1 and R2 want to perform a sum operation concurrently: although both operations are read-only, the requests can only be processed sequentially (right side of Figure 4.7 and Figure 4.8).

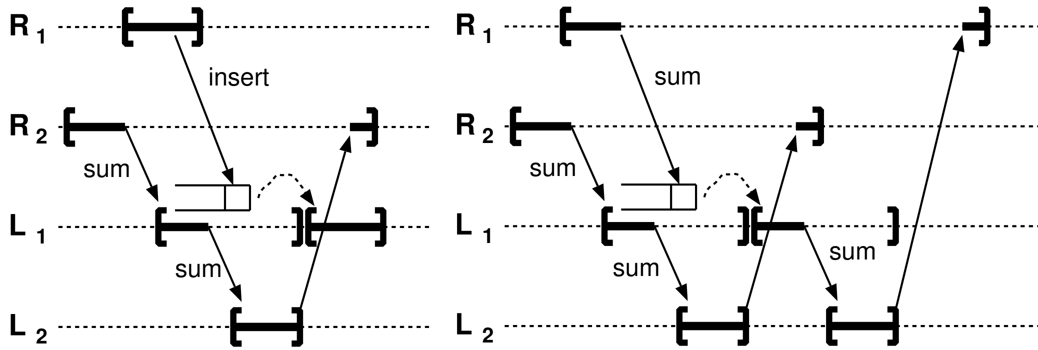

Figure 4.7: Sequential processing of the coordinated sum operation with an insert operation (left) and another sum operation (right).

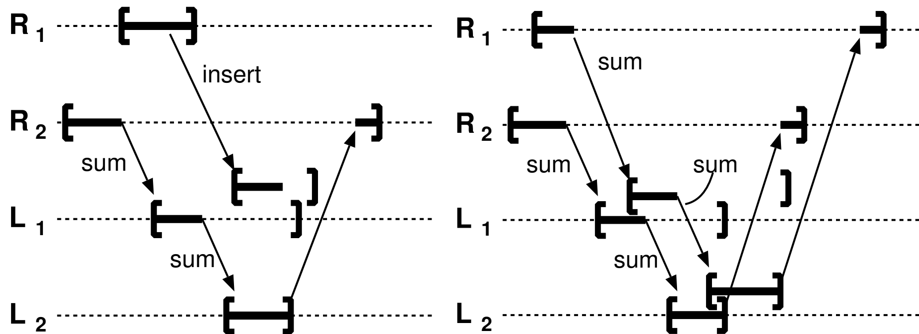

Figure 4.8: Concurrent processing of the coordinated sum operation with an insert operation (left) and another sum operation (right).

We execute the benchmark on a 48-core machine equipped with four 12-core AMD Opteron 6172 CPUs running at 2.1GHz. Each core has private L1 and L2 caches and a shared L3 cache. The sizes of both instruction and data caches are 64KB at L1, 512KB at L2, and 5MB at L3.

We apply each of the extensions—concurrent message processing and non-blocking coordinated processing—to a read-dominated workload and then to a write-dominated workload. Each sample corresponds to the geometric mean of 7 runs.

We first evaluate both extensions separately to better assess their benefits and drawbacks, i.e., for the first results, non-blocking coordinated processing does not include concurrent message processing. Then, we conduct experiments with both approaches combined. Their performance is compared against sequential message processing, i.e., using default Akka/Scala constructs without transactions for read and write operations. The sum operation is put into a coordinated transaction as provided by Akka/Scala. We finally complete our evaluation with a comparison against the Habanero-Scala implementation.

Our experiments are either read-dominated (lookup) or write-dominated (insert, remove). These workloads additionally contain a number of sum operations, which are treated as coordinated messages. More precisely, each actor performs $R = x\%$ reads, $W = y\%$ writes, and $S = z\%$ sum operations, where $R + W + S = 100\%$. Since sum requests are likely to be rare in comparison with other operations, we keep this parameter constant at $S = 1\%$. For read-dominated workloads, we choose $R = 97\%$ and $W = 2\%$. The write-dominated workload is configured with $R = 1\%$ and $W = 98\%$.

Insert and remove operations are handled independently and are chosen at random, but each evaluation run gets the same input. We vary the number of list and request actors, each of the latter sending 1,000 requests. The request actors wait for a response before sending the next message. The list can contain a maximum of 41,216 integers split evenly between actors. For instance, if there are 32 list actors, each will be responsible for 1,288 buckets. The list is pre-filled to 20 % of its capacity.

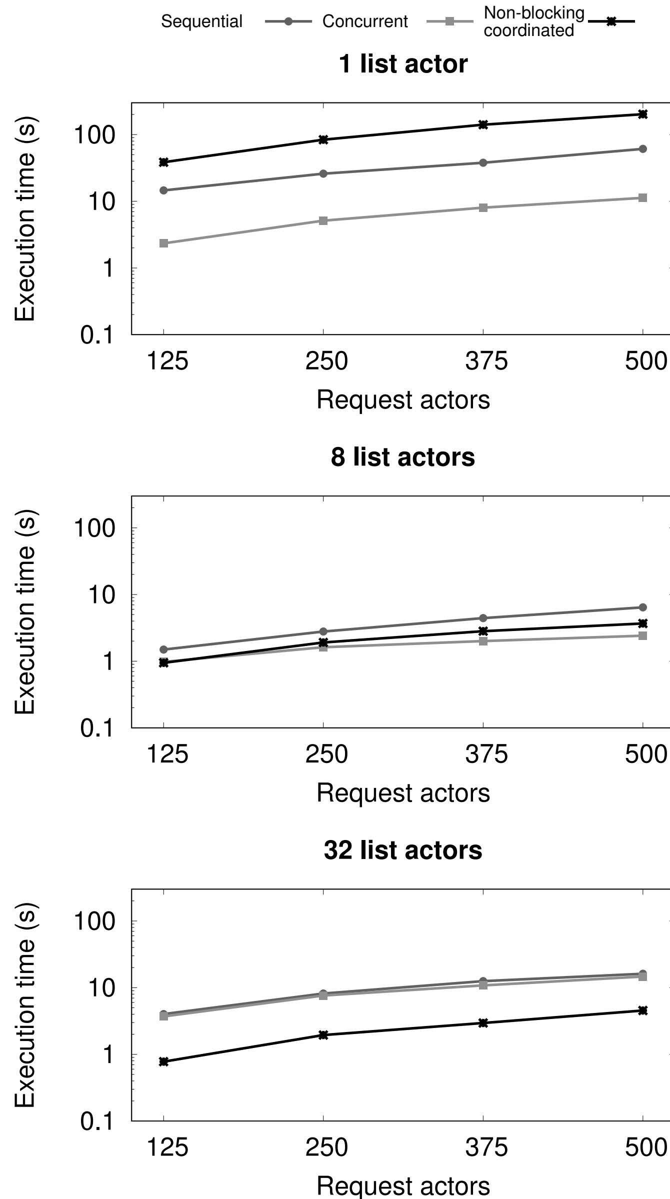

Figure 4.9: Execution time for sequential, concurrent, and non-blocking message processing using 1, 8, and 32 list actors on a read-dominated workload.

Figure 4.6 presents the results for the three separate scenarios: sequential message processing, concurrent message processing, and non-blocking coordinated processing. On the x-axis we show the effects of increasing the number of request actors (125–500), while the y-axis displays the execution time in seconds (log scale), i.e., the time needed to finish processing all requests. The lower the execution time, the better. One can see in all cases that adding more request actors leads to higher execution times, which is not surprising because the workload becomes higher.

The concurrent execution time (Figure 4.6, center) is lower than the sequential scenario up to 16 list actors, which indicates that allowing multiple messages to be processed concurrently introduces an immediate performance gain. In our linked-list example, messages sent from request actors R1 and R2 to a list actor L1 will be treated within a transaction. If there are no conflicts—which is likely in a read-dominated workload—concurrent transactions commit successfully. Therefore, the time required to process a batch of messages will be equal to the execution time of the longest associated transaction ($\max(T_1, T_2, \ldots)$) instead of the sum of all execution times ($\sum T_i$). The performance with 16 and 32 actors starts to degrade because the workload provides less exploitable concurrency. With 32 list actors the performance is even worse than with a single list actor.

When considering non-blocking coordinated message processing (Figure 4.6, right), the explanation for the increase of the execution time for concurrent processing with 16 and 32 list actors is clear: when the number of list and request actors is high, coordinated transactions are likely to fail because of increased contention. Non-blocking coordinated message processing allows us to reduce this execution time considerably. Actors can process other messages while the sum operation is in progress. The reduction of execution time is especially high for read-dominated workloads, because a lookup operation and the read of the partial sum are non-conflicting operations. With large numbers of list and request actors, however, the likelihood of insert and remove operations increases significantly.

Figure 4.9 shows a more detailed comparison for an increasing number of list actors. The left graph presents the execution time for a single list actor. One can see that concurrent message processing improves the execution time considerably, while non-blocking coordinated processing exhibits worse performance than sequential message processing. Indeed, since there is only one list actor, no coordinated transactions are executed, i.e., the sum operation only returns the partial sum of the current list actor. Hence, the execution time of the non-blocking coordinated processing shows the overhead of executing all operations inside transactions. When increasing the number of list actors, this overhead is compensated by the benefits of non-blocking coordinated processing. When increasing the number of list actors to at least 8 (Figure 4.9, center), the contention of coordinated transactions increases and non-blocking processing performs even better than concurrent message processing. When the number of coordinated transactions and write-write conflicts becomes too high, concurrent message processing yields performance similar to sequential processing, as can be observed in the right graph of Figure 4.9 for 32 list actors. In contrast, non-blocking coordinated transactions lead to significantly lower execution times than both sequential and concurrent message processing.

To summarize our findings so far, concurrent message processing has the highest impact if the number of list actors is low because each will have more messages to process, i.e., the penalty from serialization is more important and the workload provides more exploitable concurrency. The opposite trend can be observed with non-blocking coordinated transactions: they benefit most when the number of list actors is high because coordinated transactions become longer, i.e., the penalty of the blocking operation is higher and contention is relatively low. Therefore, the combination of both techniques is expected to provide good overall performance for all considered scenarios.

4.5.2 Write-dominated Workload

We expect to observe more conflicts with a write-dominated workload because each insert and remove operation also modifies the value of the partial sum. As a consequence the execution

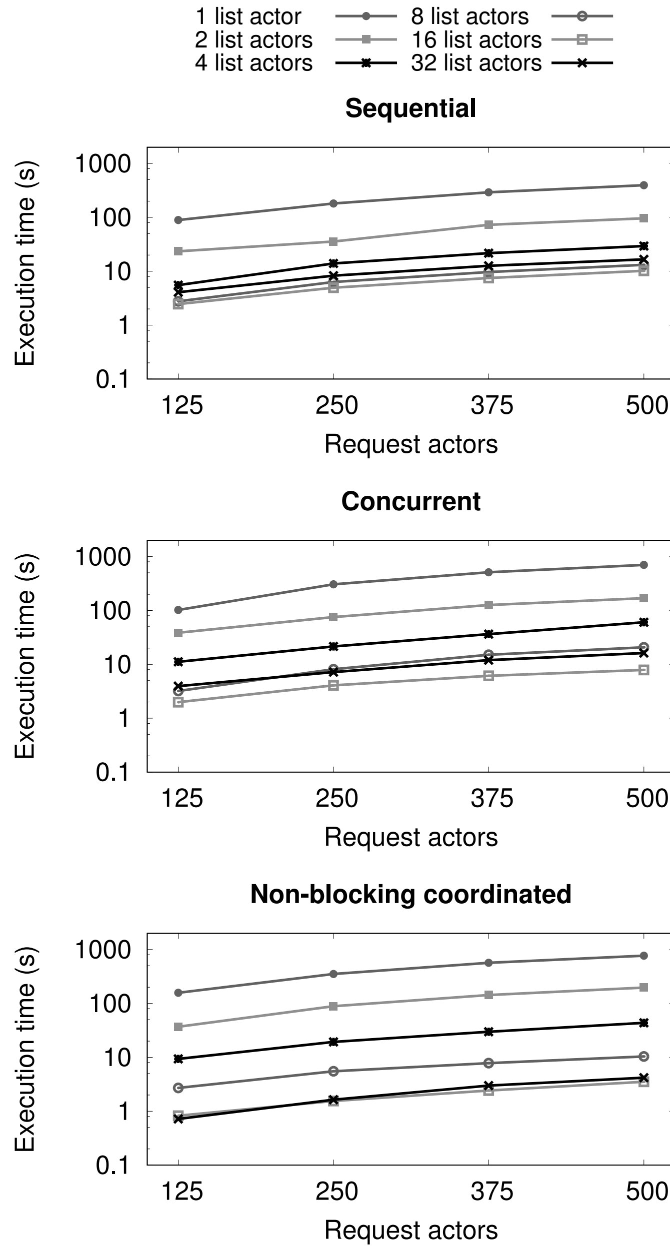

Figure 4.10: Execution time for sequential, concurrent, and non-blocking message processing on a write-dominated workload.

time generally increases in comparison with the read-dominated load, as shown by the graphs in Figure 4.10.

Sequential processing (Figure 4.10, left) performs similarly to the read-dominated workload case. The execution time first improves when adding more list actors. Then, we observe similar execution times for 8 and 16 list actors, and the degradation starts for 32 actors as the impact of coordinated transactions becomes more significant. Concurrent processing (Figure 4.10, center) provides better overall performance, but the best improvement is obtained with 16 list actors when there is sufficient exploitable concurrency. Finally, with non-blocking coordinated processing, performance improves with the number of list actors. The execution time is better than concurrent processing starting from 4 list actors. The reason is that sum operations are read operations. Thus, for coordinated transactions a write-

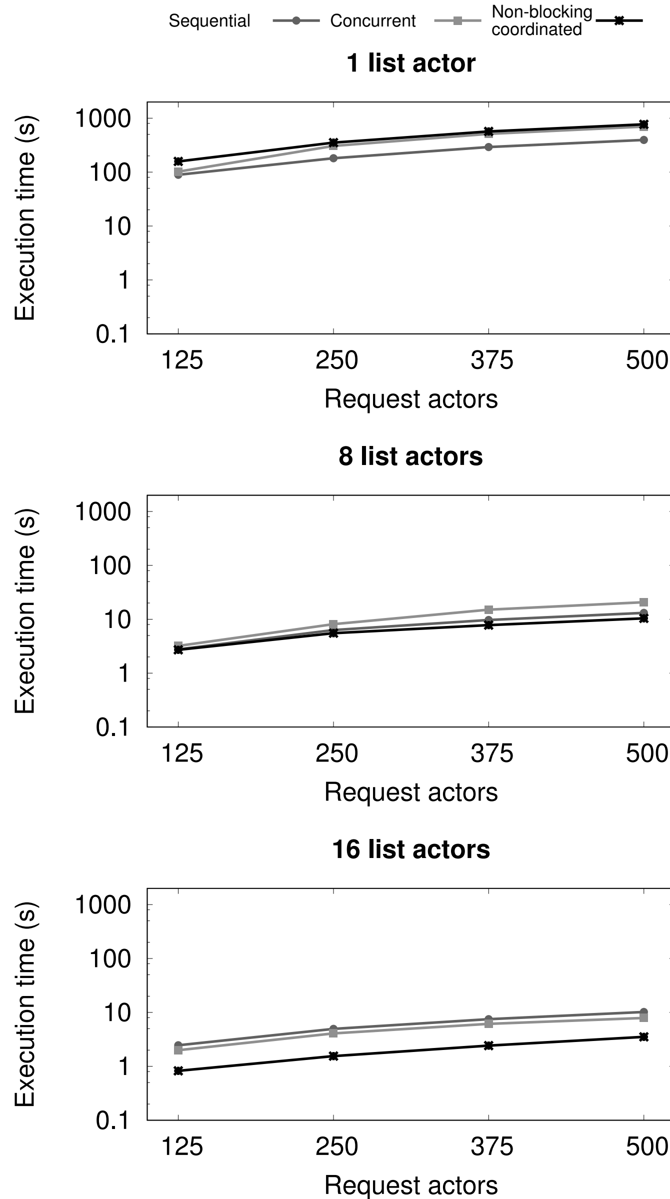

Figure 4.11: Execution time for sequential, concurrent, and non-blocking message processing using 1, 8, and 16 list actors on a write-dominated workload.

write conflict is not possible. We expect that write operations in coordinated transactions conflicting with other writes lead to execution times close to concurrent processing.

Figure 4.11 shows the execution times of sequential, concurrent, and non-blocking coordinated processing for various sizes of list actors. With a single list actor (Figure 4.11, left), we observe that both concurrent and non-blocking coordinated processing perform poorly due to the many write-write conflicts and resulting aborts.

With 8 list actors (Figure 4.11, center) the performance of non-blocking coordinated processing becomes close to sequential processing, whereas concurrent processing still has a higher execution time. Finally, with 16 list actors the advantage of non-blocking coordinated processing is obvious, while concurrent processing now performs similarly to sequential processing because the coordination delays dominate.

Summing up the write-dominated workload results, we conclude that concurrent processing becomes less beneficial when the likelihood of write conflicts is high. In some

cases, the high number of roll backs becomes high enough that sequential processing should be preferred. Coordinated transactions have a high influence on the execution time and significantly improve performance with many list actors. Therefore, a write-dominated workload can benefit more from non-blocking coordinated transactions than concurrent ones, and it is debatable whether the latter extension should be used at all when the number of write-write conflicts becomes very high.

4.5.3 Non-blocking Concurrent Processing

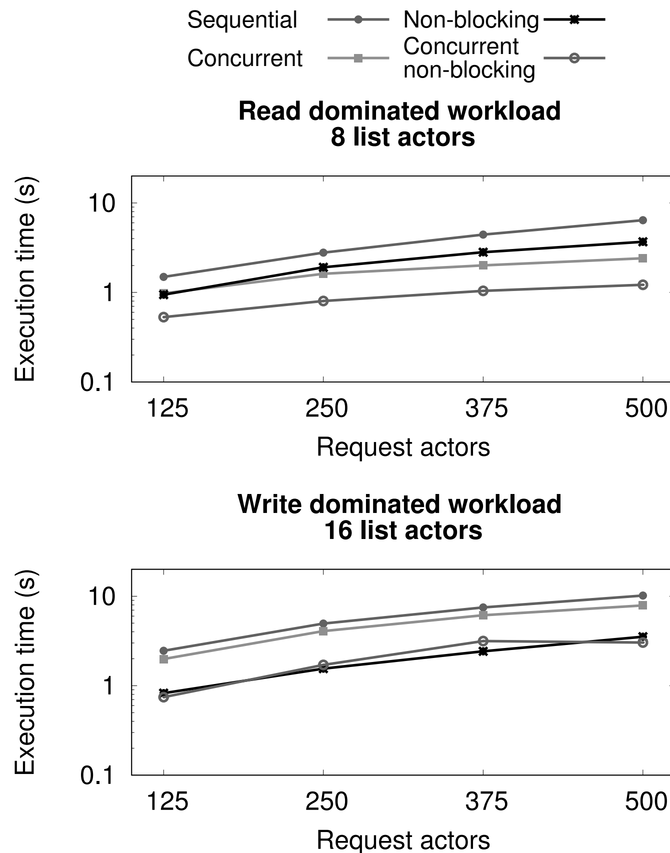

Figure 4.12: Execution time for sequential, concurrent, non-blocking, and combined message processing.

We conducted the same experiment as before, but combined concurrent and non-blocking coordinated message processing, which we call *non-blocking concurrent processing*, for the read and write-dominated workload. In the read workload, concurrent processing leads to lower execution times when the number of list actors is below 16. Indeed, one can observe that the performance of non-blocking concurrent processing is even better than non-blocking coordinated processing (Figure 4.12, left). When we increase the number of list actors, we see again the same behavior as for the write-dominated workload. The combination is thus useful when both concurrent and non-blocking coordinated processing lead to a lower execution time than sequential processing.

The results for the write-dominated workload show that concurrent processing does not have much influence on the execution time (Figure 4.12, right). In the 16 buckets scenario, pure concurrent processing has execution times similar to sequential processing, while non-

blocking concurrent processing results in performance close to non-blocking coordinated processing.

To fully exploit the capabilities of the proposed mechanisms, it is therefore necessary to properly understand the nature of the workload. If it is read-dominated and the number of list actors is high, one should favor non-blocking coordinated processing. If the number of list actors is low, one should rather use non-blocking concurrent processing. Finally, with a write-dominated load, one should prefer non-blocking coordinated processing or even switch back to sequential processing.

4.5.4 Comparison to Habanero-Scala

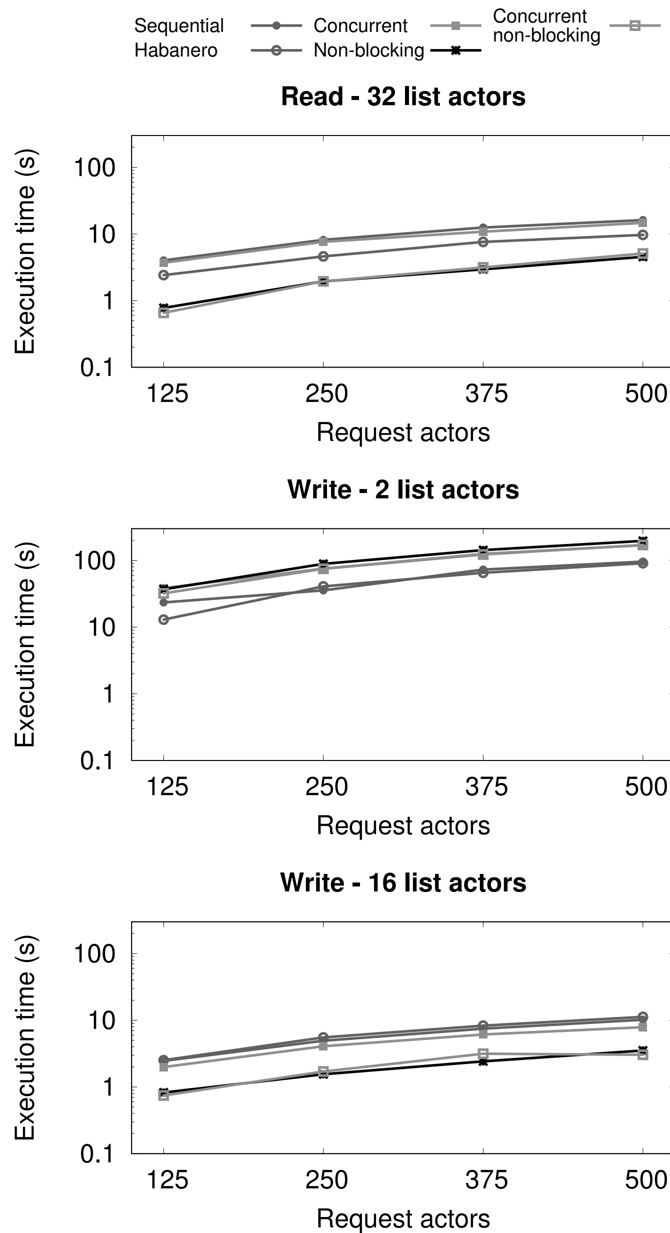

Figure 4.13: Execution time comparison with Habanero-Scala.

In the original Habanero-Scala benchmark shown in Figure 20 in [65], the authors ran their experiments with 64 list actors, each responsible for 400 buckets. Additionally,

the authors used a workload with a balanced mix of reads and writes (50:50). 50 read actors are used to access the list elements, with 32,000 accesses by actor. For these settings, Habanero-Scala (*LightActor* implementation) performed slightly better than default sequential processing (Akka). Note that the LightActor implementation of the list benchmark does not spawn any sub-tasks. It uses a *finish* construct instead of the default countdown latch for coordinating list and request actors of the list. Thus, the difference in performance is due to their lightweight implementation of actors, a custom task scheduler, and a different thread-pool implementation.

To show the full capabilities of our approach, we executed our experiments with Habanero-Scala. For this, we use the default Habanero-Scala constructs for read and write operations (no transactions). The behavior of the sum operation is provided by a barrier implemented in the list actors using the *DataDriveFuture* construct (including sub-tasks) of Habanero-Scala.

For the read-dominated workload, Habanero-Scala performs similarly to sequential message processing, except for 32 buckets where Habanero-Scala performs better by approximately 40%. There, the difference is higher, because of the penalty of the blocking coordinated sums used in the sequential implementation. As seen in the left graph of Figure 4.13, our approach outperforms Habanero-Scala by 50 to 70%.

In the write-dominated workload, Habanero-Scala performs again similarly to sequential message processing. With our speculative extensions, the contention is too high for less than 8 list actors. With 16 list actors the improvement of our approach is significant (Figure 4.13, right).

4.5.5 Discussion

Our approach, which combines concurrent and non-blocking coordinated processing, guarantees the same correctness as the original actor model. We perform the processing of messages within transactions, which means that concurrent operations on the actor's state will execute atomically and in isolation. Therefore, conflicting operations will be serialized, but non-conflicting messages should be processed concurrently.

It is important to note that the actor model does not impose any order on the processing of messages that are in their incoming queues. Therefore the non-deterministic order in which transactions will commit does not break the semantics of the actor model. If ordering were required, we could extend the STM as proposed in [19] to enable parallel processing but commit transactions in order.

As we rely on transactional memory to process messages equivalently to a serial execution and we preserve the original actor model, concurrent processing also provides the same correctness guarantees. The same applies for non-blocking coordinated processing. All messages are handled within transactions, which means that the conflicts are handled as for concurrent message processing. However, for read-write conflicts (e.g., concurrent sum and insert operations) the order will be preserved by our delayed commit mechanism. An issue that currently limits our approach is that the code of "transactified" message processing should not contain any action that is not under the control of the TM, such as I/O or OS library code (irrevocable code). Strategies for supporting irrevocable actions are left to the responsibility of the underlying TM and are not specific to our extensions.

Our approach has the important benefit of being adaptable. The transactional processing of a message can be aborted at any time without side effects on the current state of the actors. This implies that each of our extensions can be enabled or disabled at any

time during execution, and one can switch from one extension to the other within the same execution. Such flexibility can be exploited to play with trade-offs between performance and resource utilization. On the performance side, messages need to be processed anyways, either sequentially or concurrently. Hence, if we have idle resources and we can process messages concurrently, the overall task can be completed in a shorter time. On the resource utilization side, the adaptability of our approach allows us, for example, to apply simple energy efficiency strategies that enable or disable transactional execution, possibly even temporarily switching off some cores, in order to fit the consumption of the application to the desired energy requirements.

4.6 Summary

The actor model implements synchronization by means of message passing. This decoupled communication paradigm is particularly scalable since it allows multiple actors to perform independent computations concurrently as they do not share state. However, each actor processes arriving messages sequentially.

To address this limitation, we proposed an approach that combined transactional memory (TM) and actors as implemented by the Akka Framework. Incoming messages are dequeued in batches and processed speculatively inside transactions. The atomicity, consistency, and isolation properties of TM guarantee that messages do not interfere when being processed concurrently. In addition, as the actor model does not impose any order on the handling of user messages in the incoming queues, our approach preserves its semantics.

To further improve concurrency, we also extended the coordinated transaction mechanism of Scala-STM to support non-blocking operations. The traditional design prevents actors involved in a coordinated transaction to process any additional message until the transaction commits. The resulting delays can be especially high when actors are distributed on several nodes and communication has non-negligible latency. We solved this issue by speculative concurrent message processing.

Together, these two mechanisms significantly lowered the queuing and coordination delays, and hence increased concurrency. We implemented both mechanisms in the Scala language using the integrated Akka framework. Experiments on a 48-core server showed that our extensions provided important performance benefits over sequential processing on both read-dominated and write-dominated workloads.

Future work regards dynamic switching between sequential and concurrent processing and the investigation of a real-world application.

Chapter 5

Dynamic Scheduling for Message Processing

5.1 Introduction

Recent scaling trends lead to ever growing data centers and cloud computing is gaining attention. Further, the scaling trends at the CPU level let us expect increasing core counts in the following years. This causes limitations of performance gains as it is difficult to program distributed and concurrent applications efficiently. The current methods using shared memory do not keep up with the hardware scaling trends and might need to be abandoned.

The actor model, initially proposed by Hewitt [58], is a successful message passing approach that has been integrated into popular local and distributed frameworks [68]. An actor is an independent, asynchronous object with an encapsulated state that can only be modified locally based on the exchange of messages. Received messages are processed sequentially avoiding the necessity of locks. With increasing numbers of actors the model is inherently parallel. To sum up, the actor model introduces desirable properties such as encapsulation, fair scheduling, location transparency, and data consistency to the programmer.

While the data consistency property of the actor model is important for preserving application safety, it is arguably too conservative in concurrent settings as it enforces sequential processing of messages, which limits throughput and hence scalability. With sequential processing, access to the state will be suboptimal when operations do not conflict, e.g., modifications to disjoint parts of the state and multiple read operations.

In Chapter 4, we improved the message processing performance of the actor model while being faithful to its semantics. Our key idea was to use transactional memory (TM) to process messages in an actor concurrently as if they were processed sequentially. TM provides automatic conflict resolution by aborting and restarting transactions when required. However, we noticed that in cases of high contention, the performance of parallel processing dropped close to or even below the performance of sequential processing. To improve the performance of such high contention workloads, we propose to parallelize the processing of messages in a transactional context with messages that can run in a non-transactional context.

First, we determine the optimal number of threads that execute transactional operations as the performance of an application is dependent on the level of concurrency [33]. To avoid high rollback counts in high contention phases fewer threads are desired, in contrast to low contention phases where the number of threads can be increased. Second, we extract messages for which we can relax the atomicity and isolation and process them as non-transactional messages. Didona et al. [33] argue that the performance of an application is dependent on the level of concurrency, and propose to dynamically determine the optimal

number of threads depending on the workload. This means that in high contention phases fewer threads are required, whereas the level of concurrency can be increased in low contention phases. Since we consider workloads with high contention, the majority of threads will be unassigned.

Much of the contention in our tests was caused by read-only operations on TM objects [54]. Rollbacks could be avoided by relaxing the semantics of read operations. Some classes of applications can accept inconsistent values for heuristic decisions such as proposed by Herlihy et al. [56] who introduced *early release*, which allows to delete entries from the read set. Another way is to suspend the current transaction temporarily, which is called *Escape Action* [80].

These approaches are only partly realized in current STMs such as the Scala STM (based on CCSTM [22]). Scala STM uses *Ref* objects to manage and isolate the transactional state. The Ref object does not permit accessing its internal state in a non-transactional context. Instead, Scala STM provides an *unrecorded read* facility, in which the transactional read does not create an entry in the read set but bundles all meta-data in an object. At commit time the automatic validity check is omitted, but may be done by the caller manually. Without the validity check, the result of the unrecorded read might not be fully consistent but can often safely be used for tasks such as heuristic decisions or approximate computations. However, in the presence of concurrent writes on the same value, the unrecorded read might cause conflicts and hence performance would degrade. Our proposal is to break the isolation guarantees of the Ref object in specific cases, i.e., if we limit the reads to specific isolated values, we can omit using the unrecorded read and grant direct access. By using direct access for a number of scenarios in the context of the actor model, we can process a substantial amount of read-only messages while not conflicting with messages processed in regular transactions (TM messages).

Possible candidates for such read-only operations can accept inconsistencies while not interrupting with transactional states. Examples are operations that can be used to make heuristic decisions, or operations that are known to be safe because of algorithm-specific knowledge. Furthermore, debugging and logging the current state in long-running applications are candidates for read-only operations.

The read-only and the TM messages may require different levels of concurrency. Following this observation, during high contention phases, we reduce the number of threads processing regular TM messages, which in turn allows us to increase the number of threads processing read-only messages. By handling these two message types separately (i.e., providing a separate queue for each of the message types) we can optimally use all available resources. We show the applicability of our approach by using a micro-benchmark and a real-world application.

This chapter is organized as follows: In Section 5.2, we introduce the basics of our former work and in Section 5.3, we discuss the proposed extensions. The benchmarks are described and evaluated in Section 5.4. Finally, we present the summary.

5.2 Concurrent Message Processing

To motivate our proposed work, we relate to the enhancements of the actor model as presented in our former work [54]. There, we reduced the execution time without violating the main characteristics of the actor model. Our main idea was based on the observation that we can guarantee atomicity and isolation if we encapsulate the handling of messages inside

transactions. Thanks to the rollback and restart capability of transactions, several messages can be processed concurrently, even if they access the same state. The concurrent message processing only changes the message handling provided by the Akka framework as integrated in Scala 2.10.0. Specifically, we altered the behavior of the actor's mailbox processing code. In the original Akka implementation a dispatcher is responsible for ensuring that the same mailbox is not scheduled for processing messages more than once at a given time. We adapted the dispatcher to assign each message processing to a thread of the thread pool. Further, each message processing is handled in a transaction for which we use the default Scala STM.

Our work performed well for read-dominated as well as write-dominated workloads and we outperformed the state-of-the-art Habanero Scala [65] in their distributed list benchmark shared amongst list actors with 97% of reads, 2% writes and 1% sum operations created by request actors. Figure 5.1 shows performance improvements over sequential processing and Habanero Scala for all numbers of list actors in a shared linked list. On the x-axis we show the effects of increasing the number of request actors (125-500), while the y-axis displays the execution time in seconds (log scale), i.e., the time needed to finish processing all requests. The lower the execution time, the better. Instead of processing messages sequentially, each message from a batch is submitted to a thread-pool for execution. A higher number of list actors also leads to higher contention, increased rollback counts and hence decreased performance. With 16 list actors, concurrent processing as well as Habanero Scala perform close to sequential processing.

5.3 Dynamic Concurrent Message Processing

For improving the performance in high contention workloads we propose the following combination of methods. First, we adapt the level of concurrency for processing actor messages according to the current contention level. Second, we extract read-only message processing from the transactional context. And third, we exploit the fact that the two types of messages do not interfere and occupy the existing threads with both types of messages according to the current contention level. As a result, we occupy all threads with work. To differentiate between these two types of messages, we adapted the concurrent mailbox implementation in Akka (as part of Scala 2.10) to handle two different queues as shown in Figure 5.2. One queue collects the messages to be processed in a transactional context (STM messages), and another one for the processing of read-only messages. We further adapted the actor message dispatcher such that it picks messages from both queues and forwards them for processing to the thread pool. Our new dispatcher automatically adapts the number of STM messages and read-only messages to be processed according to the current contention of the workload. The detailed principles of processing STM messages and read-only messages are described in the following sections.

5.3.1 STM Message Processing

At the beginning of an application we start with a random number of threads from the thread pool to process transactional messages (STM messages) and then, driven by a predefined threshold α, the level of concurrency is adapted. The dispatcher assigns the rest of the threads to process read-only messages. The α value is dependent on the knowledge of the current number of commits and rollbacks of transactionally processed messages and their ratio. Instead of focusing on the throughput such as Didona et al. [33], we are able to

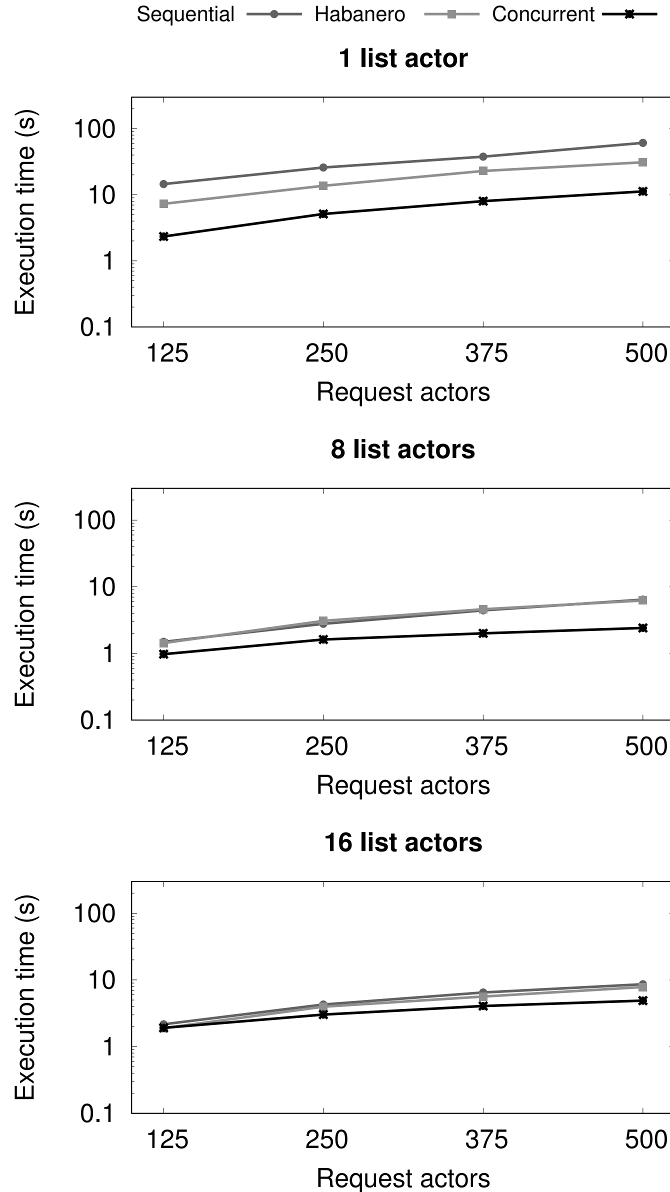

Figure 5.1: Concurrent message processing in a read-dominated workload within a shared linked list.

support transactions of any granularity by steering α with the commit-to-rollback ratio. If the current commit-to-rollback ratio is lower than α, we divide the number of threads processing transactional messages by two; if it is higher, we multiply them by two. We chose the commit-to-rollback ratio in combination with the fixed α threshold due to its simplicity. To find the right α value for the current workload, we consider a short profiling phase monitoring the relation of commits and rollbacks. Additional performance benefits could be obtained by considering a combination with hill-climbing or gradient descent.

During the application execution we use an additional thread for monitoring rollback and commit count, which operates every *80 milliseconds*. By using the information about rollbacks and commits, along with the α value, that thread is also responsible for identifying the optimal number of threads processing STM messages.

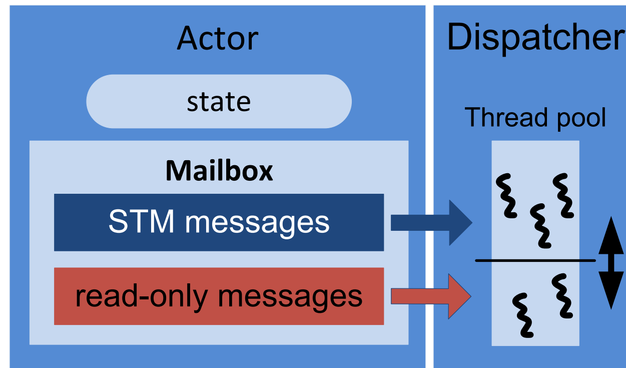

Figure 5.2: The different handling of STM messages and read-only messages. Dispatcher assigns a dynamic number of threads for message processing.

5.3.2 Read-only Message Processing

We will show that the adaptive number of threads has a huge impact on the performance. However, if the contention is so high that only a few threads are occupied, resources are wasted possibly causing decreased performance. Therefore, we consider a second category of messages, which are handled at a different granularity than messages that require TM context. The second category of messages are read-only messages that can be handled concurrently with STM messages. Scala STM [21] is based on CCSTM [22] (which extends SwissTM [34]). It is a write-back TM, where writes are cached and written to the memory on commit. Further, it provides eager conflict detection for writes and lazy conflict detection for reads. Validation is done based on a global time stamp. In Scala STM transactional objects are encapsulated in so-called *Ref*s. This implies that any access to a transactional object has to be within an atomic block, which ensures strong atomicity and isolation.

We argue that in some cases we can relax isolation, e.g., for performing approximate read-only operations. These operations could be used for approximate results, debugging and heuristic decisions. A ubiquitous example is a sequential data structure such as a linked list. While traversing the list in read-only mode, a concurrent write of a value, which has been already read, causes a conflict. For such cases, Scala STM provides a mechanism called *unrecorded read*. An unrecorded read is a transactional read, which returns an `UnrecordedRead` object containing the value, the version number before read, and a validity field of the read. The validity field returns true when there were no changes to the value, which helps to resolve the ABA problem [22].

In Scala STM the unrecorded read can be accessed through calling the *relaxedGet* method. By using it, we can perform a read that will not be validated during commit. The *relaxedGet* method has to be executed inside of an atomic block:

```
atomic{
  val unvalidatedValue = ref.relaxedGet({(_,_) => true})
}
```

Unrecorded reads do not yield new entries in the read set, but still need to ensure reading the latest version of a TM object. Scala STM checks for concurrent writes and forces eager conflict detection, which, in turn, causes a rollback of the writing transaction. As we see later this resolution strategy leads to similar runtime behavior as of the regular transactions.

To remove the overhead associated with unrecorded reads, we propose to provide direct access to the Ref object's data (which is safely possible due to the write-back characteristic of Scala STM). We extended Scala STM's Ref implementation with a *singleRelaxedGet* method and provide an example for generic references below.

```
class GenericRef[A](@volatile var data: A) extends BaseRef[A] {
    ...
    def singleRelaxedGet(): A = data
}
```

Its usage is then straightforward as shown in the next example:

```
val relaxedValue = ref.singleRelaxedGet()
```

As the *data* variable in *GenericRef* is marked as *volatile*, the *singleRelaxedGet* read-only operation can therefore safely interleave with other transactional write operations while guaranteeing to see the latest result. Moreover, *singleRelaxedGet* does not interfere with other transactional operations, i.e., it cannot force another transaction to roll back. This behavior is desirable in our context, since the overall idea behind using available threads was to process read-only messages such that they would not interfere with potentially more important operations (e.g., application's business logic).

5.4 Evaluation

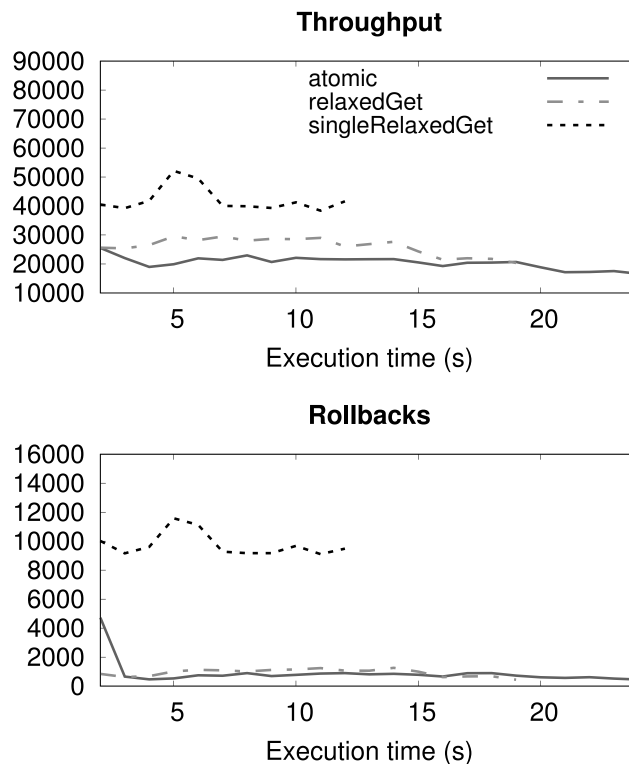

Figure 5.3: Throughput (left) and rollback count (right) for the list write-dominated workload: Static thread allocation.

Our optimizations are expected to be most useful in applications where state is shared among many actors. Hence, to evaluate our approach, we use a benchmark application

provided by Imam and Sarkar [65] that implements a stateful distributed sorted integer linked-list. It is the same benchmark as in our former work and shown in Section 5.2 in comparison to Habanero Scala. This benchmark is relevant as sequential data structures are typical applications for relaxed atomicity and isolation (see, e.g., *early release*). The read-only operation is a non-consistent sum that traverses each list element.

To show wider applicability we also consider a real-world scientific application that is used to simulate the hydraulic subsurface. The read-only operations in this application are used to control progress and debugging and hence should not interfere with any regular operation.

The thread pool is configured to support two scenarios. First, we specify a static ratio, in which 90% of threads are assigned to process STM messages and the rest processes read-only messages. In the second case we consider a dynamic ratio, but ensure that the number of threads assigned to process any message type never drops below the ratio of 10%. We compare the performance of our proposed *singleRelaxedGet* to the default *atomic* block and the *relaxedGet* method provided by Scala STM.

We execute the benchmarks on a 48-core machine equipped with four 12-core AMD Opteron 6172 CPUs running at 2.1GHz. Each core has private L1 and L2 caches and a shared L3 cache. The sizes of both instruction and data caches are 64KB at L1, 512KB at L2, and 5MB at L3.

5.4.1 List Benchmark

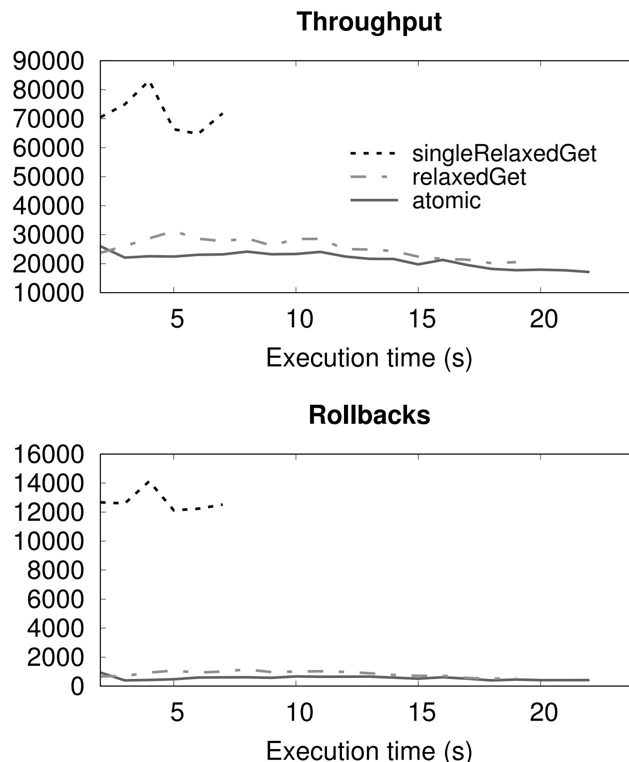

Figure 5.4: Throughput (left) and rollback count (right) for the list write-dominated workload: Dynamic thread allocation.

The list benchmark comprises two actor types: *request* and *list* actors. Request actors send requests such as *lookup*, *insert*, *remove*, and *sum*. List actors are responsible for handling

a range of values (buckets) of a distributed linked list. We implemented a list element as an object containing a *value* field and a *next* field, which is wrapped in a Scala STM Ref object.

In a list with l list actors, where each actor stores at most n elements representing consecutive integer values, the ith list actor is responsible for elements in the $[(i-1)\cdot n, (i\cdot n)-1]$ range, e.g., in a list with 4 actors and 8 entries in total, each actor is responsible for two values. A request forwarder matches the responsible list actors with the incoming requests. For the *sum* operation, we traverse each list element in every list actor. This operation is read-only and does not necessarily report a consistent value. It should not conflict with other accesses to the list.

We run the benchmark with 32 threads (to be able to divide and multiply the thread count by two) in 7 runs from which we take the median throughput and number of rollbacks for the results. Also, we create 8 list actors that are responsible for 41,216 list elements. We create 500 request actors, where each actor sends 1,000 messages to the list. After each of the request actors finished their 1,000 requests the benchmark terminates.

To read a value from the list, we consider three different options: (1) the regular transactional read (*node.next.get()*), (2) the unrecorded read accessed via (*node.next.relaxedGet()*) and (3) our direct access (*node.next.singleRelaxedGet()*) method.

In our experiments we consider a write-dominated and a mixed workload. The write-dominated workload is configured as follows: Each request actor sends 98% of requests to modify the list (insert or remove), 1% of lookup requests and 1% of sum requests.

In the first experiments, we evaluate the impact of different access approaches to the sum operation if the threads are assigned statically. The static approach (90% STM:10% read-only) reserves 3 threads (10%) for the processing of read-only messages and 29 threads (90%) for processing STM messages. Figure 5.3 demonstrates the message throughput (left) and the rollback count over time (right). The shorter the line, the better the execution time is. The *singleRelaxedGet()* outperforms the other operations with respect to both, execution time (50%) and throughput (40%). However, we observe a drastic increase of rollbacks (90%). This observation is counter-intuitive, as one would expect to have a lower number of rollbacks to achieve higher throughput. In fact, when we use *atomic* and *relaxedGet()* to implement the sum operation, we cause significantly more read-write conflicts. Scala STM resolves them by waiting for the write operations to finish to follow up with the execution of the read operations. On the contrary, when we use the *singleRelaxedGet()* operation, accessing the value of the Ref directly, we remove transactional read operations, which increases the likelihood of concurrent write operations. As a result, we get more write-write conflicts, which are typically resolved eagerly by rolling back one of the transactions.

Since the performance of transactional operations can be further improved, we dynamically determine the optimal number of threads as described in Section 5.3.1. We schedule a thread, which obtains the total number of commits and rollbacks every second, and decides the optimal number of threads to schedule for the processing of STM and read-only messages. Initially, the thread counts are randomly chosen and within the first two seconds the number of threads converges to the optimal values with the help of α. For determining α we profile the application for a short time. In the case of the list benchmark this results in $\alpha = 0.21$, hence if the commit-to-rollback ratio is below α the number of threads will be reduced else increased (multiplied or divided by two).

In Figure 5.4, we see that the throughput and rollback values for the *atomic* and *relaxedGet()* operations are not significantly different when compared to the static approach. This behavior is expected as the operations interfere with the concurrently executing transactional write operations. This causes more contention as both the STM messages and

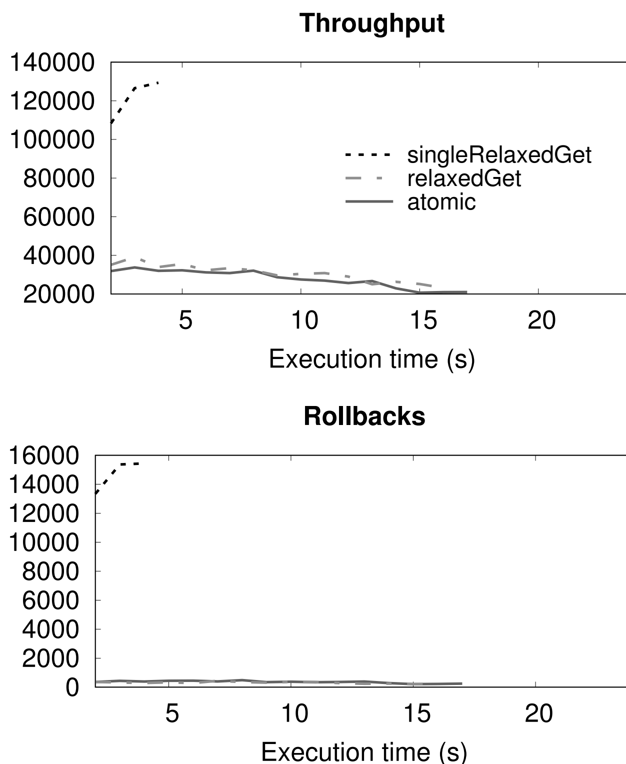

Figure 5.5: Throughput (left) and rollback count (right) for the list mixed workload: Static thread allocation.

the read-only messages are conflicting, even though we dynamically adjust the number of threads processing read-modify operation, the unused threads are also running conflicting sum operations, creating more contention. On the contrary, the *singleRelaxedGet()* operation never conflicts with other list operations. Hence, we can efficiently use the threads not processing STM messages resulting in increased throughput (65%) and reduced runtime (70%), thereby not impeding the work of other concurrently executing list operations. Also, as we dynamically reduce the number of threads processing transactional messages during high contention phases, we can schedule the leftover threads to processes sum messages. Our optimization is thus an efficient solution for reducing the overall runtime of the write-dominated list benchmark.

In the mixed read-write workload, consisting of 50% of requests to modify the list, 49% lookup requests and 1% sum requests, we show the applicability of *singleRelaxedGet* in a more read-dominant scenario. With the help of a profiling phase, we set α to 0.08. Figures 5.5 and 5.6 show similar results as the write-dominated list benchmark, amplifying the benefits of *singleRelaxedGet*. The dynamic thread assignment further reduces the rollback counts in comparison to the static assignment. In order to compare to Figure 5.1 the request ratio has to be increased, leading to results below 1 second for the *singleRelaxedGet*. It would hence clearly outperform the default implementation of Habanero Scala.

5.4.2 Simulation of the Hydraulic Subsurface

Commonly, the cost associated with the determination of the hydraulic properties of the subsurface is prohibitively high. Hence, the aim of multiple-point geostatistical simulations is to simulate the hydraulic properties of the subsurface. A number of techniques has

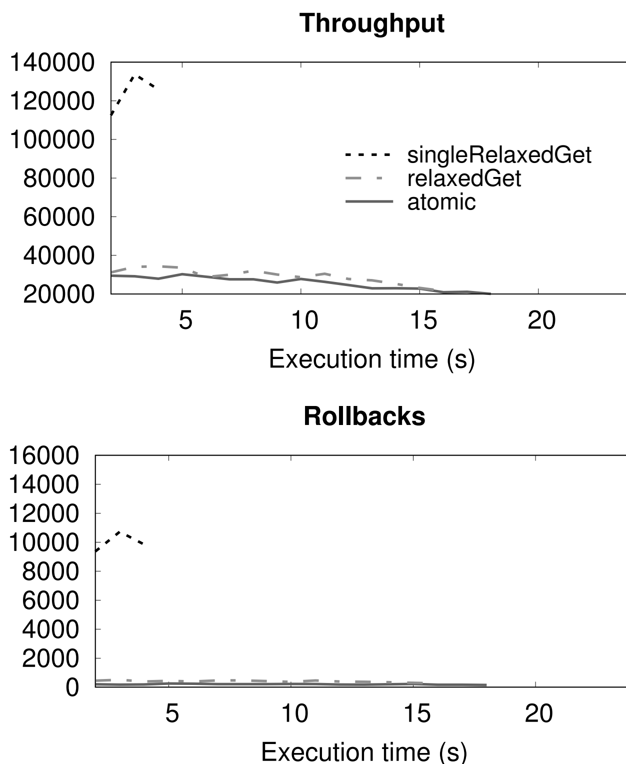

Figure 5.6: Throughput (left) and rollback count (right) for the list mixed workload: Dynamic thread allocation.

been developed for executing the simulation, the most popular of them is multiple-point geostatistics [46], which analyzes the relationship between multiple variables in several locations at a time. For its implementation, we can use the Direct Sampling simulation introduced by Mariethoz et al. [78]. A simulation consists of a Training Image (TI) and a Simulation Grid (SG). The task is to fill unknown points of the simulation grid according to the known points from the training image (see Figure 5.7(a)). The algorithm starts with selecting a random point x to be filled in the SG. Then, it locates n closest neighbors, which are already filled. The neighbors and x are considered as a pattern, for which the algorithm searches in the TI. If found, x can be filled.

Given that in the field of hydraulic subsurface simulation the simulation grid and the training image are of a very large size, sequential processing is not efficient. Two cases of parallelization are possible: (1) the parallelization of searching within the training image, (2) the parallelization of the node filling within the simulation grid. While the first case is trivial, since the training image is static and all accesses are only read operations, the parallelization of the simulation grid is more complicated. In this section, we concentrate on the parallelization of the node filling within the SG as parallelizing the TI would only comprise independent read operations. Our implementation of the actor-based simulation considers a *main actor* that stores the SG and the TI. Consider Figure 5.7(b) having two points $T1$ and $T2$ to be simulated concurrently. We can see that these are close and if simulated, are likely to be part of the n closest neighbor pattern. Thus, the point that finishes first invalidates the current simulation of the other point and a synchronization mechanism is required. In our implementation, all SG points are protected by a Scala STM Ref object, which causes a rollback once a SG point is simulated that has been visited during a nearest neighbor search. Besides the main actor we implemented a number of worker actors, either

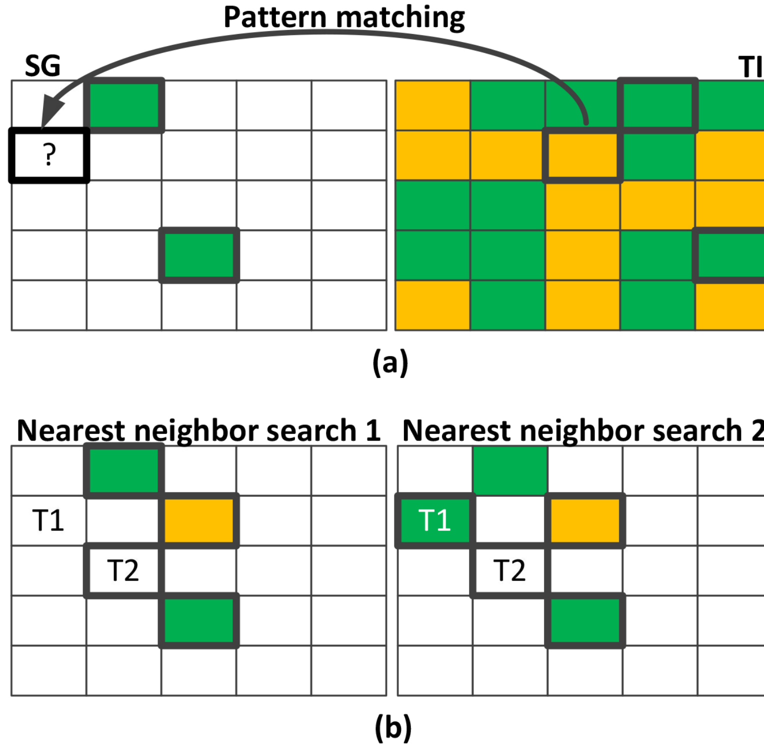

Figure 5.7: (a) Matching an SG point with the same pattern in the TI. (b) Two possibilities of closest neighbor pattern with probable invalidation of the closest neighbor selection if T1 finishes before T2.

simulating the SG points (*simulation actors*) or responsible for logging the current state (*log actors*). *Simulation actors* claim a number of points to be simulated and initiate the closest neighbor search and the TI matching for each of the points at the *main actor*. We run the benchmark with 32 *worker actors*, with each *simulation actor* requesting to process 30 simulation points at once, finally sending 167 messages during the benchmark.

Researchers using the multiple-point geostatistics simulation usually validate the results manually by visualizing the simulated result. Since the simulation can take several hours, intermediate results are of interest; we can use them to see whether the simulation is on the right track. This case is especially of interest, because once the points are simulated, they do not change anymore. Therefore, these intermediate results do not require full consistency and should not interfere with the main simulation. We select these messages to be handled as read-only messages sent by *log actors*. The list benchmark considered a constant number of sum messages. In contrast, here each *log actor* sends a request to the *main actor*, repeating it upon receiving a response. In the experiments we consider 8 *log actors*. The size of the SG is 750 times 750 elements and we investigate the 15 closest neighbors for each point, finally we profiled $\alpha = 1.2$. We use the sample image provided by the MPDS [78] distribution as the TI.

As shown in Figure 5.8 the static thread allocation results for throughput suggest that it is possible to increase the number of processed query messages with the help of *singleRelaxedGet*, while reducing the execution time in comparison to the *atomic* case by approximately 40 seconds. The total number of rollbacks, however, remains stable. Figure 5.9 (left) demonstrates that the dynamic thread allocation improves the execution time of the *singleRelaxedGet* by another 20 seconds. The throughput seems to decrease in comparison to the static scenario, which is not true considering the total throughput. While the amount

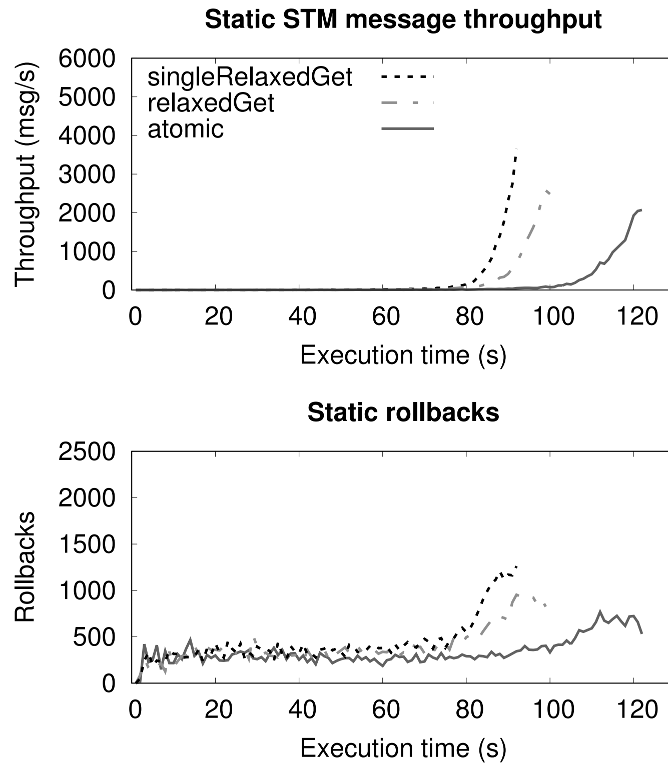

Figure 5.8: Throughput (left) and rollbacks (right) of STM messages and read-only messages over time: Static thread allocation.

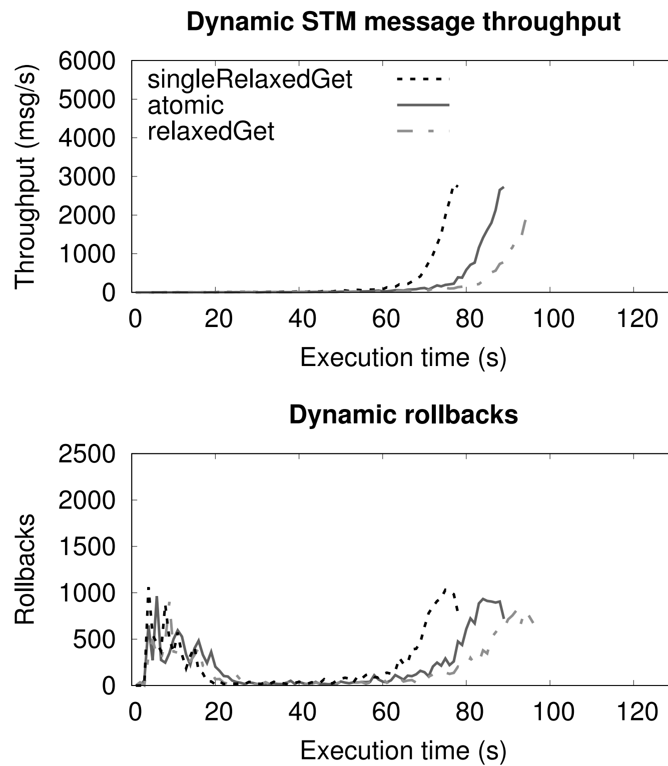

Figure 5.9: Throughput (left) and rollbacks (right) of STM messages and read-only messages over time: Dynamic thread allocation.

of work performed is in both scenarios the same, we are able to successfully process more messages in the period of second 20-50 in the dynamic scenario. This can be seen by the low rollback counts shown in Figure 5.9 (right) for the same period. In the beginning of the execution, however, the rollback counts reflect the effects of adjusting the concurrency level. In total the dynamic scenario results in lower rollback counts than the static scenario. In all of the cases *singleRelaxedGet* in combination with the dynamic handling of STM messages is able to outperform concurrent message processing.

5.5 Summary

In this chapter, we introduced dynamic concurrent message processing in the actor model for high contention workloads. We proposed to extract read-only messages from the transactional context if consistency was not required as well as adapted the number of threads to the workload. By handling transactional messages with a different level of concurrency we could efficiently use the remaining resources (threads in a thread pool) for processing read-only messages. We showed the applicability of our approach as well as candidates for messages processed with relaxed consistency. In a list benchmark and a real-world application we demonstrated that our approach helped reducing the execution time, while also decreasing the rollback counts. As a possible extension, we could improve the performance of our approach by introducing a learning phase (e.g., with the help of hill-climbing) for guiding the level of concurrency.

Chapter 6

Exploiting Heterogeneity for Energy Efficiency

6.1 Introduction

Energy efficiency of data centers and clouds has become a major concern. As claimed in 2012 by a Greenpeace report [29], current cloud computing systems consume the same amount of energy as a whole country such as Germany and India. While going *green* is from the users' and operators' perspective often done voluntarily or for economic benefits, today's systems reach physical limitations—the so-called "power wall"—that enforce focusing on energy efficiency [23].

Researchers tackle the challenge of lowering the power consumption for higher energy efficiency on both, hardware and software levels. One example is GreenLight [99], a research consortium that focuses on energy efficiency across a complete grid computing stack. The goal is to build an efficient green research facility with detailed energy profiling, monitoring, and visualization. The ParaDIME [84, 90] project has similar objectives for data centers. It focuses on software-hardware techniques to reduce power consumption, notably by using a programming model based on message passing for running computing tasks across heterogeneous resources.

Usually, work on improving energy efficiency is limited to isolated strategies. For instance, on a data center level, power consumption is reduced by adaptively shutting down nodes; on a single system level, power consumption is reduced by providing more efficient hardware or runtime support and by dynamically adapting the CPU frequency using dynamic voltage and frequency scaling (DVFS) [14]. While these approaches are effective per se, we believe that software and hardware have to be considered together to best enable energy-efficient resource usage. In general, the energy consumption E of an application relates to its power consumption P and its execution time T ($E = P \cdot T$). Hence, to reduce energy consumption, one can either radically (1) reduce the power consumption (usually at the cost of execution time) or (2) reduce the execution time (usually at the cost of power consumption).

As shown by Trefethen et al. [112] the CPU frequency has a major impact on power consumption. We therefore exploit the CPU frequency scaling features of Linux where possible and use predefined "governors".

To reduce the execution time, a possible way is to exploit all available hardware resources, e.g., graphical processing units (GPUs). Programming applications that run both on CPUs and GPUs is a challenging task, as the memory model differs and specific programming languages, such as CUDA or OpenCL, are required. Additionally, as parts of a program might be better targeted at a CPU, while other parts are data parallelizable and

run more efficiently on GPUs. As it might be necessary to provide two versions of the same application (e.g., CUDA/C++), to provide incentives for more energy-efficient software, we focus our contributions especially on programmable solutions for heterogeneous applications.

As a basis we rely on the *actor model* [58]. Besides simplifying the development of concurrent programs, the model offers a high degree of isolation between its main entities, called actors. The characteristics of the actor model improve the parallelizability and scalability of programs. Additionally, when using actors, implementation details like placement, scheduling, name resolution, buffering, etc. are transparent to the programmer, thus enabling seamless interoperability between heterogeneous components [4]. This allows us to differentiate between actors running on a CPU or GPU and consequently support *heterogeneous actors*. Actors are useful for data parallelizable applications; they, however, might cause overhead if applications are iterative and maintain state. To open the usage of the actor model to iterative applications often used in machine learning, data clustering and visual computing, some actor frameworks allow implementing stateful actors.

In this chapter, we investigate several strategies for implementing heterogeneous actors focusing on iterative applications. We start from a manually crafted and optimized implementation, in which an actor running in Scala calls the CUDA/GPU code written in C/C++ using the Java native interface (JNI). Later, we propose to decouple this design by using a middleware component, RabbitMQ.[1] While this decoupling helps simplify the programming of heterogeneous applications, it is still not fully intuitive and transparent as two different languages for the CPU and GPU are required, and interactions between heterogeneous components are explicit.

Another solution is to use a domain-specific language (DSL) for generating both CPU and GPU codes. Frameworks such as Delite [103], which provide automatic code generation, expect the entire application to be written with the DSL and executed by the provided runtime. With actors it is desirable to be able to decide on a fine-grained level whether a task, encapsulated in an actor, should execute on a CPU or on the GPU. Therefore, we adapt the actor model by introducing heterogeneous actors by incorporating concepts from the Delite framework for the efficient exploitation of GPU and CPU resources. Actors capable of running on a GPU are written in one of the DSLs provided by Delite, while other actors use a general-purpose programming language.

From a programmer's point of view we show that the heterogeneous actors based on DSLs represent the simplest solution and lead to a reduced energy consumption of up to 40% in comparison to CPU-only actor implementations, with JNI actors allowing for savings of up to 80%. These cases consider that all actors performing the work are running on the GPU, while the synchronization among these actors is performed on the CPU. However, while the GPU performs all the work the CPU might be idling, and cannot be turned off as the GPU is a co-processor. We present our final contribution, which is a scheduler that balances workload among GPU and CPU resources.

The rest of the chapter is organized as follows. We introduce the generic ideas on power consumption and execution time reduction in Section 6.2, providing implementation details on heterogeneous actors in Section 6.3. The load balancing of actor tasks is introduced in Section 6.4. In Section 6.5 we describe the hardware and software setup used for evaluation. We present and analyze results in Section 6.6. We then present the summary in Section 6.7.

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

6.2 Improving Energy Efficiency

One way to reduce the energy consumption is to decrease the power consumption ($E = P \cdot T$). This can be achieved either by influencing the hardware (e.g., by changing the frequency of a CPU), or by lowering the resource usage of the application itself (e.g., only use a single CPU with a sequential program). Another way is to focus on the improvement of the application's performance. In the following sections we discuss mechanisms for both approaches in detail.

6.2.1 Reducing Power Consumption

The overall power consumption of a machine is highly influenced by the power consumption of the CPU. Although CPUs become more and more energy-efficient, the overall energy consumption increases as we usually trade power for performance [12]. There exist several mechanisms to influence the power consumption. Here, we focus on two strategies that are readily applicable to common systems and are easy to configure: (1) the level of parallelism and (2) the voltage/frequency of a CPU.

If the level of parallelism (i.e., thread count) of an application is not properly chosen, the performance and the power consumption are negatively affected. For example, the system scheduler might interfere with the program execution, impeding the application's performance. For example, it is often possible to configure the number of threads in a concurrent application. If the number of threads exceeds the number of CPU cores, the scheduler will interfere with the program and performance might even drop below sequential processing. If the number of threads it too low, the available resources are not fully used and energy efficiency suffers as well.

The power consumption of a CPU can be influenced by changing the CPU frequency. The Linux kernel provides a tool, `cpufreq`,[2] allowing us to configure *governors* that automatically set the desired CPU frequency. Specifically, we are interested in three governors. (1) **Performance**: the CPU will be automatically set to the highest available frequency; (2) **Powersave**: the CPU will be automatically set to the lowest available frequency; (3) **Ondemand (DVFS)**: the governor monitors the CPU utilization and, if it is on average more than 95%, the frequency will be increased. The dynamic approach with the *ondemand* governor is the most promising, as it provides DVFS to fit the needs of an application.

One would suspect that simply using the *powersave* governor will be enough for being more energy-efficient. However, there is a tradeoff between power and performance and lowering the CPU frequency will have a negative impact on the execution time. The *userspace* governor requires application profiling to find the best fitting frequency. These two runtime mechanisms are easy to configure and hence can be part of the settings for service level agreements (SLAs). For instance, energy efficiency could be included as a parameter in classical deadline-driven SLAs, or serve as a basis for new cost models for data centers.

6.2.2 Reducing Execution Time

If the performance gain is significant, it can be translated into a reduction of the total energy consumption. Concurrent programming is one measure to reduce execution time. Programming with threads and locks, however, is challenging to make both efficient and correct. Furthermore, shared-memory synchronization typically limits scalability.

[2] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

As as already discussed in Section 2.4, the actor model has been introduced by Hewitt et al. [58] as a popular mechanism for implementing parallel, distributed and scalable systems. An actor is an independent, asynchronous object with an encapsulated state that can only be modified locally based on the exchange of messages. Considering a typical data-parallel

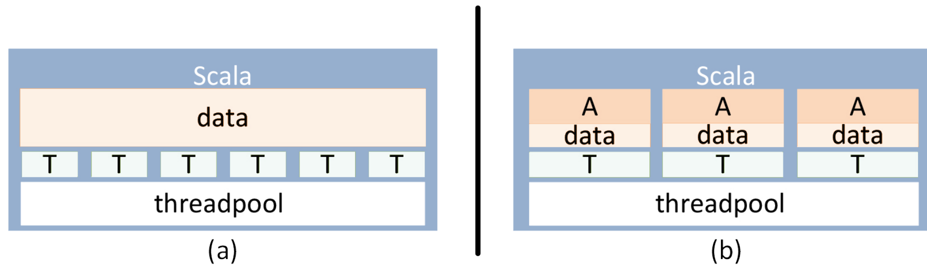

Figure 6.1: (a) Thread-based implementation, (b) task-based implementation with actors.

algorithm as an example, we can easily design an application with a set of dedicated worker actors performing the required computations and a separate coordination entity actor that distributes the data and collects the results (see Figure 6.1b). In contrast to a classical multithreading approach (Figure 6.1a), we do not need to account for synchronizing shared memory accesses. Note that a thread pool is used in both cases, yet, in the actor case, a thread represents a single actor that processes messages sequentially. For our actor implementations we use Akka[3], an official platform to manage actors in Scala.

Actors encapsulate data and tasks, and allow for interoperability not only on a single CPU but also across its boundaries. Communication, however, is not yet supported between different kinds of processors such as GPUs. GPUs, however, are already used for many parallel applications for radically reducing execution time.

6.3 Enabling Heterogeneous Actors

To reduce the energy consumption while ensuring programmability, we exploit heterogeneous computing (CPU/GPU programming) with the help of actors. For GPU programming CUDA is a de facto standard.[4] In CUDA, a function to be executed in parallel is called a kernel. Threads that execute a kernel are grouped into blocks, and threads that are part of the same block share memory and can hence cooperate. The latency to access shared memory is far lower than that that of the off-chip DRAM memory, making shared memory effective as a per-block software-managed cache. Any thread has access to the global GPU memory, but such accesses are expensive. CUDA provides a C/C++ binding for communicating with the GPU. As a GPU is a co-processor, the CPU is always necessary for communication, management, and data exchange.

If programs port data parallel parts to the GPU and keep the sequential parts on the CPU we speak of heterogeneous programming. While with C/C++ and CUDA the program would be tightly interwoven, the actor model provides inherent decoupling by separating tasks into actors. As stated before, the communication between actors on different processors is not straightforward. As such, we provide support for actors that are able to run on either a GPU or a CPU, calling them *heterogeneous actors*. Such an integration requires the properties of

[3]http://akka.io

[4]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

the actor model to be preserved. In what follows, we present three different possibilities for implementing heterogeneous actors.

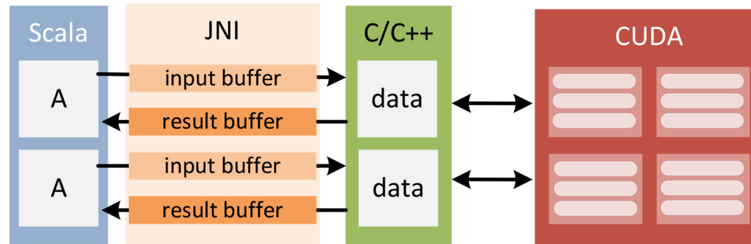

Figure 6.2: Heterogeneous actors using JNI.

6.3.1 JNI

The Java native interface (JNI) can be used for communicating with native libraries written in C/C++, supporting the communication with the GPU. In data-parallel programs, actors responsible for interacting with the GPU are initialized with a portion of input data (see Figure 6.2). A copy of the actor-local data is propagated to the actor-local GPU memory as well. This keeps the data isolated, hence no synchronization is required. In each GPU actor the final result is then stored in a result buffer, which can be accessed from either C or Scala using JNI. The same strategy is used to transfer input data. Note that, if the target system does not comprise GPUs, this implementation would not work and an alternative CPU-only implementation is required.

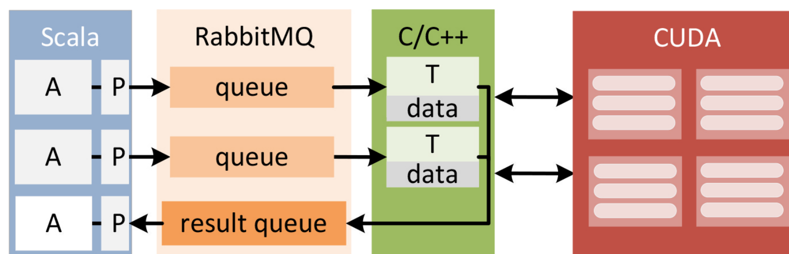

Figure 6.3: Heterogeneous actors using RabbitMQ.

6.3.2 RabbitMQ

An alternative for decoupling CPU and GPU code is to use a middleware component like RabbitMQ.[5] RabbitMQ enables the communication (via queues) between programs written in different languages and amongst distributed machines. By using RabbitMQ we can connect CPU actors with GPU actors. Communication is supported via a proxy (P in Figure 6.3(b)) that passes data from Akka to RabbitMQ. On the C/C++ side, each actor is associated with one thread (T) that waits for work in its RabbitMQ queue and, once available, fetches and forwards the data to the GPU for processing. The data is still isolated and accesses do not have to be synchronized. Upon completion, threads dispatch their result to the shared RabbitMQ result queue. The results are then collected and merged by a coordination actor in Akka. With RabbitMQ it is still necessary to provide both the CPU and the GPU

[5] http://www.rabbitmq.org

implementations. It also requires the development of custom code to interact with the communication middleware (the proxy is not part of Akka).

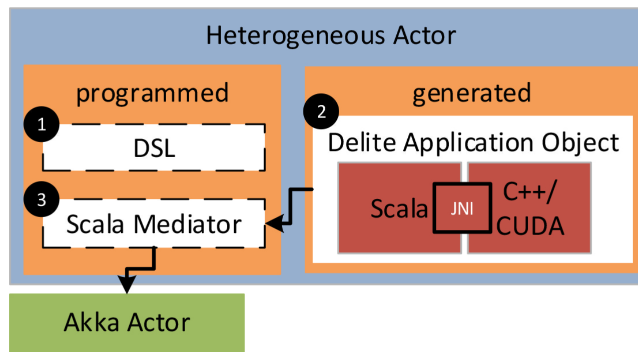

Figure 6.4: Heterogeneous actor using DSLs from the programmer's view.

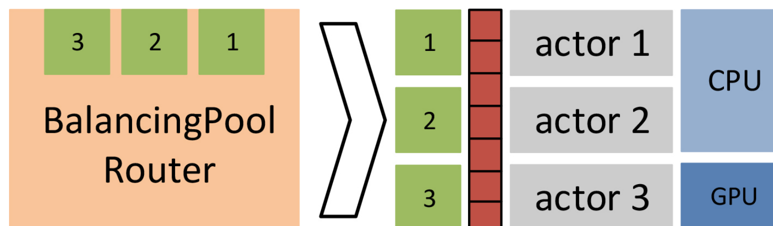

Figure 6.5: BalancingPool Router in Akka.

6.3.3 DSL

For the DSL implementation we base our efforts on Delite [104], a framework that provides high level DSLs and runtime for heterogeneous programming. Delite expects that the programmer writes the entire application in the provided DSLs and executes the generated code in a dedicated runtime environment. Just like MATLAB, DSLs leveraged by Delite and, in particular, OptiML, provide an intuitive syntax for working with vectors as well as the capacity to exploit optimized implementations of linear algebra operations. The code that follows demonstrates an implementation of the core part of k-means, a simple machine learning application written in OptiML.

```
/* An example of k-means clustering */
untilconverged (centroids, tol) { centroid =>
  // assign each sample to the closest centroid
  val clusters = samples.groupRowsBy { sample =>
 // calculate distances to current centroids
 val allDistances = centroids mapRows { centroid =>
   dist(sample, centroid)
 }
 allDistances.minIndex
  }
  // move each cluster centroid to the
  // mean of the points assigned to it
  val newCentroids = clusters.map(e => e.sum / e.length)
```

```
  newCentroids
}
```

In this example we see that OptiML's expressive syntax, which was partially inspired by Scala, facilitates performing matrix math. The code also demonstrates the use of control structures, such as *untilconverged*. These structures effectively hide implementation details and allow exploiting course-grained parallelism.

As it is not always feasible to write the entire application in a DSL, our goal is to provide finer control to the programmer such that only some parts of his application have to be written in a DSL. In particular, only heterogeneous actors will be written in one of the Scala-like intuitive DSLs provided by Delite. Delite DSLs use Scala-like syntax and are more intuitive than handcrafted GPU code. Delite does not support, however, the execution of a combination of generated GPU and hand-written code.

To support the execution of generated code from the actor environment, we need to provide custom communication facilities. Delite currently supports communication to generated Scala code with an intermediate packaging step into a *Delite application object*. To interact with this object (stored in a JAR file), so-called *Scopes* are needed as entry points [103]. They are limited, however, to mapping simple Scala data types to generated Scala code, and they cannot forward data from Scala to the generated C++/CUDA code. As a first measure, we enhanced Scopes with a JNI method for forwarding data to the generated C++/CUDA code. We further extended Scopes to automatically load the generated C++/CUDA code and enable the interaction between Scala and C++/CUDA, which was not supported by Delite.

Another limitation of Delite is the lack of support for applications that maintain state between iterations. Typically, upon start-up the generated C++ code allocates main memory and GPU memory for storing input and intermediate data. Before completion, Delite cleans all the memory that it used during its execution. We adapted Delite such that the state-relevant memory (e.g., input dataset chunks copied to GPU memory) is only cleaned after the last iteration of the application has been executed. With this measure we avoid copying data between CPU and GPU at each actor message exchange.

In Figures 6.4 and 6.5, we show the overall heterogeneous actor approach. The programmer must provide actor code targeting the GPU in the Delite DSL (Figure 6.4 (1)), which will generate and build the Delite application object (Figure 6.4 (2)). A lightweight mediation part in Scala (3) is required to convert Akka messages into data structures for the Delite application object and vice versa. We also provided support for Delite-generated CUDA code to return the result to the calling Scala code.

6.4 Resource Load Balancing with Heterogeneous Actors

In the previous section we introduced the generic principle of heterogeneous actors. Since the CPU and the GPU have different performance characteristics, load imbalances can happen. This can lead to CPU actors waiting for GPU actors to finish work, or vice versa. Hence, this section focuses on efficient workload balancing strategies for runtime and energy reduction.

To distribute work among actors on a CPU, Akka provides so called *Routers* that schedule messages targeted to a set of actors accomplishing a similar task. Specifically, the

BalancingPool router embraces “work-stealing”[6] by balancing workload dynamically among worker actors. When an actor accesses its mailbox to fetch the next available message to be processed, Akka transparently forwards that request to a shared message queue started by the BalancingPool Router (see Figure 6.5). Since the mailbox queue is shared, any worker actor should be capable of processing any message in the queue. Hence, Akka imposes a requirement for worker actors to be stateless, thus limiting its usage for iterative applications.

To overcome this limitation, we propose to use the following strategies. First, to enable iterative applications to be used with routers, we encapsulate all state required for the execution into messages. Each message contains the required context for having it processed on either the GPU or the CPU. For example, to avoid copying input data on each iteration, we store a pointer to it in a message. Also, to avoid synchronization issues between the CPU and the GPU memory, the message also contains a result object, stored in CPU memory, to which all implementations write intermediate results for the next iteration.

To avoid multiple actors from accessing the same data, Akka requires that messages are immutable. We relax this requirement for messages targeting the BalancingPool router, making sure not to keep copies of the data packed in these messages elsewhere in application code. When a stateless worker actor instantiated by a router receives such a message, it will have all the required data for executing its task. Note that the message itself only contains pointers to the data to avoid communication overhead.

For actors running on both processing units, both implementations are required and any of the strategies discussed in Section 6.3 can be used. Also, since for iterative applications, the behavior will be repetitive, it is likely that the number of actors running on a CPU/GPU will not change at runtime. Hence, it is sufficient to find the optimal actor CPU/GPU configuration at startup. As such, at application start we introduce a brief profiling phase. For each configuration (e.g., 0 GPU actors/8 CPU actors; 1 GPU actor/7 CPU actors; etc.), we measure the execution time using 1% of messages to be processed. Once finished, we select the configuration with the lowest execution time and use it for the processing of remaining messages.

To summarize, our approach enables actors, irrespective of whether they run on the CPU or GPU, to request work when required, thus leading to reduced idle time and more balanced workloads.

6.5 Experimental Setup

Hardware. Our experiments are executed on a server, the characteristics of which are listed in Table 6.1. For the CPU we can experiment with five frequencies as well as use a number of governors as listed in Section 6.2.

CPU	AMD FX-8120, 8 cores, no hyperthreading, 8GB RAM
GPU	Nvidia GeForce GTX 780 Ti, 2880 CUDA cores, 3GB RAM
Power meter	Alciom PowerSpy v2.0

Table 6.1: Hardware characteristics of the experimental environment.

Software. We base our evaluations on the well-known k-means [6] algorithm used for splitting an input dataset into different clusters. K-means is a good case study as it exhibits

[6]The actual implementation more precisely follows a *work-sharing* approach.

iterative and processing-intensive characteristics representative for data-parallelism and is part of numerous benchmarks in different fields of research (e.g., STAMP [24], Rodinia [25], Minebench [81]). We further focus on k-means as it is a well-understood algorithm that can be represented in a straightforward manner in Delite's OptiML DSL. As such, we chose depth over breadth regarding our analysis, presenting the results of k-means only. Despite exclusively focusing on k-means, the core premise of heterogeneous actors is applicable for implementing other iterative algorithms (e.g., coordinate descent, logistic regression, deep belief learning with a restricted Boltzmann machine).

The baseline k-means algorithm has been introduced by Alpaydin [6]. and its basic steps are summarized in the following Algorithm 1.

Algorithm 1: The baseline sequential k-means logic.

Data: input set, number of clusters
Result: clusters
Initialize K cluster centers;
while *Termination condition is not satisfied* **do**
    Assign input points to the closest cluster center;
    Compute final cluster centers for this iteration;

We used the thread-based STAMP [24] implementation of k-means as a basis for creating the actor version. For the actor-based implementation the following data structures are required: (1) input matrix; (2) current cluster center matrix; (3) points to cluster center *map* (holds the current cluster center index for each input point); and (4) per-cluster member count structure (holds the number of points assigned to each cluster).

6.5.1 Parallel Implementation with Actors

For the actor-based implementation, we retrofitted the aforementioned thread-based version of k-means. Our actor-based algorithm uses two types of actors: *iteration actors* (Algorithm 2) and *worker actors* (Algorithm 3). While a typical thread-based version maintains a shared copy of the current cluster center matrix and the per-cluster member counts, the actor-based version maintains a private copy of these data structures in each of the worker actors. The iteration actor then merges the data sent by each of the worker actors to calculate the final cluster centers. A more detailed explanation of the described processes as well as the algorithm is described in what follows.

1. Each worker actor claims responsibility of a fixed number of rows of the input matrix. As one of the characteristics of the actor model is isolation of resources, the input to be processed by the worker actor has to be copied into its local storage.
2. The worker actor then instantiates a version of k-means that processes a sub-range of the received input but does not calculate the final cluster centers. Specifically, for each point to be processed, it locates the closest cluster center and adds the points to the new cluster centers matrix as well as updates the cluster member count map.
3. After each worker actor assigns its local points to appropriate clusters by populating its local cluster centers matrix and the cluster member count map, it sends them to the iteration actor.

4. The iteration actor merges all partial solutions received from the worker actors and calculates the final cluster centers for the current k-means iteration. The merging of partial results is done in a straightforward way. The iteration actor holds the global cluster centers matrix and the cluster member count map. The iteration actor performs merging by adding the worker actors' local cluster centers matrix to the global cluster centers matrix, and the worker actors' local cluster member count map to the global member count map. Once the iteration actor has received and merged all the worker actors' local data structures, it calculates the mean of the cluster centers.
5. If the termination condition is not met, the worker actors base their work on the updated cluster centers matrix.

Algorithm 2: K-means iteration actor logic.

```
Data: input set, number of clusters, number of workers
Result: clusters
Initialize K cluster centers
foreach Worker do
  | Create workers and pass partial input set
while Termination condition is not met do
  | Send current cluster centers to worker actors
  | foreach Worker do
  |   | Receive partial results
  | Compute final cluster centers by merging partial results
```

Algorithm 3: K-means worker actor algorithm.

```
Data: partial input set, current cluster center
Result: local cluster centers, local member count
foreach Assigned input point do
  | Assign point to the closest cluster center
  | Update the local cluster centers matrix and member count
Send local cluster centers and cluster counts to iteration actor
```

6.5.2 Heterogeneous Implementation with JNI

For the heterogeneous implementation we extend the baseline actor implementation. Specifically, we execute the worker actor code on the GPU, while leaving the iteration actor unchanged for execution on the CPU. We further preserve the communication patterns between worker actors and the iteration actor. We reimplemented the worker actor to access the GPU resources by calling the C/CUDA code using JNI with the help of shared byte buffers as shown in Figure 6.2(a). For communicating data between Scala and C, we use a shared byte buffer, the contents of which are visible in both, Scala and in C. Hence, the only remaining part in the worker's Scala code is responsible for reading/writing values to/from shared byte buffers.

Worker actors connect to a C implementation that start two CUDA kernels, one for finding the closest cluster center for each input point (on block memory), and one for finding the total number of points that changed clusters as compared to the previous iteration (on

GPU global memory). Each kernel is mapped to CUDA blocks on the GPU—where each block comprises 128 threads to support multiple architectures. The number of blocks per actor depends on the size of the input data set. Within a CUDA block each CUDA thread is responsible for processing a single input point. CUDA threads are working independently from each other, but in the end the shared memory of a CUDA block is used for calculating block-local results. Once the GPU execution has finished, the results are transferred to the result buffer and to the iteration actor.

6.5.3 Heterogeneous Implementation with RabbitMQ

In this implementation we reused the CUDA code of the worker actor from the JNI implementation, but adapted the communication pattern between Scala and C/CUDA. Each worker actor now includes a proxy (as shown in Figure 6.2(b)) that is responsible for marshaling the messages and sending them to the RabbitMQ queue.

The C implementation launches the same number of threads as the number of worker actors, where each thread waits for work to be published to a predefined queue. Once work is available, the aforementioned CUDA implementation is launched, omitting the shared byte buffers. In the end, a proxy actor connecting to the iteration actor transfers the results.

6.5.4 Heterogeneous Implementation Using a DSL

We define the worker actor's logic using OptiML [102]—a Delite DSL. In contrast to the RabbitMQ and JNI implementations, Delite automatically generates GPU code, which avoids handcrafting and manually optimizing CUDA code. However, the ratio of code that runs efficiently on the GPU is lower than in the case of JNI and RabbitMQ.

As an example we show below the code of the `findNearestCluster` function written with the OptiML DSL. It matches an input point p to cluster centers $clusters$ and calculates the Euclidean distance. The output is the cluster index that the point p matches best.

```
private def findNearestCluster
(p: Rep[DenseVectorView[Double]], clusters: Rep[DenseMatrix[Double]])
: Rep[Int] = {
    (clusters mapRowsToVector { row =>
       dist(p, row, SQUARE)}).minIndex}
```

We then write the mediation code to connect to the generated code (Figure 6.4). The mediation code extracts the current cluster centers from an Akka message, converts them to a Delite array (to map the `Rep` data structures in the DSL), and then calls that generated code with the array as input. Once the result is available, the mediation code converts it to an Akka message and forwards it to the iteration actor.

6.5.5 Heterogeneous Work Balancing Implementation

For the implementation of the work balancing use case any of the before mentioned implementations can be used. We decided to use JNI as it showed the best performance (see Section 6.6). We define the worker actor code just like in the heterogeneous implementation with JNI, but the worker actors are able to execute on both the CPU and the GPU. To enable load balancing, we require stateless actors, hence moved their state to messages (see Section 6.4). The profiling uses 1% of the overall workload for testing each possible configuration, hence we

allow the programmer to set the number of desired iterations manually. We however, move all the state out of worker actors and move it to messages, which we pass around between the Iteration actor and Worker actors. Also, in the previous implementations, during each iteration, the Iteration actor sends one message to each Worker actor, collecting responses in the end.

Given this setup, the work stealing implementation would never be able to steal unprocessed tasks, since each Worker actor will have exactly one task to process during each round. To enable work stealing, on each iteration the Iteration actor sends 4 times more tasks then the number of Worker actors, leaving unprocessed tasks in actors' mailboxes for a longer period of time. Lastly, as the baseline k-means implementation converges relatively quickly (around 10 seconds), we used a version of k-means where we manually set the number of iterations that have to be executed in order for the application to converge.

6.5.6 System Configuration

K-means is a representative candidate for this evaluation as it is able to work on different input sizes. For the first three implementations of k-means (CPU/GPU), we chose a default data set from the STAMP benchmark with 65,536 input rows and 16 clusters. To test the profiling and selection process of the best share of CPU/GPU actors we use three different datasets: *small* (4,096 rows), *medium* (10,240), and *large* (131,072). We set the worker actor count to match the CPU core count (i.e., 8).

To enable efficient load balancing, the iteration actor divides the work into more tasks than the number of worker actors (32 tasks per iteration). As the run times can be considerably reduced with a GPU, we increased the load to gather reasonable results. The profiling takes 450 iterations per configuration, with an overall benchmark length of 5,000 iterations. We run each implementation 5 times and take the median execution time and power readings; the energy is then calculated out of these two values.

6.6 Results and Discussion

In this section, we discuss the results of the different k-means implementations from Section 6.5 with respect to power consumption and execution time, and relate them to energy consumption. Although we consider k-means as our main application, the results provide us with the capacity to evaluate trends for data-parallel applications.

6.6.1 Reducing Power Consumption

We investigate the reduction of power consumption by varying the number of workers, as well as the governors impacting the frequency of the CPU. The simplest way of influencing the CPU frequency, is to use different governors. We first discuss the results of the different governors, which can be easily configured, and later focus on fixed frequencies. The default governor is *ondemand*; its goal is to provide good performance when work is available and downscaling of the frequency otherwise (DVFS).

On the left side of Figure 6.6 we present the power consumption of the three CPU-only Scala implementations (seq: sequential, par: parallel thread-based, act: actor). We scale the number of threads/actors (4, 8, 16, 32) in separate runs and average the results for each chosen frequency. The sequential implementation consumes the least power since only one core is used while the others are idling. When using the powersave governor the difference

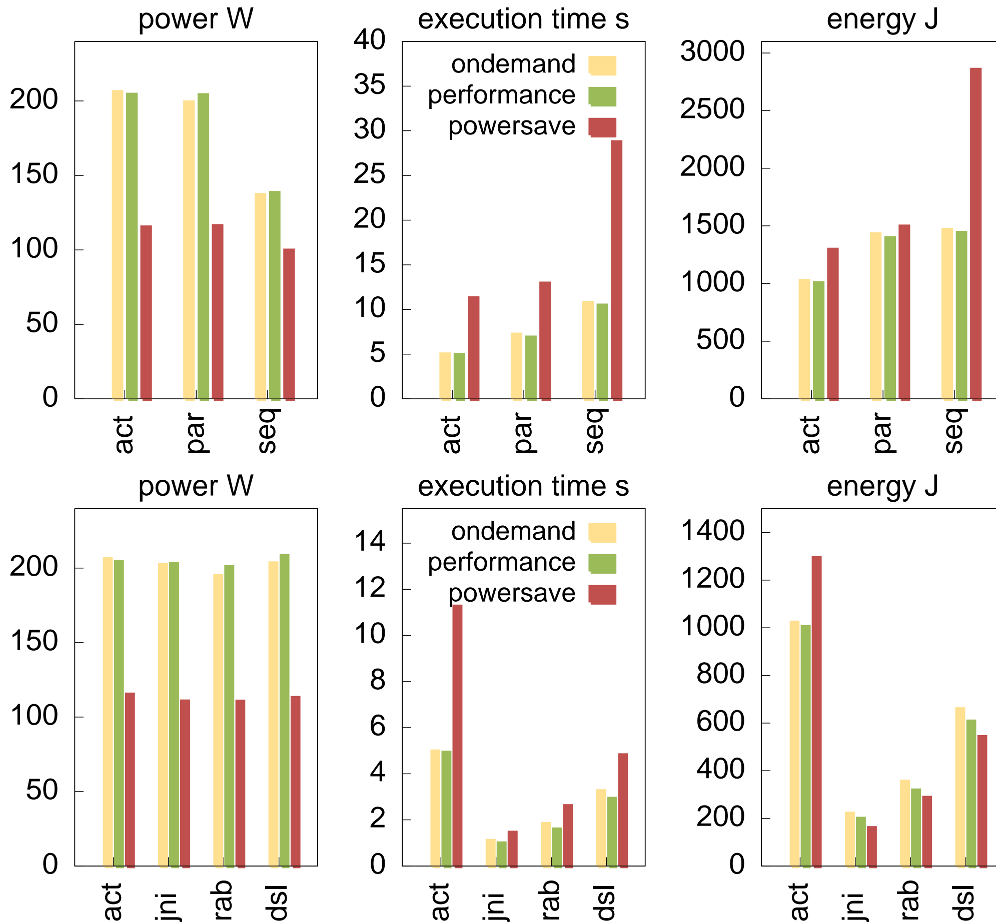

Figure 6.6: Comparison of the power consumption (left), execution time (middle), and energy consumption (right) using different governors. The top graphs refer to CPU execution. The bottom graphs include one bar for CPU-only execution and three bars for mixed execution (CPU/GPU).

between the parallel and actor implementation is less than 10 W. The *ondemand* governor depends on CPU utilization and, since k-means is CPU-intense, power consumption of the *ondemand* and *performance* governors is comparable. The *powersave* governor sets the CPU to the lowest frequency, hence power consumption is reduced. The difference between the *powersave* and other two governors is 70 W for the parallel implementations and around 40 W for the sequential implementation.

The middle part of Figure 6.6 shows the impact of the governors on the execution time. The sequential algorithm using the *powersave* governor is about 3 times slower than with any other governor. The parallel implementations exhibit a slowdown of about 2 times if the *powersave* governor is used.

Based on power and execution time measurements, we can compute the energy consumption as shown in Figure 6.6 (right). The actor implementation outperforms the parallel and the sequential implementations. We can further see that, while the *powersave* governor decreases the power consumption, the execution time is significantly higher. This leads to higher energy consumption than when using the *ondemand* and *performance* governors. In general, the *ondemand* governor seems to be the best choice independent of the type of implementation.

6.6.2 Reducing Execution Time

This section focuses on execution time reduction and its impact on energy consumption. In the heterogeneous implementation, CPU actors cooperate with the GPU in different ways. We compare the JNI implementation (jni) with RabbitMQ (rab) and DSL actors (dsl) as described in Section 6.5. We also vary the frequencies for the CPU running the remaining code.

Figure 6.6 shows the power consumption (left), execution time (middle) and energy consumption (right). We see that the power consumption is not impacted by the usage of the GPU. All GPU implementations execute faster than CPU implementations, yielding lower total energy consumption (Figure 6.6 (right)). The DSL implementation in *powersave* mode consumes 540 J, which is lower than the 1,001 J of the actor implementation in *ondemand* mode. In contrast to the CPU-only execution, we see that reducing the CPU frequency with the *powersave* governor does not have a drastic impact on performance. Therefore, the best choice would be the *powersave* governor in the heterogeneous scenarios. Interestingly, the *ondemand* governor leads to a higher energy consumption than the *performance* governor. We can see that the power consumption of the implementation in *ondemand* mode is lower than in the *performance* mode, which means that the chosen frequency is suboptimal.

With respect to the different implementations, JNI provides the most direct way of communication with the GPU. This implementation does not provide the decoupling nor the flexibility for seamlessly exchanging the code to be executed on the GPU or the CPU. While the RabbitMQ implementation provides the possibility of exchanging worker (GPU) code, it still requires the programmer to implement the actual GPU code in C/CUDA. In comparison, by using heterogeneous actors with the DSL, programmers only need to provide the worker code for message processing in one of the Delite DSLs.

A measure of the effort for the programmer is shown in Figure 6.7, where we can compare the lines of code required to write the k-means GPU logic and the communication logic on the C/C++ side for the different implementations. Whereas the JNI implementation is the most energy-efficient one, it requires 336 lines for the k-means logic written in CUDA and C, as well as 67 lines for providing JNI functionality (communication logic). RabbitMQ uses the same k-means logic as JNI, but requires another 337 lines for the communication logic. In comparison, our DSL implementation does not require communication logic because it is automatically generated by Delite. Hence, the overall effort for the DSL implementation is as low as 39 lines of code.

In terms of the lines of code required to implement k-means, JNI implementation is the most energy-efficient one, it requires 336 lines for the k-means logic written in CUDA and C, as well as 67 lines for providing JNI functionality (communication logic). RabbitMQ uses the same k-means logic as JNI, but requires another 337 lines for the communication logic. In comparison, our DSL implementation does not require communication logic because it is automatically generated by Delite. Hence, the overall effort for the DSL implementation is as low as 39 lines of code.

The marginal performance loss when using DSLs can be explained by a lower ratio of code executed on the GPU due Delite not being able to fuse all GPU operations in one kernel execution. As the Delite-generated GPU code[7] used in the actor model context is still substantially more energy-efficient than a plain CPU implementation, we conclude that the heterogeneous actor model with DSLs provides a good tradeoff between implementation simplicity and energy efficiency. However, from an energy efficiency point of view heterogeneous

[7]Note that at deployment time we can generate not only CUDA, but also OpenCL and C++ code.

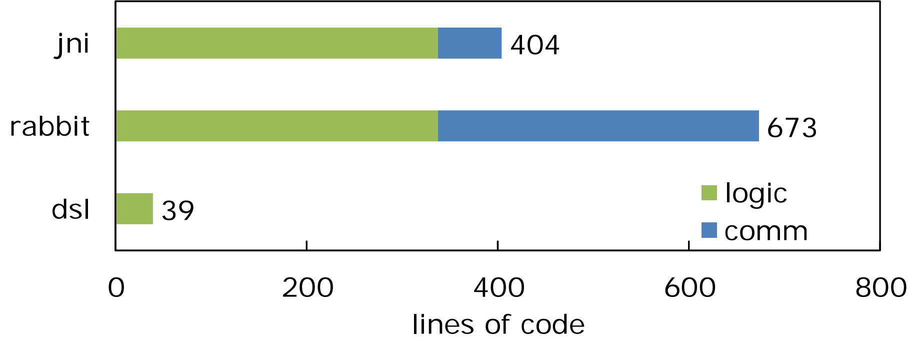

Figure 6.7: Programmability of different heterogeneous implementations.

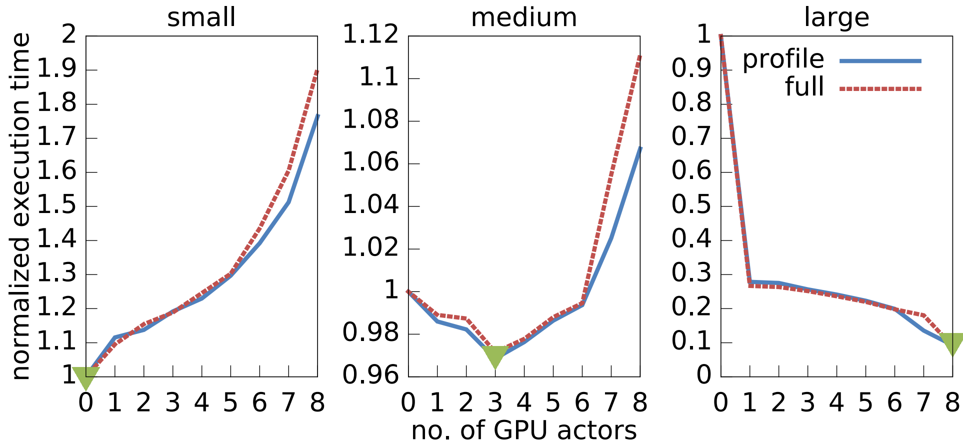

Figure 6.8: Normalized execution time of the profiling phase as well as of the remaining workload using different configurations of CPU/GPU actors.

actors with JNI are preferable, as with it it is possible to reduce the energy consumption of up to 80% in comparison to a solution running on the CPU.

6.6.3 Resource Load Balancing with Heterogeneous Actors

This section presents our proposed load balancing approach. We execute the profiling phase as well as the full run on all datasets and combine the prediction capability of the profiling phase. Clearly, the execution time of the full running phase will be a multitude higher than the execution time of the profiling phase. Therefore, we normalize all values to the execution time of the first configuration. Figure 6.8 presents the results for different dataset sizes showing that the profiling phase can mimic the execution time of the remaining workload reasonably well.

In more detail, for the *small* dataset we see that the problem is not scaled significantly to amortize overheads associated with executions on a GPU. When using the *medium* dataset we see that the most efficient configuration is composed of 3 GPU actors and 5 CPU actors. When we use a *large* dataset (e.g., also the STAMP sample dataset from our former experiments), it is always beneficial to process all the workload using GPU actors.

The energy consumption reveals the same trends, however, we further noticed that with increased load, the CPU tends to use its turbo frequencies and hence draws about

20 W more than in the experiments before. In the case of the *large* dataset the CPU has a significantly higher power value if it shares work with the GPU while the execution time is not reduced significantly. Therefore, from an energy consumption point of view any sharing of work with the CPU in our hardware setup would be disadvantageous. With the *medium* and *small* datasets the load is not as high, hence sharing the work between the CPU and the GPU leads to slightly increased power consumption but lower execution time, and in total to lower energy consumption.

6.7 Summary

In this chapter, we tackled the problem of reducing energy consumption of parallel programs in heterogeneous environments. As energy depends on both power consumption and execution time, we investigated the impact of each independently. We first reduced the power consumption with the help of frequency scaling. We then reduced the execution time by running parts of an application on a GPU, while the sequential parts remained on the CPU. We evaluated a number of strategies for heterogeneous actors regarding their energy efficiency and programmability. JNI and RabbitMQ provided a more direct way of accessing a GPU, while the DSL implementations provided a concise and simple way for building heterogeneous actors.

In the first step, all heterogeneous implementations required manual assignment to the best processing unit. Hence, our final contribution enabled automatic sharing of resources among actors yielding the highest energy efficiency. Our contributions lead to significant reductions of energy consumption in the range of 40-80% as compared to CPU-only implementations. Future work may include adjusting the profiling duration based on the input size, as well as tailoring the profiling phase to measure energy consumption for different settings.

Chapter 7

Conclusion

Up until recently, the cloud computing community was exclusively focused on performance optimizations. Energy efficiency, however, needs to be treated as a first-class citizen in the context of effective data center operation. There have been numerous improvements in reducing the energy consumption on the hardware level. The lack of energy-efficient software in practice suggests missing incentives for energy-efficient programming. In this thesis, I therefore argued that the software-based methods for energy efficiency have not received as much attention as hardware-based methods. Chapters 4 and 5 discussed how transactional memory could be used to adapt message passing models for speculative concurrent message processing in a single actor in a controlled manner. Chapter 6 presented the heterogeneous actor model, which enabled users to exploit GPU resources as seamlessly as CPU resources. We argued that the heterogeneous actor model provided opportunities to directly exploit GPU resources to cater energy efficiency.

In this chapter, I summarise the contributions of this thesis and conclude by presenting possible future directions, in which this work could be extended.

7.1 Summary of Contributions

While the message processing strategy of the actor model is important for preserving application safety, it is arguably too conservative in its default concurrency settings that impede its performance and energy efficiency. In Section 1.2, I identified a number of problems with the actor model. In what follows, I present the main contribution of this thesis by first briefly outlining the identified problems, and then the key insights of the proposed solutions.

The *first problem* that I identified was **the message queuing delay during coordination**. The queuing delay was caused by the requirement to spawn a distributed transaction during coordination, which we called *a coordinated transaction.* I argued that the blocking of actors i.e., the coordination delay, was not necessary in some cases, and could be hidden. I showed how speculative message processing, with the help of transactional memory [55], could significantly improve actors's message processing performance

The *second problem* that I identified was **the sequential message processing**. The requirement for sequential processing of messages was inevitable in earlier distributed systems where each physically separate computer could perform a single task at a time. Today, however, with the multi-core architectures each computer is capable of performing multi-tasking, this requirement is too restrictive. To concurrently perform message processing on a multi-core machine, I also used transactional memory. Transactional memory provided safety guarantees for concurrent applications, enabling us not to break the actor semantics.

The *third problem* that I identified was **the degradation of message processing performance during the concurrent processing of messages in high contention phases**. Introducing concurrency within an application, i.e., concurrent processing of actor messages, increased performance only if the concurrently executing entities did not interrupt each other often. Generally, the sources of these interruptions are application and workload specific, such as the contention on accessing common resources or the communication between executing entities. If the interruptions are frequent, the overhead engendered due to the interruptions could even result in worse performance than sequential execution. As such, we proposed an adaptive message processing model that dynamically switched between modes of execution by adapting the number of concurrent message processing in a single actor according to the contention levels.

Finally, the *last problem* that I identified was **the inability of the actor model to seamlessly exploit GPU resources**. We were motivated to use GPUs in the actor model since GPUs are compelling for the energy-efficient execution of data-parallel tasks. To enable the users to exploit GPU resources as seamlessly as CPU resources we investigated several strategies for implementing heterogeneous actors focusing on iterative applications. We started from a manually crafted and optimized implementation, using the Java native interface (JNI). Later, we proposed to decouple this design by using a middleware component, RabbitMQ. And finally, we proposed a solution which used domain-specific language (DSL) for generating both CPU and GPU code. By using this design, actors capable of running on a GPU were written in one of the DSLs provided by the Delite framework, while other actors used a general-purpose programming language. From a programmer's point of view we showed that the heterogeneous actors based on DSLs represented the simplest solution and led to a reduced energy consumption of up to 40% in comparison to CPU-only actor implementations, with JNI actors allowing for savings of up to 80%.

7.2 Future Directions

The presented outcomes of this thesis motivate additional ideas in the directions of extending the actor model for enhanced performance and energy efficiency. In the following sections, for each of the main contributions we list possible future directions.

7.2.1 Concurrent and Speculative Message Processing

The work on concurrent and speculative message processing in Chapter 4 can be improved by enforcing stronger guarantees on the message processing order. Consider an example, in which we have two actors: actor A and actor B. Further consider that actor A sends a message m_1 to actor B and, later in time, actor A sends another message m_2 to actor B. In such a scenario the actor model does not enforce a requirement for message m_1 to arrive before message m_2. Since we use transactional memory for processing messages concurrently in the actor model, we rely on the absence of guarantees on messages ordering. Consider the previous example again. When processing messages m_1 and m_2 concurrently with the help of transactional memory, one message processing can be rolled back due to conflicts, letting the second message proceed executing until finished. As stated, since we rely on the absence of message ordering guarantees in the actor model, all possible executions of message processing are valid.

The Akka framework, on the other hand, provides stronger guarantees on message ordering then the actor model. Specifically, given two actors in the system, messages

dispatched from one actor to another are guaranteed not to be received out of order. Recent work on transaction ordering [94] could be used as a base for enhancing concurrent message processing to provide message ordering guarantees, e.g., messages that have arrived earlier should be given priority.

7.2.2 Dynamic Message Processing

The work on dynamic message processing, discussed in Chapter 5, is driven by a predefined threshold α, based on which the level of concurrency is selected. The α threshold is dependent on the knowledge of the current number of commits and rollbacks of transactionally processed messages and their ratio. This parameter thus far, however, has to be selected manually. If incorrectly selected, the performance of message processing can be impeded. Therefore, dynamic message processing could be enhanced by enabling the automatic selection of the α threshold.

Also efforts could be targeted at improving the decision algorithm so that it is more stable. At the moment, the number of concurrent messages processed is very frequently modified leading to a lot of fluctuations. It is desirable to obtain a version of the algorithm that changes the number of threads less frequently. The challenge here is to keep the algorithm's tracking nature such that it does not only find the optimal number of messages that should be processed for the current workload but also keeps track of the variation in workload, so that if the workload changes it readapts itself to converge to a new optimal value.

Additionally, at the beginning of an application we started with a random number of threads from the thread pool to process transactional messages. We could provide faster convergence towards optimal number of messages (satisfying maximal throughput) by identifying a near optimal value based on the prior knowledge of workload characteristics. Furthermore, we can introduce energy consumption as a second parameter to the decision process. This can enable us to find a single metric that we can use to reason about both performance/throughput and energy.

7.2.3 Heterogeneous Actor Model

The work on heterogeneous actor model from Chapter 6 could be extended by enabling actor code to be specified in both, Scala and DSL code. Currently, in the developed system, actor's code cannot be mixed and needs to be specified either in Scala or in the DSL code. Also, the Delite framework, which we use for the DSL support, enables the use of the Apache Mesos Framework [59]. Since the Heterogeneous actor model considered running the actor code on one server node only, it would be of interest to see how we could make use of the facilities provided by Mesos (e.g., multi-resource scheduling and fault-tolerance) to enable executing actor tasks in a collective pool of computing resources.

An additional measure which could be introduced in the context extending the actor model is the introduction of energy measurements within the system. Measurement should be done within the measurement phase of the actor algorithm as well in the other phases in order to be able to reason the energy overhead of the decision phase (i.e., considering measured metrics and determining what to do next to adapt the system towards the performance/energy consumption goal) and adaptation (i.e., configuration of the application with a different number of actors). This would also potentially require retrofitting the benchmarks to perform energy measurements.

Appendix A

Publications

1. Yaroslav Hayduk, Anita Sobe, Derin Harmanci, Patrick Marlier, Pascal Felber.
Speculative Concurrent Processing with Transactional Memory in the Actor Model.
In *OPODIS 2013: 17th International Conference on Principles Of Distributed Systems*, pages 160–175, Nice, France, Dec 2013. Springer.

2. Santhosh Kumar Rethinagiri, Oscar Palomar, Anita Sobe, Thomas Knauth, Wojciech M. Barczynski, Gulay Yalcin, Yarco Hayduk, Adrián Cristal, Osman S. Unsal, Pascal Felber, Christof Fetzer, Julien Ryckaert, Gina Alioto.
ParaDIME: Parallel Distributed Infrastructure for Minimization of Energy.
In *DSD 2014: 17th Euromicro Conference on Digital System Design*, pages 191–198, Verona, Italy, Aug 2014. IEEE.

3. Yaroslav Hayduk, Anita Sobe, Patrick Marlier, Pascal Felber.
Dynamic Concurrent Message Processing with Transactional Memory in the Actor Model.
In *TRANSACT 2014: 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, Utah, USA, March, 2014.

4. Yaroslav Hayduk, Anita Sobe, Pascal Felber.
Dynamic Parallel Message Processing with Transactional Memory in the Actor Model.
In *DMTM 2014: Joint Euro-TM/MEDIAN Workshop on Dependable Multicore and Transactional Memory Systems, In conjunction with HIPEAC 2014*, Vienna, Austria, January 2014.

5. Yaroslav Hayduk, Anita Sobe, Pascal Felber.
Dynamic Message Processing and Transactional Memory in the Actor Model.
In *DAIS 2015: 15th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 94–107, Grenoble, France, June 2015, Springer.

6. Oscar Palomar, Santhosh Kumar Rethinagiri, Gulay Yalcin, J. Rubén Titos Gil, Pablo Prieto, Emma Torrella, Osman S. Unsal, Adrián Cristal, Pascal Felber, Anita Sobe, Yaroslav Hayduk, Mascha Kurpicz, Christof Fetzer, Thomas Knauth, Malte Schneegaß, Jens Struckmeier, Dragomir Milojevic.
Energy minimization at all layers of the data center: The ParaDIME project.
In *DATE 2016: 2016 Design, Automation & Test in Europe Conference & Exhibition*, pages

684–689, Dresden, Germany, 2016. IEEE.

7. Yaroslav Hayduk, Anita Sobe, Pascal Felber.
Enhanced Energy Efficiency with the Actor Model on Heterogeneous Architectures. *Best Paper Award.*
In *DAIS 2016: 16th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 1–15, Heraklion, Crete, June 2016, Springer.

References

[1] Evan Ackerman. Heat your house with someone else's computers, (2014). In *IEEE Spectrum*. `http://spectrum.ieee.org/energywise/energy/the-smarter-grid/heat-your-house-with-someone-elses-computers`. Accessed on May 24, 2016.

[2] Gul Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, 1990.

[4] Gul Agha. Actors programming for the mobile cloud. In *ISPDC 2014: Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Computing*, pages 3–9, Marseilles, France, June 2014. IEEE Computer Society.

[5] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] Ethem Alpaydin. *Introduction to Machine Learning*. MIT press, 2004.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'67 (Spring): Proceedings of the Joint Computer Conference*, pages 483–485, Atlantic City, New Jersey, USA, April 1967. ACM.

[8] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. *Robust Adaptation to Available Parallelism in Transactional Memory Applications*, pages 236–255. Springer, 2011.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[10] Keyvan Azadbakht, Frank De Boer, and Vlad Nicolae Serbanescu. Multi-threaded actors. In *ICE 2016: Proceedings of the 9th Interaction and Concurrency Experience Satellite Workshop of DisCoTec 2016*, pages 51–66, Heraklion, Greece, June 2016. EPTCS 223.

[11] Anys Bacha and Radu Teodorescu. Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors. In *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 297–307, Tel-Aviv, Israel, June 2013. ACM.

[12] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[13] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *The Journal of Instruction-Level Parallelism*, 10(1):1–26, 2008.

[14] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Elsevier Advances in Computers*, 82(2):47–111, 2011.

[15] Ralph Benko. Happy earth day: How to use capitalism to bring us abundant, cheap and emission-free energy, Forbes. `http://ow.ly/WEjo306BmAv`. Accessed on August 24, 2016.

[16] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Computer Society Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[17] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[18] Gemma Brady, Nikil Kapur, Jonathan Summers, and Harvey Thompson. A case study and critical assessment in calculating Power Usage Effectiveness for a data centre. *Energy Conversion & Management*, 76(1):155–161, 2014.

[19] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS'08: Proceedings of the 2nd International Conference on Distributed Event-Based Systems*, pages 265–275, Rome, Italy, July 2008. IEEE Computer Society.

[20] André R. Brodtkorb, Trond R. Hagen, Christian Schulz, and Geir Hasle. GPU computing in discrete optimization. Part I: Introduction to the GPU. *EURO Journal on Transportation and Logistics*, 2(1):129–157, 2013.

[21] Nathan Bronson. ScalaSTM. `http://nbronson.github.com/scala-stm/`. Accessed on May 24, 2016.

[22] Nathan Bronson, Hassan Chafi, and Kunle Olukotun. CCSTM: A library-based STM for Scala. In *Proceedings of the 1st Annual Scala Workshop at Scala Days*, Lausanne, Switzerland, April 2010.

[23] Chang Cai, Lizhe Wang, Samee Ullah Khan, and Jie Tao. Energy-aware high performance computing: A taxonomy study. In *ICPADS 2011: Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, pages 953–958, Tainan, Taiwan, December 2011. IEEE Computer Society.

[24] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, September 2008. IEEE Computer Society.

[25] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09: Proceedings of The IEEE International Symposium on Workload Characterization*, pages 44–54, Austin, TX, USA, October 2009. IEEE Computer Society.

[26] Hui Chen, Shinan Wang, and W. Shi. Where does the power go in a computer system: Experimental analysis and implications. In *IGCC'11: Proceedings of the 2011 International Green Computing Conference and Workshops*, pages 1–6, Orlando, Florida, USA, July 2011. IEEE Computer Society.

[27] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys'10: Proceedings of the 5th European conference on Computer systems*, pages 27–40, Paris, April 2010. ACM.

[28] William Douglas Clinger. Foundations of actor semantics. Technical report, MIT, 1981.

[29] Gary Cook. How clean is your cloud? (2012). `http://www.greenpeace.org/international/en/publications/Campaign-reports/Climate-Reports/How-Clean-is-Your-Cloud/`. Accessed on May 24, 2016.

[30] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`, 2011. Accessed on May 24, 2016.

[31] Adrian Cristal, Osman Unsal, Gulay Yalcin, Christof Fetzer, Jons-Tobias Wamhoff, Pascal Felber, Derin Harmanci, and Anita Sobe. Combining error detection and transactional memory for energy-efficient computing below safe operation margins. In *TRANSACT'13: 8th ACM SIGPLAN Workshop on Transactional Computing*, pages 1–9, Houston, TX, USA, March 2013. IEEE Computer Society.

[32] Roxana Diaconescu. *Object based concurrency for data parallel applications: Programmability and effectiveness*. PhD thesis, Norwegian University of Science and Technology, 2002. URL `http://www.idi.ntnu.no/grupper/su/publ/phd/roxana-thesis.pdf`.

[33] Diego Didona, Pascal Felber, Derin Harmanci, and Jörg Schenker. Identifying the optimal level of parallelism in transactional memory systems. In *NETYS 2013: Proceedings of the International Conference on Networked Systems*, pages 233–247, Marrakech, Morocco, May 2013. IEEE Computer Society.

[34] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM SIGPLAN Notices*, 44(6):155–165, 2009.

[35] Datacenter Dynamics. Cebit: Cloud&Heat claims pue of 1.05 with waste heat reuse. `http://www.datacenterdynamics.com/content-tracks/servers-storage/cebit-cloudheat-claims-pue-of-105-with-waste-heat-reuse/95880.fullarticle`. Accessed on August 24, 2016.

[36] Oguz Ergin, Osman Unsal, Xavier Vera, and Antonio González. Reducing soft errors through operand width aware policies. *IEEE Transactions on Dependable and Secure Computing*, 6(3):217–230, 2009.

[37] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *ACM SIGARCH Computer Architecture News*, 39(3):365–376, 2011.

[38] Karl-Filip Faxén. Wool-A work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2008.

[39] Vincent W Freeh and David K Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *PPoPP'05: Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pages 164–173, Chicago, IL, USA, June 2005. ACM.

[40] Rong Ge, Xizhou Feng, Martin Burtscher, and Ziliang Zong. PEACH: A model for performance and energy aware cooperative hybrid computing. In *CF'14: Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–24, Cagliari, Italy, May 2014. ACM.

[41] Cloud&Heat Technologies GmbH. Home page. `https://www.cloudandheat.com/`. Accessed on August 24, 2016.

[42] Ruben González, Adrian Cristal, Alex Veidenbaum, Miquel Pericas, and Mateo Valero. An asymmetric clustered processor based on value content. In *ICS'05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 61–70, Cambridge, Massachusetts, June 2005. ACM.

[43] Daniel Goodman, Behram Khan, Salman Khan, Mikel Luján, and Ian Watson. Software transactional memories for Scala. *Journal of Parallel and Distributed Computing*, 73 (2):150–163, 2013.

[44] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS-2011: 2011 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144, Austin, TX, April 2011. IEEE Computer Society.

[45] Irene Greif. Semantics of communicating parallel processes. Technical report, MIT, 1975.

[46] Felipe B. Guardiano and Mohan Srivastava. Multivariate geostatistics: Beyond bivariate moments. *Geostatistics Tróia '92*, 1(5):133–144, 1993.

[47] Munish Gupta. *Akka Essentials*. Packt Publishing, 2012.

[48] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5): 532–533, 1988.

[49] Philipp Haller. On the integration of the actor model in mainstream technologies: The Scala perspective. In *AGERE!'12: Proceedings of the 2nd International Workshop on Programming based on Actors, Agents, and Decentralized Control*, pages 1–6, Tucson, Arizona, USA, October 2012. ACM.

[50] James Hamilton. Cost of power in large-scale data centers. `http://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers`, . Accessed on May 24, 2016.

[51] James Hamilton. PUE and total power usage efficiency. `http://perspectives.mvdirona.com/2009/06/pue-and-total-power-usage-efficiency-tpue/`, . Accessed on May 24, 2016.

[52] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP'05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, USA, June 2005. ACM.

[53] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

[54] Yaroslav Hayduk, Anita Sobe, Derin Harmanci, Patrick Marlier, and Pascal Felber. Speculative concurrent processing with transactional memory in the actor model. In *OPODIS 2013: Proceedings of the 17th International Conference on Principles Of Distributed Systems*, pages 160–175, Nice, France, December 2013. Springer.

[55] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2): 125–141, 1993.

[56] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for dynamic-sized data structures. In *PODC'03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, July 2003. ACM.

[57] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. In *IFIP'77: Proceedings of the International Federation for Information Processing*, pages 987–992, Dallas, Texas, USA, June 1977. North-Holland.

[58] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for Artificial Intelligence. In *JCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, Stanford, USA, 1973. Morgan Kaufmann.

[59] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, Boston, MA, March 2011. USENIX Association.

[60] Urs Hoelzle and Luiz Andre Barroso. *The datacenter as a computer: An introduction to the design of warehouse-scale machines, 2nd edition*. Morgan & Claypool, 2013.

[61] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *ISCA'10: Proceedings of the 37th International Symposium on Computer Architecture*, pages 280–289, Saint-Malo, France, June 2010. ACM.

[62] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. SEEP: Exploiting symbolic execution for energy-aware programming. *ACM SIGOPS Operating Systems Review*, 45(3):58–62, 2011.

[63] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–48, San Diego, CA, USA, June 2003. ACM.

[64] Song Huang, Shucai Xiao, and Wuchun Feng. On the energy efficiency of Graphics Processing Units for scientific computing. In *IPDPS 2009: Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, pages 1–8, Rome, Italy, May 2009. IEEE Computer Society.

[65] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. In *OOPSLA'12: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 753–772, Tucson, Arizona, USA, October 2012. ACM.

[66] Lightbend Inc. The official Akka webpage. `http://akka.io/`. Accessed on May 24, 2016.

[67] James M. Kaplan, William Forrest, and Noah Kindle. Revolutionizing data center energy efficiency. Technical report, McKinsey & Company, 2008.

[68] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A comparative analysis. In *PPPJ'09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, Calgary, Alberta, Canada, August 2009. ACM.

[69] Kiran Kasichayanula, Dan Terpstra, Piotr Luszczek, Stan Tomov, Shirley Moore, and Gregory D. Peterson. Power aware computing on GPUs. In *SAAHPC'12: Proceedings of the 2012 Symposium on Application Accelerators in High-Performance Computing*, pages 64–73, Chicago IL, USA, July 2012. IEEE Computer Society.

[70] Thomas Knauth and Christof Fetzer. DreamServer: Truly on-demand cloud services. In *SYSTOR'14: Proceedings of the 7th ACM SIGOPS International Systems & Storage Conference*, pages 1–11, Haifa, Israel, June 2014. ACM.

[71] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Computer Society Annals of the History of Computing*, 33(3):46–54, 2011.

[72] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[73] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Active Object: An Object Behavioral Pattern for Concurrent Programming*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[74] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *ISCA'10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, Saint-Malo, France, June 2010. ACM.

[75] Dong Li, Bronis R de Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *IPDPS 2010: Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, pages 1–12, Atlanta, GA, USA, April 2010. IEEE Computer Society.

[76] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, New York, December 2009. ACM.

[77] Michael J. Lutz. The Erlang approach to concurrent system development. In *FIE 2013: Proceedings of the 43rd Annual Frontiers in Education Conference*, pages 12–13, Oklahoma City, Oklahoma, USA, October 2013. IEEE Computer Society.

[78] Gregoire Mariethoz, Philippe Renard, and Julien Straubhaar. The Direct Sampling method to perform multiple-point geostatistical simulations. *Water Resources Research, American Geophysical Union*, 46(11):1–14, 2010.

[79] Microsoft. How 16-Bit and 32-Bit programs multitask in Windows 95. `https://support.microsoft.com/en-us/kb/117567/`. Accessed on May 24, 2016.

[80] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. *ACM SIGPLAN Notices*, 41(11):359–370, 2006.

[81] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *IISWC'2006: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 182–188, San Jose, CA, USA, October 2006. IEEE Computer Society.

[82] Michael Nash and Wade Waldron. *Applied Akka Patterns*. O'Reilly Media, 2016.

[83] Cyprien Noël. Extensible software transactional memory. In *C3S2E'10: Proceedings of the 3rd C* Conference on Computer Science and Software Engineering*, pages 23–34, Montreal, Quebec, Canada, May 2010. ACM.

[84] Oscar Palomar, Santhosh Kumar Rethinagiri, Gulay Yalcin, Rubén Titos-Gil, Pablo Prieto, Emma Torrella, Osman Unsal, Adrián Cristal, Pascal Felber, Anita Sobe, Yaroslav Hayduk, Mascha Kurpicz, Christof Fetzer, Thomas Knauth, Malte Schneegaß, Jens Struckmeier, and Dragomir Milojevic. Energy minimization at all layers of the data center: The ParaDIME project. In *DATE 2016: Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, pages 684–689, Dresden, Germany, 2016. IEEE Computer Society.

[85] Preeti Ranjan Panda, B. V. N. Silpa, Aviral Shrivastava, and Krishnaiah Gummidipudi. *Power-efficient System Design*. Springer, 2010.

[86] Pavlos Petoumenos, Lev Mukhanov, Zheng Wang, Hugh Leather, and Dimitrios Nikolopoulos. Power Capping: What works, what does not. In *ICPADS 2015: Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems*, pages 525–534, Melbourne, Australia, December 2015. Springer.

[87] Giuseppe Procaccianti, Stefano Bevini, and Patricia Lago. Energy efficiency in cloud software architectures. In *EnviroInfo 2013: Proceedings of the 27th Conference on Environmental Informatics - Informatics for Environmental Protection, Sustainable Development and Risk Management*, pages 291–299, Hamburg, Germany, October 2013. Shaker Verlag GmbH.

[88] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread Reinforcer: Dynamically determining number of threads via OS level monitoring. In *IISWC-2011: Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, pages 116–125, Austin, TX, USA, November 2011. IEEE Computer Society.

[89] Félix Raimundo. Robots – an Actor system library implemented in pure Rust. `https://gamazeps.github.io/`. Accessed on May 24, 2016.

[90] Santhosh Kumar Rethinagiri, Oscar Palomar, Anita Sobe, Thomas Knauth, Wojciech Barczynski, Gulay Yalcin, Yarco Hayduk, Adrian Cristal, Osman Unsal, Pascal Felber, Christof Fetzer, Julien Ryckaert, and Gina Alioto. ParaDIME: Parallel Distributed Infrastructure for Minimization of Energy. In *DSD 2014: Proceedings of the 17th Euromicro Conference on Digital System Design*, pages 191–198, Verona, Italy, August 2014. IEEE Computer Society.

[91] Santhosh Kumar Rethinagiri, Oscar Palomar, Anita Sobe, Gulay Yalcin, Thomas Knauth, Rubén Titos Gil, Pablo Prieto, Malte Schneegaß, Adrian Cristal, Osman Unsal, Pascal Felber, Christof Fetzer, and Dragomir Milojevic. ParaDIME: Parallel distributed infrastructure for minimization of energy for data centers. *Microprocessors and Microsystems*, 39(8):1174–1189, 2015.

[92] Mahsan Rofouei, Thanos Stathopoulos, Sebi Ryffel, William Kaiser, and Majid Sarrafzadeh. Energy-aware high performance computing with Graphic Processing Units. In *HotPower'08: Proceedings of the 2008 Workshop on Power Aware Computing and Systems*, pages 11–11, San Diego, CA, USA, December 2008. ACM.

[93] Thomas Rouvinez and Anita Sobe. Comparison of Active Objects and the Actor Model. Technical report, IIUN Department, University of Neuchatel, 2014.

[94] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. On ordering transaction commit. In *PPoPP'16: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 46:1–46:2, Barcelona, Spain, March 2016. ACM.

[95] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, San Jose, California, USA, June 2011. ACM.

[96] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini. Towards transactional memory support for GCC. In *GROW09: Proceedings of the 1st International Workshop on GCC Research Opportunities*, pages 1–12, Paphos, Cyprus, January 2009. ACM.

[97] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the Actor Model. In *SBLP 2010: Proceedings of the 14th Brazilian Symposium on Programming Languages*, pages 52–64, Salvador, Brazil, September 2010. Elsevier.

[98] Michael C. Shebanow. Pervasive massively multithreaded GPU processors. In *CF'09: Proceedings of the 6th ACM Conference on Computing Frontiers*, pages 227–227, Ischia, Italy, May 2009. ACM.

[99] Larry Smarr. Project GreenLight: Optimizing cyber-infrastructure for a carbon-constrained world. *IEEE Computer*, 43(1):22–27, 2010.

[100] Burton J. Smith. Parallel computing in science and engineering. In *Shared memory, vectors, message passing, and scalability*, pages 27–34. Springer, 1988.

[101] William Stallings. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Prentice-Hall, Inc., 2005.

[102] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *ICML'11: Proceedings of the 28th International Conference on Machine Learning*, pages 609–616, Bellevue, Washington, USA, June 2011. ACM.

[103] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP 2013: Proceedings of the 27th European Conference on Object-Oriented Programming*, pages 52–78, Montpellier, France, July 2013. Springer.

[104] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *TECS: ACM Transactions on Embedded Computing Systems*, 13(4s):1–25, 2014.

[105] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *ASPLOS'08: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, Seattle, WA, USA, March 2008. ACM.

[106] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):1–7, 2005.

[107] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[108] Janwillem Swalens, Stefan Marr, Joeri De Koster, and Tom Van Cutsem. Towards composable concurrency abstractions. In *EPTCS 155: Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, pages 54–60, Grenoble, France, April 2014. Open Publishing Association.

[109] Hiroyuki Takizawa, Katsuto Sato, and Hiroaki Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. In *Cluster 2008: Proceedings of the IEEE International Conference on Cluster Computing*, pages 386–393, Tsukuba, October 2008. IEEE Computer Society.

[110] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice Hall PTR, 1984.

[111] Haluk Topcuouglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[112] Anne E. Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, 2013.

[113] Abhinav Vishnu, Shuaiwen Song, Andres Marquez, Kevin Barker, Darren Kerbyson, Kirk Cameron, and Pavan Balaji. Designing energy efficient communication runtime systems for data centric programming models. In *GREENCOM-CPSCOM'10: Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, pages 229–236, Hangzhou, China, December 2010. IEEE Computer Society.

[114] Jia Wang, Xiaoping Li, and Jie Yang. Energy-aware task scheduling of MapReduce cluster. In *ICSS 2016: Proceedings of the 8th International Conference on Service Science*, pages 187–194, Weihai, China, May 2015. IEEE Computer Society.

[115] Josh Whitney and Pierre Delforge. Data center efficiency assessment - scaling up energy efficiency across the data center industry: Evaluating key drivers and barriers. Technical report, Natural Resources Defense Council, 2014.

[116] Gulay Yalcin, Anita Sobe, Derin Harmanci, Alexey Voronin, Jons-Tobias Wamhoff, Pascal Felber, Osman Unsal, Adrian Cristal, and Christof Fetzer. Combining error detection and Transactional Memory for energy-efficient computing below safe operation margins. In *PDP 2014: Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 248–255, Torino, Italy, February 2014. IEEE Computer Society.

[117] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Cluster 2010: Proceedings of the IEEE International Conference on Cluster Computing*, pages 19–28, Heraklion, Crete, September 2010. IEEE Computer Society.

[118] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[119] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA'08: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, Munich, Germany, June 2008. ACM.

[120] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud'10: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, Boston, MA, USA, June 2010. USENIX.

[121] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC'07: Proceedings of the 4th International Conference on Autonomic Computing*, pages 27–27, Jacksonville, FL, June 2007. IEEE Computer Society.