

COORDINATING RESOURCE USE IN OPEN
DISTRIBUTED SYSTEMS

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Xinghui Zhao

©Xinghui Zhao, June 2012. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

ABSTRACT

In an open distributed system, computational resources are peer-owned, and distributed over time and space. The system is *open* to interactions with its environment, and the resources can dynamically join or leave the system, or can be discovered at runtime. This dynamicity leads to opportunities to carry out computations without statically owned resources, harnessing the collective compute power of the resources connected by the Internet. However, realizing this potential requires efficient and scalable resource discovery, coordination, and control, which present challenges in a dynamic, open environment.

In this thesis, I present an approach to address these challenges by separating the functionality concerns of concurrent computations from those of coordinating their resource use, with the purpose of reducing programming complexity, and aiding development of correct, efficient, and resource-aware concurrent programs.

As a first step towards effectively coordinating distributed resources, I developed DREAM, a Distributed Resource Estimation and Allocation Model, which enables computations to reason about future availability of resources. I then developed a fine-grained resource coordination scheme for distributed computations. The coordination scheme integrates DREAM-based resource reasoning into a distributed scheduler, for deciding and enforcing fine-grained resource-use schedules for distributed computations. To control the overhead caused by the coordination, a tuner is implemented which explicitly balances the overhead of the control mechanisms against the extent of control exercised.

The effectiveness and performance of the resource coordination approach have been evaluated using three case studies. Experimental results show that the approach can effectively schedule computations for supporting various types of coordination objectives, such as ensuring Quality-of-Service, power-efficient execution, and dynamic load balancing. The overhead caused by the coordination mechanism is relatively modest, and adjustable through the tuner. In addition, the coordination mechanism does not add extra programming complexity to computations.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my supervisor, Professor Nadeem Jamali, for his valuable guidance, advice, and encouragement over the last nine years. Professor Jamali offered tremendous help in my research, and this thesis would never have been completed without his generous intellectual and moral support. More importantly, I am grateful that Professor Jamali taught me how to think as an independent researcher and encouraged me to pursue a career in academia. What I have learned from him will definitely keep benefiting me throughout my academic career.

I would like to thank the members of my advisory committee, consisting of Professor Michael Bradley (Department of Physics and Engineering Physics), Professor Christopher Dutchyn, Professor Derek Eager, Professor Evangelos Milios (Dalshousie University), and Professor Nathaniel Osgood, for their suggestions, comments, and insights from different perspectives. Their expertise and generous advice are critical in improving the quality of my work. I also would like to thank my external examiner, Professor Sathish Gopalakrishnan (University of British Columbia), for his valuable feedback and constructive comments on the work.

Special thanks go to Professor Gul Agha (University of Illinois at Urbana Champaign), for hosting me as a visiting student at his Open Systems Laboratory. During my visit, Professor Agha provided opportunities for me to interact with researchers and students within/outside of the department, discover open problems, and explore interesting research topics in the field of open distributed systems. He also gave valuable feedback on my research, and shared his insightful ideas with me. The two-month visit at the early stage of my Ph.D study has been such a rewarding experience.

Thanks to the College of Graduate Studies and Research, the Department of Computer Science, at the University of Saskatchewan, as well as the Natural Sciences and Engineering Research Council (NSERC), for the financial support they have provided, which allowed me to concentrate on my research. I am also thankful to the Canadian Foundation for Innovation (CFI) for the generous support on the cutting edge hardware facilities, on which I carried out my experiments. Also I would like to thank College of Graduate Studies and Research,

which provided travel grants for a number of my conference trips.

I would like to thank Professor Qigang Gao and Professor Evengelos Milios, for their generous financial support (through NSERC and MITACS) during the time I spent at Dalhousie University. I moved to Halifax because of a family relocation in 2009, and since then their support has allowed me to continue my own research smoothly in Dalhousie, as well as explore more interesting research projects. I enjoyed working with them, and thanks to them for treating me like their own student. I am also thankful to Professor Peter Bodorik, who offered help when I was new to the department.

I am especially grateful that besides my supervisor, a number of professors have offered their generous advice to me in pursuing an academic career. Thanks to Professor Gul Agha, Professor Evangelos Milios, and Professor Nathaniel Osgood, for their encouragement and support throughout my application process. Thanks to Dr. Alex Brodsky, Professor Qigang Gao and Professor Denis Riordan, for their generous support, valuable feedback on my talk, and for sharing their own experiences. Special thanks to Professor Gordon McCalla, for his encouragement and valuable advice.

I am thankful that both the Department of Computer Science at the University of Saskatchewan and Faculty of Computer Science at Dalhousie University offer friendly environments and sufficient resources for their graduate students. Working in both wonderful departments is the experience I will always treasure. I also would like to thank my colleagues in both Agents Lab and MALNIS Lab, for all the meetings, discussions, presentations, and especially for all the fun we had which made the journey of my Ph.D study very much enjoyable.

I would like to express my gratitude to my parents. My father always encourages me to make effort for achieving my goals, and he is not only a very thoughtful and supportive father, but also a good friend of mine. My mother taught me to be an honest and faithful person. I always feel her love and blessing, although she is no longer with us.

Last but not the least, I would like to thank my family. My husband, Dr. Jian Wu, is always willing to offer his unconditional support to me. Our dearest son, Jerod, provides unexpected but also necessary distractions. Over the years, their love and unwavering confidence in me have been the most reliable source of energy through ups and downs.

CONTENTS

Permission to Use	i
Abstract	ii
Acknowledgements	iii
Contents	v
List of Tables	viii
List of Figures	ix
List of Abbreviations	x
1 Introduction	1
1.1 Thesis Statement	2
1.2 Approach	2
1.2.1 Resource Coordination Mechanisms	2
1.2.2 Evaluation	4
1.3 Contributions	5
1.4 Organization	6
2 Related Work	7
2.1 Actor Model	7
2.1.1 Actor Model	8
2.1.2 Implementation of Actor Semantics	9
2.2 Formal Approaches	10
2.2.1 Multi-Agent Planning	10
2.2.2 Formal Logic	12
2.3 Resource Management Models	13
2.4 Language-Based Approaches	15
2.5 Distributed Scheduling	17
2.6 Summary	18
3 Resource Coordination	19
3.1 Distributed Resource Estimation and Allocation Model (DREAM)	19
3.1.1 Representing Resources	19
3.1.2 Representing Computations	23
3.1.3 Formal Definition of DREAM	29
3.2 Resource Coordination Mechanisms	36
3.2.1 Challenges	37

3.2.2	Installing A Resource Control Mechanism	38
3.2.3	Implementation	43
3.3	Summary	47
4	Case Study: QoS Support	48
4.1	Related Work	48
4.2	Approach	51
4.2.1	Deadline Analysis	52
4.2.2	QoS Policy	53
4.3	Experimental Results	55
4.3.1	Threading Benchmark with Deadline Constraints	55
4.3.2	Live Video Conferencing	57
4.4	Summary	60
5	Case Study: Power-Efficient Multicores	61
5.1	Challenges	62
5.2	Related Work	63
5.3	Effect of Frequency Scaling on Energy Consumption	66
5.3.1	Per-Core Frequency	68
5.3.2	Turbo Boost	70
5.4	Reasoning about Multicore Energy Consumption	72
5.4.1	Representing Multicore Resources and Computations	72
5.4.2	DREAM-MCP	73
5.5	Experimental Results	75
5.6	Summary	78
6	Case Study: Dynamic Load Balancing	80
6.1	Introduction	81
6.2	Related Work	82
6.2.1	Dynamic Load Balancing	82
6.2.2	Benchmarks for Dynamic Load-Balancing	84
6.3	Unbalanced Cobwebbed Fruit Tree	85
6.4	Resource-Coordination Framework for Load Balancing	87
6.4.1	Resources and Computations	87
6.4.2	Resource-Coordination Policy for Dynamic Load-Balancing	88
6.5	Benchmark Implementation	92
6.5.1	Tree Generation	92
6.5.2	Traversal	93
6.6	Experimental Results	94
6.6.1	Experimental Design	94
6.6.2	Baseline Approaches Implementation	95
6.6.3	Experimental Results	97
6.7	Summary	101
7	Conclusion and Future Work	103

7.1	Conclusion	103
7.1.1	Resource Coordination Approach	103
7.1.2	Case Studies	104
7.2	Future Directions	106
7.2.1	Multi-Dimension Tuning	106
7.2.2	Smart Space: Spatial-Temporal Coordination	106
7.2.3	Phone Grid	108
7.2.4	Power-Efficient HPC Clusters	108
7.3	Concluding Remarks	109
	References	110
	A Raw Data from Case Study 1	121
	B Raw Data from Case Study 2	123
	C Raw Data from Case Study 3	124

LIST OF TABLES

3.1	Possible Relations Between Two Time Intervals	21
5.1	GNBP: Execution Time at Maximum Frequency	77
A.1	Raw Data from the ThreadRing Experiments	121
A.2	Tuning Events Triggering from the AF_D Experiment	121
A.3	Raw Data from the Live Video Conferencing	122
A.4	Tuning Events Triggering from the AF_D Experiment	122
B.1	Raw Data from the N-Body Experiments	123
B.2	Tuning Events Triggering from the N-Body Experiments	123
C.1	Raw Data from the UCFT Experiments (Performance Vs. Computation Size; TreeSize=100, StdDev=50)	124
C.2	Tuning Events Triggering from the UCFT Experiments	124
C.3	Raw Data from the UCFT Experiments (Performance Vs. Standard Deviation; TreedSize = 100, AveCompSize = 500)	125
C.4	Tuning Events Triggering from the UCFT Experiments	125
C.5	Raw Data from the UCFT Experiments (Performance Vs. Tree Size; Ave- CompSize = 500, StdDev = 100)	126
C.6	Tuning Events Triggering from the UCFT Experiments	126
C.7	Raw Data from the UCFT Experiments (Responding to Urgent Computations; TreeSize = 100, AveCompSize = 500, StdDev = 100)	127

LIST OF FIGURES

2.1	Structure of an Actor	8
3.1	DREAM Semantics	35
3.2	ActorFoundry Scheduler	39
3.3	Integrating DREAM Reasoning into ActorFoundry Scheduler	40
3.4	System Architecture (AF-D Node Instance)	43
4.1	Threadring Benchmark	57
4.2	Multimedia Application: Video Conference	58
4.3	Audio/Video Stream Synchronization for Live Video Conferencing	60
5.1	Saved Energy on Non-uniform Per-Core Frequency Technology	70
5.2	Saved Energy on Turbo Boost Technology	71
5.3	GGBP: Execution Time at Maximum Frequency	77
5.4	GGBP: Energy Saving and Cost	78
6.1	Unbalanced Cobwebbed Fruit Tree	86
6.2	DREAM Reasoner	89
6.3	Performance Vs. Computation Size (TreeSize=100, StdDev=50)	98
6.4	Performance Vs. Standard Deviation (TreeSize = 100, AveCompSize = 500)	99
6.5	Performance Vs. Tree Size (AveCompSize = 500, StdDev = 100)	100
6.6	Responding to Urgent Computations (TreeSize = 100, AveCompSize = 500, StdDev = 100)	101
7.1	Smart Space	107

LIST OF ABBREVIATIONS

AF	ActorFoundry
AF-D	ActorFoundry with DREAM Reasoning
CCS	Calculus of Communicating Systems
CPS	Continuation Passing Style
CSP	Constraint Satisfaction Problem
DCT	Dynamic Concurrency Throttling
DREAM	Distributed Resource Estimation and Allocation Model
DR	DREAM Reasoner
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
FCFS	First-Come First-Served
FIFO	First-In First-Out
GNBP	Gravitational N-Body Problem
HPC	High Performance Computing
JVM	Java Virtual Machine
LSC	Local Synchronization Constrains
MCVR	Multicore Voltage Regulator
QoS	Quality of Service
RFP	Resource Feasibility Problem
RMS	Resource Management and Scheduling
UCFT	Unbalanced Cobwebbed Fruit Tree
UTS	Unbalanced Tree Search

CHAPTER 1

INTRODUCTION

With the growing ubiquity of networked computers, there is an ever increasing potential for executing computations by utilizing distributed peer-owned resources. As a result, new computation paradigms – such as grid and cloud computing – have emerged, where distributed applications can use resources over the Internet. However, accurately reasoning about resource availability on a network of peer-owned resources – necessary for assurance of resource delivery – remains a significant challenge. The challenge is even more pronounced when these computations are executing in open distributed environments, because in such a context, resources can dynamically join or leave the system at any time, or can be discovered at runtime; the same is true for computations with needs for the resources. On the one hand, there is uncertainty inherent in both the evolving requirements of computations and the highly dynamic environment in which they would be executing; on the other hand, it has long been recognized that matching computations against resources has a high computational complexity [25, 67].

Meeting these challenges can be helped by computations' ability to reason about future availability of resources, and their ability to adapt to the dynamic execution environment as well as their own evolving resource requirements. The mechanisms for this adaptation can be both micro and macro. Micro mechanisms are those that a distributed computation can use in order to adapt; macro mechanisms are those used by the wider system for supporting such computations. Key among these mechanisms are methods for observing the changes to adapt to. Micro mechanisms observe at a fine grain the changes with implications for an individual computation. Macro mechanisms observe the state of the execution environment as a whole at a higher level of abstraction; it is characterized by its aggregation of the observation rather than the coarseness of the grain of observation. The goal of this work is to support

this observation and the decision processes it enables.

In this chapter, I first present the thesis statement in Section 1.1, and then describe the approach in Section 1.2. The contributions of the research are presented in Section 1.3, and finally, Section 1.4 outlines the organization of the thesis.

1.1 Thesis Statement

Fine-grained coordination and control for computational resources distributed over time and space can be programmed elegantly and provided efficiently, leading to opportunities including ensuring quality of service, and enhancing power efficiency and dynamic load balancing.

1.2 Approach

The ability to reason about future availability of resources is essential for computations to adapt to the evolving environment. Reasoning about resource-bounded computations has received a significant amount of attention recently [7, 8, 48, 55]; however, the emphasis has been on adapting behaviors of computations as they try to adapt to resource bounds, rather than empowering computations with the reasoning ability to better navigate in the space of resource uncertainty in search of new resources – to seek out new frontiers, in a manner of speaking. One of the objectives of the approach I present in this thesis is to enable computations to have this ability.

1.2.1 Resource Coordination Mechanisms

As a first step towards coordinating resource use for distributed computations, I developed a Distributed Resource Estimation and Allocation Model (DREAM), which can be used for reasoning about feasibility of carrying out deadline-constrained distributed computations in resource bounded open distributed systems. The following question is considered: “Can we know at time T whether a distributed computation C can complete its execution by deadline D ?” By providing a reasoning scheme for accurately predicting resource availability in the

future, DREAM gives a solution to this question, in the form of a sequence of possibly parallel paths of resource utilization that the computation can take to its completion.

DREAM focuses on computational resources – defined in a broader sense to also include communication resources required by distributed computations – the consumption of which enables computations to make progress. In DREAM, computational resources are defined over time and space. Resources are represented using *resource terms*, which contain several key attributes of the specified resources: type, density, time, and location of existence.

Because resources are the focus of this work, computations which seek to use resources are represented in terms of the resources they require. In other words, we will be interested in which resources, when and how much will computations consume, rather than what the computations do while consuming them. Specifically, each computation is represented by its resource requirements. The computation’s requirements for the near future are assumed to be known, with evolving requirements potentially needing a reevaluation. Representing computations in this manner together with the resource availability over time makes it possible to accurately reason about whether the deadline constraint of a distributed computation can be assured. DREAM uses labeled transition rules to describe the process of resource consumption.

Using DREAM as the underlying basis for resource reasoning, I then designed a resource coordination mechanism, and installed it in the latest optimized version of ActorFoundry [77], a Java library which attempts to faithfully implement the Actor model of concurrency [2]. ActorFoundry was a particularly attractive choice also because of its uniquely accessible code. In addition, the careful analysis and comparison carried out by its developers of the optimizations have shown that ActorFoundry is an efficient Java implementation of Actor semantics [77]. In particular, Karmani et al. showed that with the optimizations, ActorFoundry exhibited performance comparable to that of other Java implementations which were less faithful to Actor semantics, and came close to performance of Erlang [15], which is a programming language supporting Actor semantics. The major challenge to integrate the coordination mechanism into ActorFoundry is that its optimizations, which improve overall performance, also make coordination of resources between sub-computations more difficult. In other words, the efficiency comes at the expense of the ability to program a significant

class of computations requiring fine-grained coordination. Instead of arguing against these limiting optimizations, I take the approach of working with them to support fine-grained coordination. Particularly, the optimizing mechanisms are carefully examined for opportunities for fine-grained control, which can be supported efficiently. Furthermore, another challenge is the overhead caused by the coordination mechanism. Finer-grained control always comes with higher overhead, and because of the dynamicity of the system, it is difficult to use analytical approaches to decide the tradeoff in advance. To address this challenge, a tuner is implemented to automatically or manually control the overhead induced by the resource coordination mechanism at runtime. The automatic tuning happens periodically along with computations.

1.2.2 Evaluation

The effectiveness, efficiency, and flexibility of the resource coordination approach have been evaluated using three types of applications, with different requirements for resource coordination.

- QoS Support

The effectiveness of the resource coordination approach for supporting Quality-of-Service (QoS) requirements has been evaluated using a live video conferencing application. The QoS support is achieved by efficiently calculating fine-grained deadlines for sub-computations, and meeting those deadlines by properly allocating resources and enforcing the resource allocation schedules.

- Power-Efficient Multicores

The energy consumption of a processor is typically proportional to the cube of its frequency. In the DREAM model, an important component in the resource representation is the *rate*, which for CPU resources, represents frequency. Therefore, the approach can be easily extended for accommodating energy consumption of computations. Particularly, I adapted the fine-grained resource-coordination mechanism for supporting multicore-processor resource management and energy-consumption analysis, and illus-

trated the effectiveness of the support using a scientific computing application executed on multicores.

- **Dynamic Load Balancing**

Dynamic load balancing is critical in achieving high performance in parallel systems. I adapted the resource coordination approach for supporting dynamic load balancing by reasoning about resource utilization, controlling resources at a fine grain, and minimizing resources that are expiring. I used a customized high-performance computing benchmark application to illustrate that the proposed approach outperforms the traditional dynamic load-balancing approaches, and also provides extra flexibility to promptly respond to computations with high priorities.

Through the above case studies, I illustrate that the resource coordination approach can provide effective support for different coordination purposes. In addition, because the resource coordination mechanisms are implemented separately from the computations, adapting the resource coordination mechanism does not add extra programming complexity to the computations.

1.3 Contributions

Coordinating resource use in open distributed systems is a challenge, especially when computations have timeliness constraints. The research presented in this thesis addresses this challenge, and has the following contributions.

- First, the DREAM model reifies resources and resource requirements of distributed computations in a novel way, and its use for reasoning about future resource availability and accommodation of new computation is demonstrated.
- Second, the fine-grained resource coordination mechanism based on DREAM can provide effective support for resource coordination among distributed computations at a relatively low and controllable overhead.

- Third, the approach is demonstrated to be adaptable for supporting different types of resource coordination, without adding extra programming complexity to computations.

1.4 Organization

The organization of the thesis is as follows. Related work is reviewed in Chapter 2, including resource coordination approaches in several relevant areas: formal approaches, resource management models, language approaches, and distributed scheduling. Then in Chapter 3 I present the resource coordination approach, including the DREAM model and the resource coordination mechanism which integrates DREAM reasoning into ActorFoundry. Chapters 4, 5, and 6 are case studies for evaluating the resource coordination approach, in terms of QoS support, power-efficiency support, and dynamic load-balancing support, respectively. Finally, in Chapter 7, I conclude the thesis and present future directions of this research.

CHAPTER 2

RELATED WORK

Limited resources have always been a major concern in concurrent systems [34]. Coordinating use of the limited resources by distributed computations is a challenge. Over the last few decades, significant work has been done on how to adapt computations to the resource boundedness of the system, on both the theory and application levels. Existing approaches include developing a resource-bounded reasoning mechanism or a formal logic to investigate computations' behaviors within a resource boundary, developing theoretical models, programming languages or APIs to facilitate resource management, and designing resource-aware scheduling schemes to schedule distributed computations on a network of computational resources. In this chapter, I first review the Actor model of concurrency [2], which is used in this research as the underlying model for distributed computations, in Section 2.1. Then I review different types of approaches for resource coordination, in Sections 2.2, 2.3, 2.4 and 2.5. Finally, Section 2.6 summarizes the chapter.

2.1 Actor Model

There are a number of formal models for formalizing concurrent computations in open systems. π -calculus [100] is a calculus for expressing processes with changing structures. π -calculus was extended from the process algebra CCS (Calculus of Communicating Systems) [99]. The Actor model [2] is another model of concurrency for modelling concurrent and asynchronous processes. The Actor model ¹ is used in the approach presented in this thesis for multiple reasons. First, the Actor model offers a natural programming framework

¹In the thesis, I use “Actor” to refer to the model, and “actor” to refer to the computing entity in the model.

for implementing object-oriented distributed systems. Second, the Actor model offers a convenient way for resource control, because its one-thread-per-actor feature makes it possible to control resources for actors by carefully scheduling actor threads. More details about the Actor model are reviewed in the following sections.

2.1.1 Actor Model

Hewitt first used the term “actor” in his early work for PLANNER [59], and he proposed the concept of actors in [60] in 1977. Grief developed an abstract model [56] for actors, and afterwards Clinger developed the semantics for actors [28]. Agha then extended actors to both a programming language [2, 3] and a data abstraction [2] for concurrent open systems.

Actors are autonomous computational entities which communicate with each other using buffered, asynchronous, point-to-point messages. An actor encapsulates a state, a number of methods (which can change the state of the actor), and a thread of control. Actors are distributed over time and space. Each actor has a globally unique mail address, and it maintains a queue of unprocessed messages it has received. Figure 2.1 shows the structure of an actor.

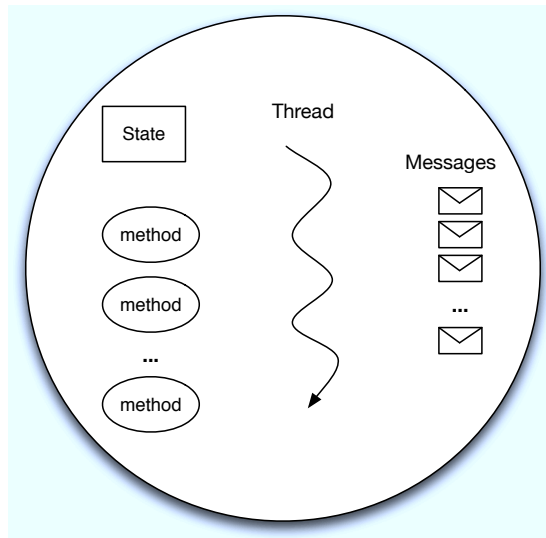


Figure 2.1: Structure of an Actor

The messages in an actor’s message queue are processed one by one according to the order

of arriving. While processing a message, three types of *actor primitives* may occur:

- Create finite number of new actors with some predefined behaviors. The creator actor knows the addresses of the new actors.
- Send messages to other actors. An actor can send a message to another actor only if it knows the name of the destination actor.
- Change the actor's own state and be ready to process the next message.

In addition, the Actor model has the *mobility* property, i.e., an actor can migrate to another location (machine) during its execution, and continue executing there. The Actor model assumes a notion of fairness: the messages that are sent by actors are guaranteed to arrive at the destination actors eventually, but there is no guarantee about the specific order of arriving. Note that another notion of fairness for the Actor model is that no actor can be permanently starved, i.e., the messages in an actor's message queue will eventually be processed.

2.1.2 Implementation of Actor Semantics

The growth of multicore computers has made it imperative for application programmers to write concurrent programs. As a result, the actor-oriented programming receives increasing attention. Some of the actor-based languages include Erlang [15], SALSA [128], E language [126], Axum [98], and Ptolemy [66]. In addition to the novel languages based on the Actor model, there are also actor frameworks/libraries which are developed using existing programming languages, such as C/C++ (Act++ [75], Broadway [123], Thal [79]), and Java (Scala Actor Library [58], Kilim [121], Actor Foundry [16], Actor Architecture [74]). Faithful implementations of the Actor model – such as Actor Foundry [16] and Actor Architecture [74] – implement all properties defined by actor semantics, including encapsulation, fair scheduling, location transparency, and mobility. Unlike these faithful implementations, many actor-oriented frameworks compromise one or more semantic properties of the standard Actor model in order to achieve better performance, such as Scala [58], in which the only actor property being implemented is fair scheduling.

The Actor model encapsulates objects along with threads of execution. Therefore, earlier actor frameworks usually use one-thread-per-actor implementation of actors, such as Scala [58] and Actor Architecture [74]. However, it turns out that in practice, one-thread-per-actor implementation of actors is not particularly efficient, because of the overhead caused by context-switching among actor threads. It is more efficient to have a pool of threads, where each thread processes messages for multiple actors in some order. Karmani et al. implemented this optimization strategy in the latest version of ActorFoundry [77], which has been shown to deliver orders of magnitude better performance than its previous version. Notably, for a Threading benchmark, in which 10 million messages are processed by 503 actors, the optimization improves Actor Foundry’s performance from 695s to 10s, 17% faster than Scala (12s), which achieves good performance by compromising several semantic properties of the Actor model. In fact, the optimized ActorFoundry’s performance comes close to that of Erlang (7s), a programming language which supports Actor semantics.

The optimized ActorFoundry is used in the work presented in this thesis as the underlying framework for supporting distributed computations.

2.2 Formal Approaches

Resource coordination in distributed systems has attracted more and more interest. One class of approaches is *resource reasoning*, which addresses the problem by providing a mechanism for reasoning about resource capacity of a system, and generating plans for resource use. Significant work has been done in this area, including multi-agent planning and formal logic.

2.2.1 Multi-Agent Planning

In the area of multi-agent planning, until the mid-1980s, reasoning systems were built with the axiomatic basis defined by probability theory [29, 64, 112] and utility theory [107], under the assumption that infinite resources were available. However, limited resources may make a complete normative analysis impossible. Horvitz’s work [63] is the first to address the concern about limitations of computational resources in the environment where reasoning systems execute. In [63], inference-related cost is included in the reasoning as a negative

utility value which is against the expected utility, and meta-reasoning is used to compare the values of alternative strategies. Since then, the boundedness of computational resources has received significant amount of attention in reasoning about multi-agent behaviours [48].

A common way to connect reasoning and planning is to build plans using theorem provers, with limited resources considered as bounds on the reasoning. ParcPlan [41] integrates temporal reasoning with resource reasoning, and it aims to solve the resource feasibility problem (RFP) for parallel actions executing in an environment of limited resources. In ParcPlan, resource feasibility is determined by checking the resource capacity constraint at starting points of resource requests, when the number of resources in use increases. TRP [23] is another approach integrating temporal and resource reasoning, by formalizing resource constraints in CSP (constraint satisfaction problem) terms. In TRP, resources are represented by resource profiles, which addresses the temporal attribute of resources. Propagation techniques are provided to synthesize new temporal constraints by reasoning on resource representation. Unlike ParcPlan and TRP, where temporal and resource reasoning is performed after a plan has been obtained, realPlan [122] separates resource reasoning (scheduling) from causal reasoning (planning), leading to improved planning performance. In realPlan, an abstract plan is generated by causal planning without considering resource aspects. The abstract plan is then post-processed for resource allocation to obtain a valid final plan.

Albore et al. proposed a different way to link reasoning and planning [5]. Instead of using theorems to build plans, they show that the problem of reasoning under bounded resources can be recast as a planning problem. In [5], a reasoning agent is modelled as a planning domain where fluents correspond to the set of formulas held in the agent’s memory, and actions correspond to applications of the agent’s inference rules. Different recasting styles are proposed, as well as an approach to evaluate the minimal time and memory bounds for solving the problem.

In most of the existing work in resource-bounded agent reasoning, the emphasis has been on the behaviors of agents/computations constrained by fixed resource bounds, rather than empowering computations with the reasoning ability to better navigate in a world of resource uncertainty. The DREAM model presented in this thesis enables computations to reason about resource availability in the future, so that the feasibility of a computation plan

(whether or not a specific computation can be completed by its deadline) can be determined well in advance.

2.2.2 Formal Logic

Besides the multi-agent planning approaches, in the area of formal logic, step logic [38, 43, 42] is the first to represent resource/time in reasoning systems. In step logic, reasoning is considered to be an on-going process. In other words, each step of the reasoning needs certain amount of time to complete. This requires that the formalism be capable of dealing with time as an object of reasoning. Although step logic eliminates logical omniscience, its semantics have not been formalized. Nirkhe et al. then proposed a possible-worlds type semantics [108] for step logic. However, logical omniscience was reintroduced in this approach.

Timed Reasoning Logics (TRL) [8], which is inspired by step logic, provides a complete and decidable context-logic style formalism to reason about time-bounded reasoners. The semantics of TRL uses syntactic notions but is grounded in the agents computation [134] (e.g., the values of the agent's internal variables or the set of facts in the agent's working memory). The limitation of this approach is that resources are simply represented by a number, and only one type of resource is considered.

Bounded Memory and Communication Logic (BMCL) [7] is the first attempt to integrate multiple resources in one reasoning system, such as time, memory, and communication resources. BMCL can be used to model reasoning agents with bounds on multiple types of resources. However, resources are not explicitly represented in BMCL. Instead, resource bounds are expressed as axioms in the logic, so that during the course of reasoning, the resource bounds can be verified. Therefore, BMCL is not capable of dynamically reasoning about the resource availability in the system. OBA logic [6] is a logic for describing the Observations, Beliefs, intentions and Actions of simple agents. OBA logic explicitly models agent-environment systems, where there are limited resources. Properties of agents specified in OBA can be verified using standard theorem-proving or model-checking techniques. However, the environment in OBA logic is closed and centralized, therefore it can not be used to model open distributed systems.

Unlike the above approaches, the DREAM model presented in this thesis explicitly rep-

resents multiple types of resource in a uniform way using resource terms. In DREAM, the availability of resources throughout the course of computation can be reasoned about.

2.3 Resource Management Models

A significant amount of work has been done in modelling distributed resources, and reifying the control over the resources using a hierarchy of entities. In this section, I review two models which are closely related to the approach presented in this thesis, Quantum [102] and CyberOrgs [69].

Quantum is a theoretical model for resource management. It was first presented in [102], and later extended in [103] in order to handle distributed and multi-type resources. In Quantum, the resource that computations require for their executions is represented by *energy*. The basic resource control unit in Quantum is called a *group*. A group hosts a set of computations, and it also serves as a tank of energy.

In the Quantum model, a group can create new groups, so a hierarchical structure is generated. Each new group is assigned an amount of energy when it is created, and the energy is used for sponsoring the computations in this group. Computations consume energy from the sponsoring group, and if a computation needs more energy than what is available in the group, an energy `exhaustion` primitive is invoked to signal that the current group has run out of energy; if all the computations complete in one group which does not sponsor any sub-groups, the event group `termination` is signalled, and all the remaining energy is returned to the parent group.

Group creation, energy exhaustion, and group termination allow flow of energy between a group and its sub-groups, but energy may also flow between groups independently of the group hierarchy using another two primitives: `pause` and `awake`. `Pause` forces a group and all its sub-groups to be exhausted, and all the energy in this whole hierarchy is transferred to the group which called and sponsored the `pause` operation. Similarly, a group may also transfer energy to an exhausted group in order to make it awake, the group which calls `awake` sponsors the execution of the `awake` primitive.

CyberOrgs [69] is a model for hierarchical coordination of resource usage by multi-agent

applications in a network of peer-owned resources. Each cyberorg² encapsulates a set of computations which are executed concurrently, and an amount of resource. A concurrent computation consumes resource, which is allocated to it by its containing cyberorg. A cyberorg has a contractual relationship with its containing cyberorg, and it may purchase resources from its containing cyberorg according to the signed *contract*. The currency that flows among cyberorgs is called eCash.

CyberOrgs organizes resources and computations as a tree. Each cyberorg except the root cyberorg is contained inside another cyberorg. A cyberorg hosted by another cyberorg purchases resources it needs from the host cyberorg, according to a pre-negotiated contract. This contract, which must be signed between two cyberorgs before one is hosted by the other, stipulates the types and quantities of resources which will be available to the hosted cyberorg as well as their costs. After satisfying its contractual obligations, a cyberorg distributes the remaining resources available to it among the computations it is managing according to its own local resource distribution strategy.

CyberOrgs distribute resources through several primitives.

- **Isolate:**

One cyberorg may create another cyberorg inside it using the `isolate` primitive. A number of actors (computations), messages, and some eCash are encapsulated by the new local client cyberorg. There is a contract between the new cyberorg and its host cyberorg, which is used to determine the trade of resources.

- **Assimilate:**

A local cyberorg can assimilate inside its host cyberorg using the `assimilate` primitive. All the contents of the assimilating cyberorg (actors, eCash, and messages) become contents of the host cyberorg after the assimilation. Furthermore, the contract between the assimilating cyberorg and its host ceases to exist.

- **Migrate:**

A cyberorg may realize that its resource requirement has exceeded what is offered by

²In the thesis, I use “CyberOrgs” to refer to the model, and “cyberorg” to refer to the entities in the model.

its contract with the host cyberorg. This triggers its attempts to migrate. A cyberorg may migrate from one host cyberorg to another. However, this must be preceded by negotiation of the terms under which the client may be hosted.

It has been shown in [72] that the hierarchical control of CyberOrgs can be implemented efficiently. The DREAM model presented in this thesis was inspired by CyberOrgs model. However, the focus of CyberOrgs is to model resource ownership and control, but the focus of DREAM is to model resources control at a fine grain.

2.4 Language-Based Approaches

Ether [83] was one of the earliest languages to address resource allocation among concurrent components. In Ether, every process needs a *sponsor* that is assigned to it to support computations. Later on, ACORE [95], a concurrent programming language based on the actor model, incorporated the idea of Ether. There are *sponsor actors* in ACORE. The sponsor actors can process requests, and *ticks* are required by the process. A similar idea was used in Telescript [133], in which the computational resources are abstracted as *teleclicks*, and processes need teleclicks to accomplish computations.

Java [53] is a language which supports distributed applications by addressing portability, but Java does not provide adequate support for resource management. Many approaches towards resource management try to address Java's deficiency, such as JRes [32], JSeal2 [19], and Java Resource Management API [31].

JRes [32] provides an interface for accounting and limiting access to different types of resources, such as CPU time, network bandwidth, and main memory. JRes was implemented using both Java bytecode editing and native code to account for resources without changing the Java Virtual Machine [94]. The unit for resource control in JRes is the individual thread.

JSeal2 [19] also focuses on resource accounting like JRes, and the API of JSeal2 is similar with JRes too. The developers of JSeal2 were influenced by research on *resource bounded actors* [71], which was the early work of CyberOrgs [69]. In JSeal2, the basic unit of resource management is a *seal*, instead of an individual thread. A seal may be either a mobile object or a service component, and each seal executes in a protected domain and shares no state

with other seals. JSeal2 supports complete portability, because the bytecode transformation technique is used for both CPU time and memory resource controlling, instead of modifying Java run time systems. Before being loaded by the JVM, bytecode is modified in order to account for resources. For the memory resource, before every memory allocation instruction, code for accounting is inserted. CPU time accounting in JSeal2 is based on measuring the number of executed bytecode instructions, so the code for CPU accounting is inserted to every basic block of code.

Java Resource Management API [31] was proposed to be a widely-applicable resource management interface for Java platform. It was recently developed in collaboration with JSeal2's developers, in order to extend resource management support in Java. The unit of resource management in Java Resource Management API is an *Isolate*, which is an encapsulation of a Java program. Isolates do not share state with each other. Resources in the RM API are represented by a set of resource attributes. A *dispenser* isolate is responsible for monitoring available resources and it serves as the connection between the resource implementation and the RM API. Resource consuming policies are encapsulated by *resource domains*, which may specify the reservations of resources and actions that should be executed upon certain events. Java RM API is an extension to Java, so the code is portable across Java implementations. It is different from JSeal2 and JRes, which modify the Java VM.

Although language based approaches can facilitate resource management, the code of computations and that of resource control often mix together, which increases complexity in programming and debugging. The resource coordination approach presented in this thesis clearly separates the concern of resource coordination from that of computations, and the separation of concerns enhances code modularity, readability, and reusability. The current implementation is Java library based, but it is also possible to be implemented as a programming language, which provides syntax and semantics for programming both computations and resource coordination.

2.5 Distributed Scheduling

Work on resource-aware scheduling for distributed computations can be classified into two categories according to their resource sharing patterns: space sharing, in which processors are allocated to different computations, with computations enjoying exclusive use of processors; and time sharing, in which multiple computations may share the same processor through time slices.

A commonly-used space-sharing scheduling is the traditional *first-come first-served* (FCFS) approach, in which resources are allocated to jobs according to their arrival order. FCFS scheduling is fair and predictable. However, the disadvantage of FCFS scheduling is that large jobs often lead to segmentation of resources, resulting in inefficient resource utilization. Studies show that system utilization is typically in the range of 50%-80% [45, 65, 87, 97] for FCFS scheduling. To address this problem, Lifka et al. developed an EASY (the Extensible Argonne Scheduling sYstem) scheduler [92] which uses an aggressive backfilling: small jobs are allowed to move ahead to fill the idle resources, as long as they do not delay the first job in the queue. EASY scheduler may result in unbounded queueing delays for other queued jobs. This problem is solved in [132], in which backfilling is allowed only if no job in the queue is delayed. It has been shown that this alternative approach has the same benefit as the aggressive backfilling in EASY scheduler, with the extra advantage of predictable queueing times. A variation of this approach – *resource-usage aware backfilling* [57] – is highly relevant to the proposed work: resource usage is tracked using a reservation table to avoid resource saturation. Specifically, instead of backfilling the first job that can be moved to the run queue based on the job arrival time or job size, the local scheduler cooperates with the local resource manager to look ahead to all queued jobs, in order to find out the allocation that minimizes the job runtime penalty due to the saturation of the resource sharing.

In time-sharing approaches, a well known approach is gang scheduling [46], in which the scheduling policy coordinates context switching across multiple processors, so that a number of interacting threads can be scheduled simultaneously on their time slices. Comparing to space sharing approaches, gang scheduling has been shown to be able to provide improved overall system utilization and responsiveness [47]. Recent work has developed variations of

gang scheduling, which reduce memory contention [44], and adapt to multicore architectures [135].

Most of the resource allocation problems, including job scheduling, fall into applicable areas of stochastic discrete optimization. Different optimization techniques [118, 52, 141] can be used to address these problems. However, the search for optimal solutions normally leads to NP-hard problems. For example, the existing job scheduling approaches aim to improve overall performance, i.e., minimize the total execution time (makespan), or shorten waiting times for computations. The overhead caused by exploring alternatives could be high. The focus of the work presented in this thesis is to adapt computations to a dynamic execution environment, and achieve required control with acceptable overhead.

2.6 Summary

In this chapter, different types of research work related to resource coordination has been reviewed, including formal approaches, resource management models, programming language approaches, and distributed scheduling. In addition, I have also reviewed the Actor model, a model of concurrency which is used in the approach presented in this thesis as the underlying computation model, as well as various implementations of actor semantics.

CHAPTER 3

RESOURCE COORDINATION

In this chapter, I first present DREAM in Section 3.1, which is a Distributed Resource Estimation and Allocation Model for reasoning about feasibility of carrying out deadline-constrained distributed computations in resource bounded open distributed systems. Then I present a resource coordination approach in Section 3.2, which integrates DREAM reasoning into ActorFoundry, an optimized implementation of Actors, for providing fine-grained resource control. Section 3.3 gives a summary of the chapter.

3.1 Distributed Resource Estimation and Allocation Model (DREAM)

In DREAM, computational resources are defined over time and space, and represented using *resource terms*, which specify key attributes of resources. Computations are represented in terms of resources they require. Syntax and semantics of DREAM are described. Theorems are derived to illustrate how the logic can be used to express resource properties of the system in time and space, track resource utilization, and calculate future availability of resources. Particularly, at any time, given a computation, it is possible to evaluate whether its deadline constraint is assured by the available resources.

3.1.1 Representing Resources

Distributed computations execute in environments where computational and communication resources are spread over time and space. In an approach inspired by the CyberOrgs model for resource bounded concurrent systems [69], I define resources in time and space. Note that

only *consumable resources* are modelled by DREAM. For example, system resources, such as CPU cycles and network bandwidth, are modelled; similarly, application resources, such as user attention and time of effort, can be modelled. Cloneable resources, such as algorithms and data/results, are not modelled. Memory resource is a special type of resource, because it is usually required for maintaining a computation’s state. Although memory resources over time and space, i.e., cacheing and paging, can be modelled by DREAM, a more detailed analysis is left for future exploration. Therefore, in this work, we assume memory resources are available as needed, and are not considered.

In DREAM, each computational resource is represented by a *resource term*: $\llbracket \mathbf{r} \rrbracket_{\xi}^{\tau}$, where:

- \mathbf{r} represents the rate of availability of the resource, in *quantity/time*
- τ is the time interval during which the resource exists ¹
- ξ denotes the *located type* of the specified resource

The located type contains both the type of the resource and the location where the resource is residing. For example, for “CPU resource on location l_1 ,” the located type is $\langle \text{cpu}, l_1 \rangle$. In comparison, the spatial information for a network resource has to identify both the source and destination nodes of the resource. For example, the located type of a network resource that can be used to send data from location l_1 to l_2 would be specified as $\langle \text{network}, l_1 \rightarrow l_2 \rangle$. Note that for a multicore processor, the located type can be either coarse-grained, i.e., only specifies the processor location, or fine-grained, i.e., specifies both processor location and core id, according to the preferred granularity of control.

Because each resource term is associated with a time interval τ , relationships between time intervals must be defined before we can discuss the operations on resource terms. In DREAM, I use Interval Algebra [9] to formalize relations between two time intervals. As shown in Table 3.1, the seven possible relations (or thirteen if we count the inverse relations) are as follows.

- before ($<$): $\tau_1 < \tau_2$ means τ_1 ends before τ_2 starts

¹The product $\mathbf{r} \times \tau$ gives the total quantity of the available resource over the course of time interval τ

- equal (=): $\tau_1 = \tau_2$ means τ_1 and τ_2 represent the same time period
- during (*d*): $\tau_1 d \tau_2$ means τ_1 starts after τ_2 starts, and ends before τ_2 ends
- meets (*m*): $\tau_1 m \tau_2$ means τ_2 starts immediately after τ_1 ends
- overlaps (*o*): $\tau_1 o \tau_2$ means τ_1 starts before τ_2 starts, and ends before τ_2 ends
- starts (*s*): $\tau_1 s \tau_2$ means τ_1 and τ_2 start at the same time point
- finishes (*f*): $\tau_1 f \tau_2$ means τ_1 and τ_2 end at the same time point

Table 3.1: Possible Relations Between Two Time Intervals

Relation	Inverse relation	Interpretation	Illustration
$\tau_1 < \tau_2$	$\tau_2 > \tau_1$	τ_1 before τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2$
$\tau_1 m \tau_2$	$\tau_2 mi \tau_1$	τ_1 meets τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2$
$\tau_1 = \tau_2$	$\tau_2 = \tau_1$	τ_1 equal τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2$
$\tau_1 d \tau_2$	$\tau_2 di \tau_1$	τ_1 during τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2 \tau_2 \tau_2 \tau_2$
$\tau_1 o \tau_2$	$\tau_2 oi \tau_1$	τ_1 overlaps τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2$
$\tau_1 s \tau_2$	$\tau_2 si \tau_1$	τ_1 starts τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2 \tau_2 \tau_2 \tau_2$
$\tau_1 f \tau_2$	$\tau_2 fi \tau_1$	τ_1 finishes τ_2	$\tau_1 \tau_1 \tau_1$ $\tau_2 \tau_2 \tau_2 \tau_2 \tau_2 \tau_2$

Each time interval τ has a start time t_{start} , and an end time t_{end} ($t_{start} < t_{end}$). I also use (t_{start}, t_{end}) as an alternative notation for time interval τ . Furthermore, binary operations on

sets, such as union (\cup), intersection (\cap), relative complementation (\setminus) are also available for time intervals.

Resources in a distributed system can be represented by a set of resource terms, each with its own located type. Resources joining or leaving the system can then be expressed by union and relative complementation operations on resource sets, respectively.

If two resource terms in a resource set have the same located type and overlapping time intervals, they can be combined by a process of simplification, where for any interval for which they overlap, their rates are added, and for remaining intervals, they are represented separately in the set:

$$\{\llbracket \mathbf{r}_1 \rrbracket_{\xi}^{\tau_1}\} \cup \{\llbracket \mathbf{r}_2 \rrbracket_{\xi}^{\tau_2}\} = \{\llbracket \mathbf{r}_1 \rrbracket_{\xi}^{\tau_1 \setminus \tau_2}, \llbracket \mathbf{r}_1 + \mathbf{r}_2 \rrbracket_{\xi}^{\tau_1 \cap \tau_2}, \llbracket \mathbf{r}_2 \rrbracket_{\xi}^{\tau_2 \setminus \tau_1}\}$$

The simplification essentially aggregates resources available simultaneously which have identical located type. Note that this simplification can lead to a larger number of terms. Resource terms can reduce in number if two identical located type resources with identical rates have time intervals that meet.

Note that if the time interval of a resource term is empty, the value of the resource term is \emptyset , or null. In other words, resources are only defined during non-empty time intervals.

The notion of negative resource terms is not meaningful in this context, therefore resource terms cannot be negative. I define an inequality operator to compare two resource terms, from the perspective of a computation's potential use of them. We say that a resource term is greater than another if a computation that requires the latter, can instead use the former, with some to spare. It can be specifically stated as follows: $\llbracket \mathbf{r}_1 \rrbracket_{\xi_1}^{\tau_1} > \llbracket \mathbf{r}_2 \rrbracket_{\xi_2}^{\tau_2}$ if and only if $\xi_1 = \xi_2$, $\mathbf{r}_1 > \mathbf{r}_2$, and one of the following is true: $\tau_2 \text{ } d \text{ } \tau_1$, $\tau_2 \text{ } s \text{ } \tau_1$, and $\tau_2 \text{ } f \text{ } \tau_1$. Note that it is not necessarily enough for the total amount of resource available over the course of an interval to be greater. Consider a computation that is able to utilize needed resources only during interval τ_2 , if additional resources are available outside of τ_2 , but not enough during τ_2 , it does not help satisfy the computation.

The relative complementation of two resource sets $\Theta_1 \setminus \Theta_2$ is defined only when for each resource term $\llbracket \mathbf{r}_2 \rrbracket_{\xi}^{\tau_2}$ in Θ_2 , there exists a resource term $\llbracket \mathbf{r}_1 \rrbracket_{\xi}^{\tau_1} \in \Theta_1$, such that $\llbracket \mathbf{r}_1 \rrbracket_{\xi}^{\tau_1} > \llbracket \mathbf{r}_2 \rrbracket_{\xi}^{\tau_2}$. The relative complementation of two resource sets is defined as follows:

$$\{\Theta_1, \llbracket \mathbf{r}_1 \rrbracket_\xi^{\tau_1}\} \setminus \{\Theta_2, \llbracket \mathbf{r}_2 \rrbracket_\xi^{\tau_2}\} = \{\llbracket \mathbf{r}_1 \rrbracket_\xi^{\tau_1} - \llbracket \mathbf{r}_2 \rrbracket_\xi^{\tau_2}\} \cup \Theta_1 \setminus \Theta_2$$

where $\{\llbracket \mathbf{r}_1 \rrbracket_\xi^{\tau_1} - \llbracket \mathbf{r}_2 \rrbracket_\xi^{\tau_2}\} = \{\llbracket \mathbf{r}_1 \rrbracket_\xi^{\tau_1 \setminus \tau_2}, \llbracket \mathbf{r}_1 - \mathbf{r}_2 \rrbracket_\xi^{\tau_2}\}$.

Following are some examples of calculations on resource sets.

$$\{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,3)}\} \cup \{\llbracket 5 \rrbracket_{\langle \text{network}, l_1 \rightarrow l_2 \rangle}^{(0,5)}\} = \{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,3)}, \llbracket 5 \rrbracket_{\langle \text{network}, l_1 \rightarrow l_2 \rangle}^{(0,5)}\}$$

$$\{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,3)}\} \cup \{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,5)}\} = \{\llbracket 10 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,3)}, \llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(3,5)}\}$$

$$\{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,3)}\} \setminus \{\llbracket 3 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(1,2)}\} = \{\llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(0,1)}, \llbracket 2 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(1,2)}, \llbracket 5 \rrbracket_{\langle \text{cpu}, l_1 \rangle}^{(2,3)}\}$$

Union and relative complementation operations on resource sets allow modelling of resources that join or leave the system dynamically, as typically happens in open distributed systems such as the Internet.

3.1.2 Representing Computations

A computation consumes resources at every step of its execution. I abstract away what a distributed computation does and represent it by the resource requirements for each step of its execution; this idea is inspired by the CyberOrgs model [69]. In general, a computation can be divided into a number of *segments*, and each segment requires the same type(s) of resource(s). The resource requirement of each segment can be represented by the total amount(s) of the required resource(s). If the computation has performance requirements, i.e., timeliness constraints such as earliest start time and deadline, the resource requirements for all segments must be satisfied within the specified time period. In the following sections, I use actor computations as an example to illustrate in detail the representation of computation in DREAM.

Actor Computations

We think of distributed computations as computations carried out by actors [2], which are autonomous concurrently executing active objects which communicate with each other using buffered, asynchronous, point-to-point messages. Actors have globally unique names, and maintain queues of unprocessed messages they have received, which are processed in the order of their arrival. Actors carry out the computations specified by their behaviors (i.e., methods) in the course of processing messages. An actor may **evaluate** expressions, **send** messages to other actors, **create** a finite number of new actors with some predefined behaviors, or change its own state and become **ready** to process the next message. In addition, in a distributed execution environment, an actor may use another primitive **migrate** in order to migrate to another location, and continue executing there. In other words, an actor's behavior is a sequence of these five types of actions. An actor utilizes processor and network resources in order to carry out these actions.

Consider a function Φ , which when provided as parameters an actor's uniquely identifying name, and the computation it is to perform, returns a set of resource amounts representing the required resources for completing the computation.² For example, resources required for actor a_1 – located at $l(a_1)$ ³ – to send a message m to actor a_2 – located at $l(a_2)$ – is $\Phi(a_1, \text{send}(a_2, m))$. The value of a required resource is represented by $\{[\mathbf{q}]_\xi\}$, where \mathbf{q} is the quantity of resource required, and ξ is the located type. For our example, ξ would be $\langle \text{network}, l(a_1) \rightarrow l(a_2) \rangle$. Natural numbers can be used for representing the quantity \mathbf{q} . If actor a_1 needs 4 units of network resource in order to send message m to actor a_2 , then we say:

$$\Phi(a_1, \text{send}(a_2, m)) = \{[4]_{\langle \text{network}, l(a_1) \rightarrow l(a_2) \rangle}\}$$

Similarly, other actions of actor a_1 , can be converted to resource amounts as well, using function Φ as follows:

²This device, although useful for simplifying our discussion, does not imply need for existence of such a function. Any-time algorithms and approximate algorithms are examples of when it is meaningful to talk about such a function. In general, at the cost of some inefficiency, estimates could be used and revised as necessary.

³ l is the location function; $l(a)$ gives the location of actor a

$$\Phi(a_1, \text{evaluate}(e)) = \{[8]_{\langle \text{cpu}, l(a_1) \rangle}\}$$

$$\Phi(a_1, \text{create}(b)) = \{[5]_{\langle \text{cpu}, l(a_1) \rangle}\}$$

$$\Phi(a_1, \text{ready}(b)) = \{[1]_{\langle \text{cpu}, l(a_1) \rangle}\}$$

$$\Phi(a_1, \text{migrate}(l_2)) = \{[3]_{\langle \text{cpu}, l(a_1) \rangle}, [6]_{\langle \text{network}, l(a_1) \rightarrow l_2 \rangle}, [3]_{\langle \text{cpu}, l_2 \rangle}\}$$

The quantities 4, 8, 5, 1, in the above equations are hypothetical amounts used for illustration purposes. Note that a single actor action may require multiple types of resources. For instance, the `migrate` operation needs both CPU and network resources, because in order to be able to resume remotely in its current state, the migrating actor needs first to be serialized, then sent to the destination node over the network, and finally unserialized at the destination node to resume execution.

An alternative to the function Φ is to perform code analysis to find out the meaning of the code and the resource needs. However, when if statements, loops, dynamic storage and recursive data structures present, it is challenging to statically analyze the code. Simplifying assumptions and approximate solutions are often used [88]. In the work presented in this thesis, we assume that the function Φ is provided by the programmer/user, in the form of a look-up table. This is achievable in scenarios such as the following. Programmers may sometimes be able to provide resource requirements. Resource requirements of computations may be obvious, as is the case for large classes of computations in scientific computing. Programmers or the runtime system could begin with an initial estimate of the resource requirements, and update the estimate as necessary.

If fine-grained resource coordination is needed, the programmer/user would provide the resource requirement information of their computations. This is similar to proof-carrying code [89], where programmers are required to provide proof of the safety of their code along with the code, which can then be verified at the remote host.

In DREAM, a sequence of these resource requirements is used to refer to an actor. Specifically, the computation to be carried out by an actor a is represented by a sequence of resource

amounts, with each amount identifying resources required by a particular action.

Furthermore, an individual actor's computation is sequential, which means that actions must be taken in a specific order. Consequently, an action may not be available for execution unless all previous actions have been completed.

Here, the notion of *possible action* is defined as follows.

Definition 1: Possible Action *An actor action γ is a possible action at time t if and only if one of the following is true:*

- γ is the first action of the actor,
- at time t , all actions which precede γ in the sequence of the actor's actions have already been completed.

The necessary and sufficient condition for an actor action to be completed can now be stated as follows:

Axiom 1: At time t , an action γ of actor a , can be carried out if and only if at time t it is a possible action of a , and its required resources $\Phi(a, \gamma)$ are available.

This axiom serves as the foundation for the reasoning made possible by DREAM.

Resource Requirements of Distributed Computations

In DREAM, a distributed computation is represented by a triple (Λ, s, d) , where Λ is a representation of the computation, s is the earliest start time of the computation, and d is the deadline by which the computation must complete. Particularly, the computation does not seek to begin before s and seeks to be completed before d . For this to happen, Λ requires the resources for completing this computation – however distributed over the course of the interval – during the interval.

If there are multiple (possibly concurrent) actor computations in Λ , I use Γ to denote an actor computation, and use γ to denote a single action of the actor (possibly carried out concurrently with other actors carrying out other actions).

The resource requirements of an actor's action are represented as a *simple resource requirement* ρ defined as follows:

$$\rho(\gamma, s, d) = [\Phi(a, \gamma)]^{(s,d)}$$

This simple resource requirement specifies the total amount of resource required for actor a 's action γ during the time interval (s, d) . For convenience, I define a function f , which takes as parameters a resource set Θ and a simple resource requirement $\rho(\gamma, s, d)$, and returns a boolean value *true* or *false*, indicating whether or not the simple resource requirement can be satisfied given the available resource set Θ :

$$f(\Theta, \rho(\gamma, s, d)) = \forall \xi, \sum_i (\mathbf{r}_i \times \tau_i) \geq \Phi_\xi(a, \gamma)$$

where $\Phi_\xi(a, \gamma)$ is the requirement for resource ξ , and $[\mathbf{r}_i]_\xi^{\tau_i}$ is a resource term in Θ , such that $\tau_i \subseteq (s, d)$.⁴

The following theorem states whether an actor action can be completed by its deadline.

Theorem 1: Single Action Accommodation A computation (γ, s, d) which only contains a single actor action γ can be accommodated by a system, if and only if by time s , γ is a possible action, and the system satisfies the simple resource requirement $\rho(\gamma, s, d)$: $f(\Theta, \rho(\gamma, s, d)) = \text{true}$, where Θ is the available resources of the system.

Proof. If $f(\Theta, \rho(\gamma, s, d)) = \text{true}$, the resources required for the computation $\Phi(a, \gamma)$ are available during the time interval (s, d) . In addition, γ is a possible action because it is the first action of actor a (the only action). Therefore, according to Axiom 1, the computation can be completed during (s, d) . This proves “*if.*”

If the computation (γ, s, d) can be completed, according to Axiom 1, there must be enough resource for the execution of γ , meaning $f(\Theta, \rho(\gamma, s, d)) = \text{true}$. This proves “*only if.*” \square

An actor's resource requirements – represented by $\rho(\Gamma, s, d)$ – are for executing a sequence of actions, which may require different types of resources. Critically, the resources needed for completing an actor's execution are required in a specific order. It is not sufficient to simply have the correct total quantities of the resources during the entire interval; the right resources are required at the right time.

⁴Note for a resource term which has a time interval overlapping with (s, d) , we need to break the term into two, and only include the one with time interval during (s, d) . Similarly, for a resource term with a time interval which includes (s, d) , we break the term to three and only choose the one with interval (s, d) .

A *complex resource requirement* is defined in terms of simple resource requirements, to represent the requirements of an actor computation Γ , as follows:

$$\begin{aligned} \rho(\Gamma, s, d) & ::= \rho(\Gamma_1, s, t_1) \cup \rho(\Gamma_2, t_1, t_2) \cup \dots \cup \rho(\Gamma_m, t_{m-1}, d) \\ & \text{where } s < t_1 < t_2 < \dots < t_{m-1} < d \\ & \mid \rho(\gamma, s, d) \end{aligned}$$

As shown in the above equation, function ρ breaks down the actor's computation Γ into a sequence of m subcomputations. As a result, the resource requirements of the actor (Γ, s, d) amount to a sequence of *simple resource requirements* for the subcomputations.⁵ Note that a sequence of actions which require the same single type of resource need not be broken down into multiple subcomputations, because this case is similar to a single actor action, where having enough amount of resource during the interval will guarantee completion of the computation.

Theorem 2 states whether a sequential actor computation can be completed by its deadline.

Theorem 2: Sequential Computation Accommodation A system with resources Θ can accommodate a sequential computation (Γ, s, d) if and only if there exist time points t_1, t_2, \dots, t_{m-1} between s and d , which divide the time interval (s, d) into a sequence of m subintervals, so that the system can satisfy the simple resource requirements for each subinterval.

Proof. Assume the computation (Γ, s, d) can be accommodated by the system. We can set up break points b_1, b_2, \dots, b_m in the computation, each of which identifies the starting point of a subcomputation. In the real-time execution of the computation, the time points when those break points are encountered will be a set of time points: t_1, t_2, \dots, t_{m-1} . These time points divide (s, d) into m subintervals, each of which satisfies the simple resource requirement for the corresponding subcomputation according to Axiom 1. This proves “*only if*.”

Assume we already have a set of time points t_1, t_2, \dots, t_{m-1} , for which the sequence of simple resource requirements are satisfied, according to the definition of complex resource

⁵The intermediate time points in a complex resource requirement, t_1, t_2, \dots, t_{m-1} , are not predetermined. They are determined as a result of the reasoning. They are intended to be descriptive rather than prescriptive.

requirement $\rho(\Gamma, s, d)$ and Axiom 1, it is obvious that the computation can be accommodated by executing each subcomputation during its corresponding time interval, during which the required resources are available. This proves “*if.*” \square

A concurrent computation involves multiple actors. Here, we limit ourselves to concurrent computations involving *independent* actors. In other words, all actors participating in the computation are deemed to be created *en masse* at the beginning of the computation and actors never have to wait for messages from other actors.

Resource requirements of a concurrent computation (Λ, s, d) , can be satisfied by satisfying the *complex resource requirements* (defined previously) of the individual actors, as follows:

$$\rho(\Lambda, s, d) ::= \rho(\Gamma_{a_1}, s, d) \cup \rho(\Gamma_{a_2}, s, d) \cup \dots \cup \rho(\Gamma_{a_n}, s, d)$$

where $\Gamma_{a_1}, \Gamma_{a_2}, \dots, \Gamma_{a_n}$ represent computations carried out by actors a_1, a_2, \dots, a_n respectively. To simplify the model, we assume that actors do not migrate for acquiring resources. In other words, they only migrate for functional reasons. Therefore, the located type of their required resources can be easily determined.

As shown in the above definition, multiple complex resource requirements overlap on the same time interval. In order to determine whether the computation can be accommodated using available resources, we need to find an answer to the following question: “Can the system accommodate one more actor computation (Γ_{a_i}, s, d) when it has already made commitments to accommodate computations $\rho(\Gamma_{a_1}, s, d), \rho(\Gamma_{a_2}, s, d), \dots, \rho(\Gamma_{a_{i-1}}, s, d)$?” If we can answer this question, the problem can be solved step by step, by trying to accommodate one more computation at a time. However, without a clear way of reasoning about resource consumption in the system, it is not possible to answer the question. Next, I introduce DREAM, which provides a framework for performing such reasoning.

3.1.3 Formal Definition of DREAM

With resource terms/sets, DREAM is capable of describing the evolution of a distributed system by reasoning about computational resources. Furthermore, important resource-related

properties can be expressed using DREAM, and propositions about deadline assurance can be verified. A formal definition of DREAM follows.

System Model

The DREAM system model can be represented by a 4-tuple, $\mathbb{M} = (\mathcal{A}, \mathcal{R}, \mathcal{C}, \Phi)$, \mathcal{A} is a set of actor names; \mathcal{R} is a set of resource terms; \mathcal{C} is a set of distributed computations, represented by sequences of actions taken by actors; Φ is a function which maps computations carried out by actors to the resources they require. I define \mathcal{S} , the state of the system as follows:

$$\mathcal{S} = (\Theta, \rho, t)$$

where Θ is a set of resource terms, representing future available resources in the system, starting from time t ; ρ represents the resource requirements of the computations that are accommodated by the system at time t ; and t is the point in time when the system's state is \mathcal{S} .

Progress of the system is triggered by the injection of resources. Resources specified in resource terms expire if there is no computation which requires those resources during the time intervals. This means the resources are only defined for a certain period of time, specified by the time interval in their resource terms.

If the evolution of a DREAM system is denoted by a sequence of states $(\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n)$, the progress of the system is regulated by a labeled transition rule:

$$\mathcal{S}_i \xrightarrow{\xi \rightarrow a} \mathcal{S}_{i+1}$$

where ξ is a resource located type, and a is the name of an actor. The transition rule specifies that the utilization of resource ξ for actor a 's next action makes the system progress from state \mathcal{S}_i to the next state \mathcal{S}_{i+1} . If we replace the states in the above transition rule with the detailed (Θ, ρ, t) format, the transition rule can be written as:

$$(\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}, \Theta\}, \{\llbracket \mathbf{q} \rrbracket_{\xi}^{(t,t'')}, \rho\}, t) \xrightarrow{\xi \rightarrow a} (\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t+\Delta t, t')}, \Theta\}, \{\llbracket \mathbf{q} - \mathbf{r} \times \Delta t \rrbracket_{\xi}^{(t+\Delta t, t'')}, \rho\}, t + \Delta t)$$

where $\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}$ is the available resource of located type ξ , $\llbracket \mathbf{q} \rrbracket_{\xi}^{(t,t')}$ is the simple resource requirement of actor a 's action associated with resource ξ , and Δt is the smallest time slice

the system can account. Every time a transition rule is applied to the system, the system progresses one step further by time Δt .⁶ Here, the transition rule states that during the time interval $(t, t + \Delta t)$, the available resource ξ is used to fuel actor a 's action. As a result, by time $t + \Delta t$, actor a 's requirement for ξ will be $\mathbf{r} \times \Delta t$ less than it was at time t .

The above transition rule is the **sequential transition rule**, because only one actor in the system obtains resource and makes progress.

The sequential transition rule represents the evolution of the system when a sequential computation is carried out. However, the behaviour of a concurrent system is more interesting. In such a system, there are multiple actors sharing resources. Each actor takes a sequence of actions, which represents a sequential computation. Multiple types of resources can be consumed at the same time. The progress of a concurrent DREAM system is given by the **concurrent transition rule**:

$$\mathcal{S}_i \xrightarrow{\xi_1 \rightarrow a_1, \dots, \xi_n \rightarrow a_n} \mathcal{S}_{i+1}$$

which is,

$$\left(\left\{ \bigcup_{i=1}^n \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t, t'_i)}, \Theta \right\}, \left\{ \bigcup_{i=1}^n \llbracket \mathbf{q}_i \rrbracket_{\xi_i}^{(t, t''_i)}, \rho \right\}, t \right) \xrightarrow{\xi_1 \rightarrow a_1, \dots, \xi_n \rightarrow a_n} \left(\left\{ \bigcup_{i=1}^n \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t+\Delta t, t'_i)}, \Theta \right\}, \left\{ \bigcup_{i=1}^n \llbracket \mathbf{q}_i - \mathbf{r}_i \times \Delta t \rrbracket_{\xi_i}^{(t+\Delta t, t''_i)}, \rho \right\}, t + \Delta t \right)$$

Similar to the sequential transition rule, $\llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t, t')}$ is the available resource which has located type ξ_i , $\llbracket \mathbf{q}_i \rrbracket_{\xi_i}^{(t, t'')}$ is the simple resource requirement by actor a_i 's next action associated with resource ξ_i , and Δt is the smallest time slice that the system can account for.

The concurrent transition rule specifies that the system evolves by consuming multiple types of resources during one time interval $(t, t + \Delta t)$, and those resources are used to fuel multiple actors' computations.

Note that in the concurrent transition rule, each computation only consumes one type of resource. This is a matter of granularity: considering resource consumption at a fine grain, a computation usually only requires one type of resource at a time, i.e., it either requires CPU resource for computing, or network resource for transferring data. Memory resources present a

⁶In practice, Δt can be defined according to the desired control granularity.

challenge because they are required for maintaining a computation's state. Although memory required in parallel with processor resources is essentially a type of cache, which is defined in time and space, and hence can be modelled by DREAM, a more detailed analysis is left for future exploration. Therefore, in this work, we assume that memory resources (disk, main memory, cache, etc.) are available as needed.

If certain resource becomes available, yet no computations require that type of resource, the resource expires. The **resource expiration rule** is defined as follows:

$$(\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}, \Theta\}, \rho, t) \xrightarrow{\xi \rightarrow} (\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t+\Delta t, t')}, \Theta\}, \rho, t + \Delta t)$$

The resource expiration rule states that with time Δt elapsing, resource ξ is expired, and no computation makes any progress.

Similarly, a concurrent version of resource expiration rule is defined as follows:

$$(\{\bigcup_{i=1}^n \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t,t')}, \Theta\}, \rho, t) \xrightarrow{\xi_1 \rightarrow, \dots, \xi_n \rightarrow} (\{\bigcup_{i=1}^n \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t+\Delta t, t')}, \Theta\}, \rho, t + \Delta t)$$

The concurrent resource expiration rule specifies the system evolution caused by the expiring of multiple resources.

The transition and resource expiration rules specify extreme cases where either all of the resources available at time t are consumed by actors or all of them expire. To represent a more likely scenario, the two rules can be combined to form a general transition rule, in which some resources are consumed, while others expire, as follows:

$$(\{\bigcup_{i=1}^m \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t,t')}, \Theta\}, \{\bigcup_{i=1}^n \llbracket \mathbf{q}_i \rrbracket_{\xi_i}^{(t,t'')}, \rho\}, t) \xrightarrow[\xi_{n+1} \rightarrow, \dots, \xi_m \rightarrow]{\xi_1 \rightarrow a_1, \dots, \xi_n \rightarrow a_n} (\{\bigcup_{i=1}^m \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t+\Delta t, t')}, \Theta\}, \{\bigcup_{i=1}^n \llbracket \mathbf{q}_i - \mathbf{r}_i \times \Delta t \rrbracket_{\xi_i}^{(t+\Delta t, t'')}, \rho\}, t + \Delta t)$$

Besides the above transitions rules which represent system evolution over time, DREAM also has two sets of transition rules which can be applied at a time instant, representing resource acquisition and computation accommodation/leaving.

In an open system, resources may join or leave the system at any time. The resource acquisition is modeled by the following **resource acquisition rule**:

$$(\Theta, \rho, t) \xrightarrow{\Theta_{join}} (\Theta \cup \Theta_{join}, \rho, t)$$

where Θ_{join} is the resource set which joins the system at time t . Note that there is no such a transition rule for resources leaving, because resources join only for a time interval, at the end of which they are claimed to leave the system. If a resource is going to leave the system in the future, the time of leaving must be explicitly specified at the time of joining the system.

Similar to the resources, computations in an open system may arrive or leave at any time. The transition rule for **computation accommodation** is as follows:

$$(\Theta, \rho, t) \xrightarrow{+(\Lambda, s, d)} (\Theta, \rho \cup \rho(\Lambda, s, d), t)$$

where $t < d$, which means that it is not possible to accommodate a computation if its deadline has passed.

Similarly, DREAM has a **computation leaving** rule to represent a computation leaving the system:

$$(\Theta, \rho, t) \xrightarrow{-(\Lambda, s, d)} (\Theta, \rho \setminus \rho(\Lambda, s, d), t)$$

where $t < s$. That is to say, a computation which has already started in the system is not allowed to leave. We make the assumption $t < s$ to simplify the model.

Syntax and Semantics

The well formed formulas ψ are defined as follows:

$$\begin{aligned} \psi ::= & \text{true} \mid \text{false} \mid \text{satisfy}(\rho(\gamma, s, d)) \mid \\ & \text{satisfy}(\rho(\Gamma, s, d)) \mid \text{satisfy}(\rho(\Lambda, s, d)) \mid \\ & \neg\psi \end{aligned}$$

A DREAM well formed formula can be an atomic proposition, which may be the value *true*, *false*, a “satisfy” function on a resource requirement ρ ,⁷ or a well formed formula with a logic operator “ \neg ” (not).

⁷The three variants of “satisfy” are for a single actor action, a sequential (single actor) computation, and a concurrent (multi-actor computation), respectively.

The semantics of DREAM are defined by the satisfaction symbol \models , on a *computation path*, which is defined as follows.

Definition 2: Computation Path *Let $\chi \subset \mathcal{S} \times \mathcal{S}$ be a binary relation such that $(\mathcal{S}_i, \mathcal{S}_j) \in \chi$ if there exists a transition rule $\mathcal{S}_i \xrightarrow{\xi \rightarrow *} \mathcal{S}_j$, where ξ is a resource located type, and “*” can be either an actor’s name, or empty. A computation path is one branch⁸ of the tree frame that relation χ on \mathcal{S} produces.*

Therefore, the tree structure that relation χ on \mathcal{S} produces represents all the potential evolutions of the system, and a computation path expresses one of the possible traces of the computation.

As shown in Figure 3.1, the DREAM semantics is defined by the satisfaction symbol \models on a computation path σ at time t . I assume the system state that σ, t specifies is $\mathcal{S} = (\Theta, \rho, t)$, and $\bigcup_{\max(s,t)}^d \Theta_{\text{expire}}$ gives the union of the resource sets which will expire during the time interval $(\max(s,t), d)$ according to path σ .⁹ In other words, these are unwanted resources which will expire unless new computations requiring them enter the system. This creates opportunity for the system to accommodate new computations.

DREAM can be used to express resource related properties of a distributed system. For example, we can answer a question such as: “Can computation Γ begin at time t and complete by deadline d ?” Theorem 3 deals with this situation.

Theorem 3: Meet Deadline *Suppose the state of the system is $\mathcal{S}_0 = (\Theta, \emptyset, t)$, having Θ resources but no computations to use them at time t , the computation Γ can be completed by deadline d , if and only if there exists a computation path σ , denoted by $(\mathcal{S}'_0, \mathcal{S}_1, \dots, \mathcal{S}_n)$, where $\mathcal{S}'_0 = (\Theta, \rho(\Gamma, t, d), t)$, such that $\mathcal{S}_n = (\Theta', \emptyset, t_n)$, and $t_n \leq d$.*

Proof. Assume we have such a computation path σ , at each of the time points when a subcomputation Γ_i is completed, we divide the path. So at the end we get m sub-paths, each of which represents a subcomputation of Γ, s, d . Apparently the time points we get, t_1, t_2, \dots, t_n satisfy the complex resource requirement of (Γ, s, d) , according to Theorem 2, the computation can be completed by time d . This proves “*if.*”

Assume the computation (Γ, s, d) can be completed by the system. Since the system tree

⁸A branch represents a possible system evolution, represented by a sequence of system states, starting from the root (starting state) to a leaf node (ending state) of the tree.

⁹I use function *max* here because the computation’s earliest start time s may be already passed.

$\mathbb{M}, \sigma, t \models true$
$\mathbb{M}, \sigma, t \not\models false$
$\mathbb{M}, \sigma, t \models satisfy(\rho(\gamma, s, d))$
iff $f(\bigcup_{max(s,t)}^d \Theta_{expire}, \rho(\gamma, s, d)) = true$
$\mathbb{M}, \sigma, t \models satisfy(\rho(\Gamma, s, d))$
iff $\exists t_1, \dots, t_{m-1}$, such that
$s \leq t_1 < \dots < t_{m-1} < d$,
$\rho(\Gamma, s, d) = \rho(\Gamma_1, t_1, t_2) \cup \dots \cup \rho(\Gamma_m, t_m, d)$,
and $\mathbb{M}, \sigma, t_1 \models satisfy(\rho(\Gamma_1, t_1, t_2))$,
$\mathbb{M}, \sigma, t_2 \models satisfy(\rho(\Gamma_2, t_2, t_3))$,
... ..
$\mathbb{M}, \sigma, t_m \models satisfy(\rho(\Gamma_m, t_m, d))$
$\mathbb{M}, \sigma, t \models satisfy(\rho(\Lambda, s, d))$ where
$\rho(\Lambda, s, d) = \rho(\Gamma_{a_1}, s, d) \cup \dots \cup \rho(\Gamma_{a_n}, s, d)$
iff $\mathbb{M}, \sigma, t \models satisfy(\rho(\Gamma_{a_1}, s, d))$
and $\exists \sigma'$, such that
$\mathbb{M}, \sigma', t \models satisfy(\rho(\Gamma_{a_2}, s, d) \cup \dots \cup \rho(\Gamma_{a_n}, s, d))$
where the state \mathbb{M}, σ', t specifies is
$(\Theta, \{\rho \cup \rho(\Gamma_{a_1}, s, d)\}, t)$
$\mathbb{M}, \sigma, t \models \neg\psi$ iff $\mathbb{M}, \sigma, t \not\models \psi$

Figure 3.1: DREAM Semantics

represents all possible evolutions of the system, there must be a path σ , in which all actions in Γ are completed by time d . This proves “only if.” \square

The next question is whether a system can accommodate a new computation at a certain state, without affecting the existing computations in the system, as shown in Theorem 4, which answers the question raised in Section 3.1.2.

Theorem 4: Accommodate Additional Computation A new computation (Γ, s, d) can be accommodated, without affecting the current executing computations in the system, if there exists a computation path σ , such that resources which are expiring on σ during the time interval (s, d) , i.e., $\bigcup_s^d \Theta_{expire}$, satisfies the complex resource requirement of computation (Γ, s, d) .

Proof. Because resources $\bigcup_s^d \Theta_{expire}$ on path σ satisfies the complex resource requirement of computation (Γ, s, d) , according to Theorem 3, we can build a path σ_Γ starting from state $(\bigcup_s^d \Theta_{expire}, (\Gamma, s, d), s)$, which eventually reaches a state $(\Theta', \emptyset, t_{end})$ where $t_{end} < d$. We then combine the two paths σ and σ_Γ , in the way that we combine the transition rules of σ_Γ to the transition rules in σ , which has the same start and end time, to form new concurrent transition rules. We call the new path generated by the combination σ' , which is a path which accommodates computation (Γ, s, d) , without affecting the existing computations which are already accommodated. According to Theorem 3, the computation (Γ, s, d) can be accommodated by the system, without affecting other computations. \square

3.2 Resource Coordination Mechanisms

DREAM reasoning has been implemented and integrated into an optimized implementation of the Actor model, for providing fine-grained distributed resource coordination. In this section, I first present the challenges of the integration and the approach I take in Section 3.2.1. A detailed description of the integration, as well as the tuner which dynamically balances overhead against the extent of control exercised are presented in Section 3.2.2. Section 3.2.3 describes the architecture of the prototype implementation.

3.2.1 Challenges

Coordinating delivery of resources to distributed computations with deadlines is a challenging problem. Comparing to other distributed runtime systems, such as PVM [124], DARTS [54], and Cilk [20], the Actor model is more suitable for implementing resource coordination, because it offers a convenient way for coordinating resources among computations by encapsulating objects along with threads of execution. Particularly, processor resources delivered to an actor can be controlled by appropriately scheduling the actor's thread of execution [72], and network bandwidth available to a communication can be controlled by reserving a part of the bandwidth for it [70]. However, it turns out that in practice, one-thread-per-actor implementations of actors are not particularly efficient; it is more efficient to have a pool of threads, each of which processes messages for multiple actors. The control flow of an actor is represented as a *continuation* [14]. This optimization strategy has recently been shown [77] – for ActorFoundry, a Java library implementing Actor primitives – to deliver performance that comes close to that of Erlang [15], an efficient direct implementation of Actor semantics. ActorFoundry carries out internal *continuation passing style* (CPS) transforms using a byte-code post-processor [120], which recognizes invocations of blocking methods and transforms the code in the caller.

It turns out that the optimization which makes Actor systems efficient also makes coordination of resources between sub-computations more difficult. For instance, processor cycles can no longer be distributed between actors by simply creating a schedule which a processor scheduler can then enforce. The question we asked was: is this efficiency worth the lost ease of coordination? In other words, can coordination mechanisms be installed in the optimized ActorFoundry in a way that largely preserves the efficiency gains of the optimization? Particularly, I carefully examined the optimizing mechanisms for opportunities for control, and then tried to exercise it efficiently.

In the following section, I present the efforts in this direction, which have shown that not only is the thread-of-execution level allocation of resources better suited for globally efficient fine-grained concurrency, but also its benefits can be largely preserved when supporting control mechanisms. Critically, installing control mechanisms in an implementation

optimized for globally efficient fine-grained concurrency, although challenging to do so, comes at a fraction of the cost of having a separate thread for every actor.

3.2.2 Installing A Resource Control Mechanism

To be precise about what we mean by distributed computations, we take them to be actor computations which are spread over a distributed execution space. Actors are the *de facto* model of concurrency underlying a number of languages, e.g., Scala [58], Erlang [15]. While a number of these are Java library implementations, Erlang [15] is directly implemented. Erlang, because of its efficiency, is of interest to us. Despite the advantages it has because of its direct implementation, it establishes a sort of performance standard by which other implementations can be compared. However, for multiple reasons, of even greater interest is ActorFoundry. Most notably, ActorFoundry attempts to faithfully implement the Actor model, which is well understood, and its code is uniquely accessible because of its modular design. For these reasons, ActorFoundry was selected in this work for the prototyping.

Because delivery of processor resources is essentially through the scheduler, the focus of my attention is ActorFoundry's scheduling mechanism. As illustrated in Figure 3.2, ActorFoundry's scheduler schedules an initially-fixed number of native JVM threads¹⁰ called *workers*, which in turn select from among the actors waiting in a *waiting queue* to get a chance to execute. The waiting queue is shared by the worker threads. An actor waits in this queue only when it has received a message in its own *message queue*; at all other times, it is essentially dormant, and does not need to execute. Whenever a worker becomes free, it picks an actor from the waiting queue to execute. The waiting queue is a FIFO queue. Actors are placed in the queue according to the order in which they received the first messages in their message queues. After dequeuing an actor from the waiting queue, a worker calls the actor's continuation. The worker continues to execute the dequeued actor until all messages in its queue have been processed. After completing with one actor, the worker dequeues another actor and starts to execute it. We can say that ActorFoundry's sched-

¹⁰The number of worker threads is increased at run time when found to be insufficient for progress in the computation. The number of workers represents the parallelism of the system. Specifically, in a multicore system, the number of workers can be set to be equal to the number of cores to achieve the maximum parallelism.

uler is *message-driven* in the sense that only actors which have received messages in their queues get to be on the waiting queue; any other actors stay off of it. It is obvious from the way that ActorFoundry schedules actors, that it cannot support timeliness requirements of computations.

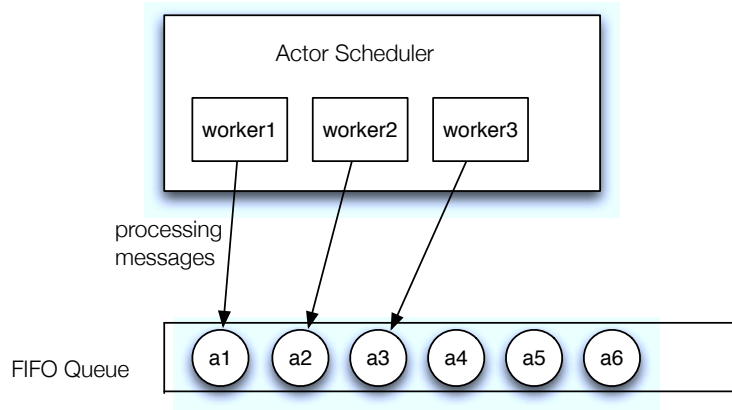


Figure 3.2: ActorFoundry Scheduler

Because ActorFoundry itself does not support computations with timeliness constraints, I introduce the notion of *deadline* to ActorFoundry. I identify the actions to which deadlines apply, and then define the various deadlines as follows. The deadline for a (typically multi-actor) computation is specified by the user/programmer, indicating the time by which the computation is required to be completed. The deadline for an actor is the deadline by which an actor should complete processing all pending messages in its message queue; this deadline is essentially the same deadline as the one by which the computation containing the actor should be completed. A more interesting type of the deadline and the key to supporting fine-grained resource control is the last, the deadline for a message: this is the deadline by which processing of the message should be completed by its destination actor. The deadline for an actor to process one of its messages depends on the actor's deadline and the remaining messages which are to be processed. A more detailed discussion about how to calculate per-message deadlines can be found in Section 4.2.1.

Extending ActorFoundry

I extended the ActorFoundry framework by integrating a DREAM reasoning component into the scheduler, and adding a tuner facility to observe and adjust the ratio of resources consumed by computations and the reasoning mechanism. Figure 3.3 shows the architecture of the modified scheduler.

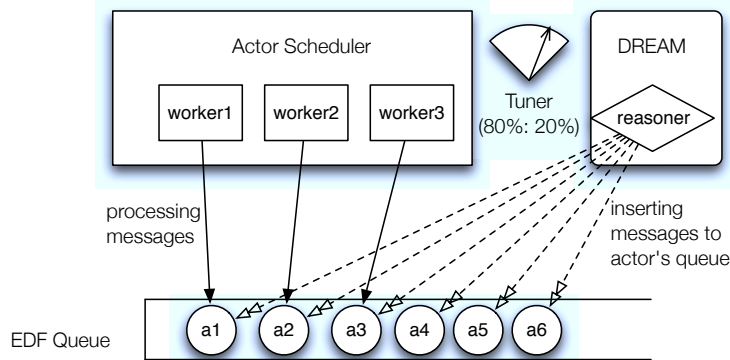


Figure 3.3: Integrating DREAM Reasoning into ActorFoundry Scheduler

First, I modify ActorFoundry’s essentially *message-driven* scheduler so that it becomes a *deadline-driven* scheduler. In other words, I replace the FIFO queue with an Earliest Deadline First (EDF) queue. Recall that we are interested in computations which have predefined deadlines, by which they are expected to be completed. When there are multiple actors executing as part of a computation, by default, each actor is thought to have the same deadline as that of the entire computation. All actors that are waiting for execution are placed in a priority queue according to these deadlines. In other words, the actors in the system are scheduled on a Earliest Deadline First (EDF) basis, which has been shown to be the *optimal* scheduling algorithm on preemptive uniprocessors [33], in the sense that if a set of real-time jobs can be scheduled by any other scheduling algorithm, they can also be scheduled by EDF. Furthermore, the message queue in each actor is changed to an EDF queue as well. Therefore, the messages received by an actor are sorted according to their deadlines. A more detailed discussion about how to calculate per-message deadlines can be found in Section 4.2.1.

Second, I use a special meta-actor, DREAM Reasoner (DR), to carry out resource reasoning using the DREAM model. DR is responsible for carrying out the resource-related tasks, including making decisions about resource allocation and enforce those decisions. DR is the key component where resource coordination tasks are separated from computation tasks. Note that different resource coordination *policies* can be employed by DR, for fulfilling different coordination requirements. Several example policies are discussed in the following chapters (Chapter 4, 5 and 6) as case studies.

Third, a *tuner* facility is developed to perform meta-level resource control. Because the reasoning mechanism itself consumes computational resources (amounting to the overhead), this tuner offers a means to balance the division of resources consumed by the computational actors and those consumed by the reasoner. This is possible to do trivially on a single processor because of the way in which I have implemented the reasoner as a separately scheduled meta-actor. The tuner can be fixed at a particular division of resources, it can be set to automatically react to observed progress of the computation, or it can be made available to a system operator in the form of a *tuning knob*. In a fixed setting, the ratio between the processing power taken for the computation vs. that for the reasoning (e.g. 80%:20%) can be initialized at the beginning of the computation. The following section discusses *self-tuning*, one way in which the tuner can set itself reactively.

Self-Tuning

If the user/system administrator would like the system to take over the high level resource control, they can set the tuner facility to be *self-tuning*. In such cases, during the execution, the tuner automatically adjusts the division of resources consumed by actors carrying out computations and the resource reasoning according to the following *events* it observes: reasoning too fast, reasoning too slow, and reasoning too costly.

Reasoning too slow:

A reasoning-too-slow event is triggered when the workers run out of actors to execute, yet the reasoner still has messages to release. In every scheduling cycle, the system checks if the workers are making progress. Reasoning too slow event is triggered if some of the workers are frequently idle, but the reasoner has more messages to be processed. It indicates that

the reasoner needs more resources than it currently occupies. In this case the tuner facility will change the ratio to increase the resources assigned to the reasoner.

Reasoning too fast:

A reasoning-too-fast event is triggered if the reasoner completes processing of all requests, yet it still owns some unused resources. This event indicates that the system has assigned more resources to the reasoner than what it needs, which slows down the progress of computations. In the runtime, whenever the reasoner runs out of messages to process, reasoning too fast event will be triggered. In this case the tuner facility will adjust the ratio, so that the resources left over by the reasoner are allocated to the actors carrying out computations. Note that if there are more requests coming in, the tuner facility will not change the ratio until a reasoning-too-slow event is triggered. This adjustment can help the computations make more progress.

Reasoning too costly:

A reasoning-too-costly event is triggered when the resources consumed by the reasoner have reached an *upper limit*, another parameter of the tuner facility that can be set manually. Upon observing this event, the tuner facility has two choices. First, it can completely shut down the reasoning component, and return its resources to actors carrying out computations, because in this case, having the reasoner may not help to generate a better schedule, given the resources that can be assigned to it. Instead, it only takes resources away from the computations. Once the reasoning component is shut down, the system becomes the original ActorFoundry, with Earliest Deadline First scheduling for actors, because the actor waiting queue has already been constructed as a priority queue when the system is initialized. For individual actor messages, since no deadlines are to be associated with them, they will be processed according to the arrival order, like in ActorFoundry. Second, the tuner facility can keep the ratio at the upper limit, and perform as much reasoning as possible, given the resources available to it. In this case, a *message filter* will be constructed, which releases a number of messages without reasoning, so actors can perform computations without delay.

Through the above adjustments, the tuner facility oversees the resource consumption division of computations and the reasoner, and attempts to achieve a balanced value at the end. Also an upper limit can be set for resources consumed by resource reasoning, in order

to prevent the reasoner from consuming too much resources which may eventually affect the overall performance of the system.

3.2.3 Implementation

A prototype implementation has been developed by extending ActorFoundry. ActorFoundry supports distributed computations by supporting actors at a number of nodes, and enabling communication between actors across node boundaries as well as actor migration.¹¹ The architecture of one AF-D (ActorFoundry with DREAM reasoning) foundry node is shown in Figure 3.4.

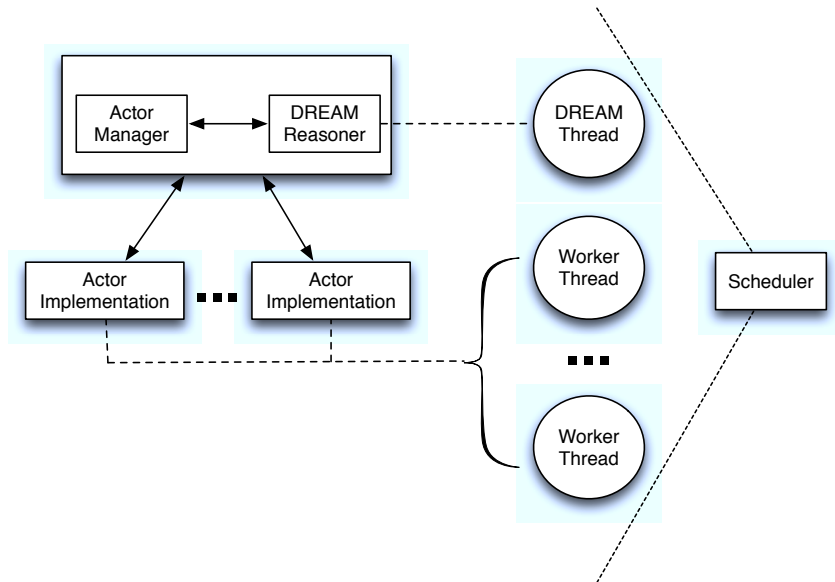


Figure 3.4: System Architecture (AF-D Node Instance)

The *Actor Manager*, together with the newly added component, *DREAM Reasoner* (DR), serve the core runtime functions of a foundry node. They are responsible for creating and scheduling new actors, as well as handling messages between local actors, and moving messages between the local foundry node and other foundry nodes. When the first actor is created as part of a computation – typically how a multi-actor computation is initiated –

¹¹This is made possible by the fact that actors have globally unique names, with mapping between the names and actual physical locations tracked using distributed name tables.

its deadline is set to be the computation’s deadline, as specified by the user. Any actor subsequently created by an actor is assigned the same deadline as its creator’s.

Whether the resource reasoning is enabled or not can be decided at the time of initiating the computation’s execution. If the resource reasoning is enabled, every actor message which involves the local foundry node – specifically, an entirely local communication, a message from a local actor intended for a remote actor, or a message for a local actor from a remote actor – is examined by the DR as a reasoning request. Upon receiving a reasoning request, the DR makes resource allocation decisions for the message, allocates local resources if the message is to be processed locally, or dispatches it if its destination is a remote actor (for the allocation to subsequently happen at the remote location).

Algorithm 1 shows how the system generates a resource allocation for processing of an actor message at the location of a message’s recipient. These allocations are made for each of the sequences of actions requiring no change in the type of resource required. These sequences of actions are referred to as *segments*.¹² The input parameters of the algorithm, therefore, are: available resources in the form of resource terms; and the resource requirements for processing each *segment* of the method corresponding to the message. The algorithm looks to accommodate the message using available resources,¹³ and returns a schedule for the message in the form of resources being reserved for it, or *null* if a feasible schedule is not found. Note that in this algorithm, when searching for available resources to accommodate computations, the *first fit* heuristic is used, which is a standard heuristic used in grid computing.

The complexity of Algorithm 1 is $O(s \times t \times r)$, where s is the number of segments of continuous resource use per message, t is the number of resource types each segment requires,¹⁴ and r is the number of resource terms. In any real application, t is usually a small constant;¹⁵ s too is typically a small constant, but depends on the granularity of control

¹²Specifically, a sequence of actions, each requiring the same single type of resource, can be combined into one segment; an action which requires multiple types of resources is considered as one segment, because these types of requirements must be reasoned about separately from others to assure simultaneous availability of multiple resources.

¹³When searching for available resources, the granularity of located type of the available resources must match the granularity of resource used in specification of the computation’s requirements.

¹⁴Here we mean something specific by types: the same kind of resource (say, processor) at two different nodes is considered two types. To be more precise, we use the term *located type*.

¹⁵In some contexts (such as choice of network links), t can be large because of conditionality and related uncertainty. However, when only one segment is being considered at a time, t typically will not be large.

Algorithm 1 AccommodateMessage(*terms, requirements*)

```
1: schedule = null
2: start = end time of the previous message accommodated
3: end = start /* starting from earliest start time, try to find end time of the message */
4: s = number of message segments
5: for  $i = 1$  to  $s$  do /* sequentially accommodate message segments */
6:    $t =$  number of resource types in  $requirements[i]$ 
7:   for  $j = 1$  to  $t$  do /* reason about multiple types of required resources separately */
8:     reserved = null /* record reserved resources for the segment */
9:      $r =$  number of resource terms
10:    for  $k = 1$  to  $r$  do /* traverse resource terms */
11:      select terms during (start, deadline), in resource type  $requirements[i][j]$  /* pick
12:        all resources of required type which exist before message's deadline */
13:    end for
14:    look for time instant  $end'$  such that portion of selected terms defined during
15:      ( $start, end'$ ) =  $requirements[i][j]$  /* try to allocate sufficient resources to complete
16:      execution of segment */
17:    if  $end'$  does not exist then /* not enough resource (type  $j$ ) available before deadline
18:      */
19:      cancel reservation, return null
20:    else /* enough resource is found */
21:      reserve selected terms during (start,  $end'$ )
22:      schedule += reserved /* add reserved resources to schedule */
23:      if  $end' > end$  then
24:         $end = end'$  /* update the end time of the segment */
25:      end if
26:    end if
27:  end for
28:  start = end /* set up earliest start time of the next segment */
29: end for
30: return schedule
```

desired. In other words, the complexity is typically $O(r)$.

The number of resource terms r during the course of reasoning depends on the fragmentation of resources. There are a number of ways in which we can improve on this complexity. Recall that resource terms are defined in time and space. This means that only a small subset of the resource terms will be relevant to a computation which needs to be carried out during a time interval (defined by the earliest start time and deadline). A way of filtering the set of resource terms for an interval of time will significantly reduce the number of resource terms to be considered. Also, the most fragmentation is likely to happen in the nearest future. This can be good and bad. It is bad because computations which need to be accommodated in the near future have to contend with reasoning involving a larger number of resource terms; it is good because the resource terms in the near future will expire once their time of existence has passed. In ongoing work, I am looking at ways to control this complexity by managing this fragmentation. Most interestingly, I am looking at another tuning opportunity involving the distance in the future to look into to accommodate a sufficiently permitting computation. In other words, one way to balance the resources devoted to reasoning versus the actual computation would be to decide how far in the future to look for the needed resources. I envision a tuning knob for this type of meta-level control very similar to the one described previously for balancing ratio of processing devoted to reasoning versus the computation.

Note that in the current implementation, DREAM Reasoner is not distributed. That is to say, in a distributed system, there is a DREAM Reasoner installed on each individual node, and it is only responsible for reasoning about resource availability on its own node. Any messages that are sent to actors on other nodes will be delivered, and handled by the DREAM Reasoner on the remote node where the destination actor resides. Reasoning about the resources on remote nodes requires global knowledge of the resource availability, which can be achieved by periodically communication among DREAM Reasoners about updates on local resource availability [73].

ActorFoundry's scheduler has been rewritten to accommodate DREAM Reasoner. A dedicated thread is used for carrying out DREAM reasoning. The scheduler schedules the DREAM thread and the collection of worker threads in turn according to the ratio set through the tuner. ActorFoundry's implementation of actors is essentially unchanged in my extension

except for the fact that there is a deadline associated with each actor’s completion.

All the modifications/extensions to ActorFoundry are made in the scheduling infrastructure, and separated from the functional code of actors. Therefore, they do not affect the correctness of computations, or the way actors are programmed. However, since the resource control happens dynamically at runtime, it presents challenges in testing and debugging the resource control. Some of the measurable metrics for resource coordination include the maximum amount of resource (in percentage) taken by the resource coordination mechanism, and the control outcomes depending on the purpose of the coordination. For example, for QoS support, one of the measurable control outcomes is the inter-stream skew. In such a context, the *correctness* of the resource coordination could be defined as follows: the overhead does not exceed the predefined maximum resources that can be consumed by the coordination, and the control outcome is within users’ acceptable range.

3.3 Summary

In this chapter, I present DREAM, a Distributed Resource Estimation and Allocation Model, which enables computations to reason about future availability of resources. In DREAM, computational resources are defined over time and space, and represented using resource terms, which specify key attributes of resources. The syntax and semantics of DREAM are described. Theorems are derived to illustrate how the model can be used to express resource properties of the system in time and space, track resource utilization, and calculate future availability of resources. Particularly, at any time, given a computation, it is possible to evaluate whether its deadline constraint can be assured by the available resources.

Then I present the work on integrating DREAM resource reasoning mechanisms into ActorFoundry, an optimized implementation of Actors. A DREAM Reasoner is added for separating resource related decision making and reasoning from the computations. Additionally, a tuner dynamically balances manually or automatically the overhead of the control mechanisms against the extent of control exercised. Different policies for the decision making can be employed by DR, for supporting different types of resource coordination. These policies are discussed in detail as case studies in the following chapters.

CHAPTER 4

CASE STUDY: QoS SUPPORT

The growing popularity of grid and cloud computing has led to a renewed interest in resource control and coordination. The openness and dynamicity of these computing paradigms lead to uncertainty about continuous availability of the needed resources, therefore present challenges for supporting applications which have timeliness constraints, including deadline constraints, and QoS requirements.

I propose to address this challenge by using the resource coordination approach presented in Chapter 3. Particularly, I apply this approach and develop a resource coordination policy for providing QoS support for distributed computations. This is achieved by using deadline-driven adaptive scheduling, which prioritizes individual message deliveries and method executions involved in a distributed computation, based on the calculated fine-grained deadlines by which each must be completed. These deadlines can be efficiently calculated at run-time for an important class of computations which use pipeline interaction style.

In this chapter, related work in the area of supporting QoS control in open environment is reviewed in Section 4.1, then the approach I take is described in Section 4.2. The experimental results are presented in Section 4.3, and finally, Section 4.4 summarizes the chapter.

4.1 Related Work

Both computational grids and clouds require large-scale resource sharing: computational grids offer opportunities for bringing together disparate computational resources to solve relatively large problems; clouds allow users to lease resources/services – provided by service providers in the cloud – on a pay-per-use basis. Both computing paradigms face an important challenge: how to coordinate resource use by heterogeneous computations, especially when

those computations have QoS requirements.

The performance of a grid/cloud application depends on how well the tasks involved in executing the application are matched against and coordinated on the available resources [18]. This is one of the main reasons for the growing interest in resource management and scheduling (RMS) in computations grids/clouds.

Traditional resource scheduling algorithms focus on minimizing the *makespan*, i.e., the total execution time of all jobs, and it has been shown that mapping jobs onto heterogeneous resources while minimizing makespan is an NP-complete problem [67]. A variety of heuristics [22] are therefore adopted to schedule jobs on grids. However, as the diversity of applications using computational grids and clouds grows, the resource requirements which need to be satisfied are also becoming more complex. Minimizing makespan is no longer the only requirement for a scheduling scheme. QoS requirements offer an instructive example, where resources are required at a certain time, for a certain period, and often there is a need for multiple types of resources to be available together. To satisfy such *timeliness* requirements, more sophisticated and finer-grained resource coordination mechanisms are required.

Two commonly used architectures which provide QoS control over grids are GARA (Globus Architecture for Reservation and Allocation) [50] and G-QoS (Grid Quality of Service Management) [4].

GARA [50] separates resource reservation from its allocation, in order to avoid increased costs resulting from excessive over-provisioning, or degradation of service for critical traffic. In GARA, a number of *resource managers* provide reservation, control, and monitoring operations for different resources. An application can make reservations for a resource by contacting the corresponding resource manager. Also, a resource discovery service provides applications access to information about resource properties such as current and future availability. The subsequent work [51] of GARA combines resource reservation with application adaptation to allow applications to adapt to resource conditions by responding to explicit or implicit feedback.

In the context of Open Grid Service Architecture (OGSA), G-QoS [4] is a service-oriented framework for supporting QoS management in computational grids. Unlike GARA, G-QoS provides a generic QoS management service which is not coupled with a specific

resource type, or a generic reservation model. G-QoS implements a usage policy service, which enables resource owners to specify their domain-specific rules, such as who is authorized to use the resources, and to what extent.

However, both GARA and G-QoS are reservation-based frameworks, and they require applications to explicitly provide the time when they need resources, based on which a reservation is constructed. Unlike these approaches, my approach presented in this chapter provides more flexibility because applications only need to specify their deadlines, and a reasoning component is responsible for generating a feasible schedule which satisfies the deadline constraints.

Another class of approaches for resource management on grids and clouds includes different types of economic models, which are especially popular in cloud computing because of the *pay-per-use* feature of clouds. In such an economic model, resources/services are organized in a market, and users who require these resources/services must pay for them. Decision making about resource allocation in these models is typically driven by common market mechanisms, such as auctions [96] and negotiations [78].

In a highly dynamic environment, uncertainty creates difficulties for using economic approaches, because both resource demand and supply can change at any time. To address this challenge, An et al. [12] proposed an automated negotiation mechanism for dynamic resource allocation in cloud computing. In this approach, consumers negotiate contracts with providers for resource leases, and a successful negotiation results in a contract which binds a set of resources to the consumer for a fixed time interval. The way of binding resources with time is similar to DREAM, which associates resources with time and space. However, the focus of DREAM is to coordinate resource use by multiple applications so that their deadline constraints can be satisfied, while the automated negotiation mechanism proposed in [12] aims to maximize the utility of resource providers by pricing their resources.

Market-based approaches are often criticized for their performance because of their high overhead, which is often caused by failed negotiations/allocations, or low occupancy of resources. Chard et al. [26] studied resource utilization strategies which aim to reduce allocation failures, increase occupancy and hence increase the performance. Chang et al. [25] proposed another approach to address the performance issue from a different perspective. Instead of

increasing occupancy, i.e., using as many available resources and as efficiently as possible, it tries to use as few resources as possible to reduce cost and administration overhead for setting up the resources. In the work presented in this thesis, I deal with overhead in a different way. Instead of measuring the performance of the system as a whole, I measure the extra overhead caused by the resource allocation mechanism itself, and set it as a parameter which can be tuned by users. More specifically, the proposed approach also provides users the flexibility to control the overhead, and different overhead settings result in different levels of optimality of the schedule.

4.2 Approach

The resource coordination approach presented in Chapter 3 can be adapted for providing QoS support, and the foundation of this adaptation is an identification of finer-grained actions to which deadlines apply. In other words, although a multi-actor distributed computation may have a deadline by which it needs to be completed, that in itself is too coarse grained an information to enable effective control. On the other hand, too exact a scheduling would be too costly in terms of scheduling overhead. It turns out that there is a middle ground where scheduling granularity is just fine enough that it offers sufficient control for a variety of applications. This is what the evaluation seeks to illustrate. The granularity of control we aim for is at the level of an actor's processing of a message. In other words, an actor's method execution will not be pre-empted once it has begun.

The deadlines at different levels are defined as follows. The deadline for an entire (typically multi-actor) computation is specified by the user/programmer, indicating the time by which the computation ought to be completed. Individual actors participating in the computation have deadlines by which they must finish processing all messages in their individual queues.¹ The deadline for an actor participating in a computation defaults to the entire computation's deadline. A more interesting type of the deadline – the one which determines the granularity

¹We make a simplifying assumption that an actor continuously participates in a single computation until the computation is completed. To generalize, we would need to track messages by the computations they belong to, and an actor would have deadlines associated with each computation it participates in, and then have separate deadlines for completed processing of all messages associated with each computation.

of control offered by this approach – is the deadline for processing of individual actor messages. The deadline for an actor to process one of its messages depends on the actor’s deadline for completing all processing and the computation required for processing any remaining messages. These deadlines can be calculated essentially by counting back from the entire computation’s completion deadline, accounting for the time required for processing each message (i.e., executing the method required for processing it). Although this is non-trivial in the general case, it is possible to do efficiently for computations where one interleaving of the computations can be decided, and processing cost of each non-communicating segment of computation is either provided or can be estimated. This is discussed in greater detail in Section 4.2.1.

Once the completion deadline for a message has been computed, it is tagged onto the message by the runtime system, to be subsequently detached by the recipient actor’s runtime (which may be the same as the sending actor’s runtime) and used in deciding when to schedule delivery of the message to the recipient, and consequently, when to schedule execution of the actor. Particularly, resource terms specifying available resources are matched against (multi-actor) computations’ requirements – represented by the deadlines and the code to be executed by those deadlines – to identify computations which can potentially be accommodated given the available resources. The matching process, which uses DREAM [137], also generates a (possibly distributed) resource allocation schedule for each computation. The enforcement of this schedule is attempted by manipulating the order in which actor messages are processed.²

4.2.1 Deadline Analysis

Calculating deadlines for messages is critical for providing fine-grained QoS control for individual actors.

In general, the deadline for any message can be calculated using a known deadline for the recipient’s subsequent deadline and counting back by the amount of computation and communication required to be carried out before that deadline must be met. This obviously is non-trivial for a computation with sufficiently complex interaction between actors. However,

²Note that modifying the processing order of messages does not violate Actor semantics. As long as messages are eventually delivered, the fairness requirement is satisfied.

for classes of computations – such as those which use the pipeline communication style – this can be efficiently achieved. Examples of such computations include multimedia delivery, as well as concurrent algorithms to solve a wide variety of programs using the pipeline style of interaction. We count back from the entire computation’s deadline to determine message-grained deadlines; this can be done in time linear in the total number of messages. In fact, as is the case with the video-conference example discussed later in Section 4.3, often the pattern is regular enough that it is a matter of tracking the order of messages, which eliminates the need for computing down to the first message, allowing the deadline-computation to proceed alongside the actual computation.

4.2.2 QoS Policy

If the resource reasoning is enabled in the system, every actor message which involves the local foundry node, including local communication, outgoing and incoming messages, must go through the DREAM Reasoner (DR) as a reasoning request.

For providing fine-grained deadline support, a resource coordination policy, QoS policy is developed and employed by DR. Specifically, upon receiving a reasoning request, DR calculates its deadline (as needed), allocates resource if the message is to be processed locally, or dispatches it if its destination is a remote actor. The algorithm of the QoS policy is shown in Algorithm 2. Here, the deadline for the entire computation is provided by the programmer/user, and we assume that the resource requirement for processing each message is known.³

In particular, DR processes different types of messages as follows:

- Local Messages:

For a local message, both sender and receiver are located on the current foundry node. Upon receiving a reasoning request of a local message, DR does the following: First, calculate the deadline for the message (as discussed in Section 4.2.1). Second, compare the required resources of the message – specified by function Φ – to the available re-

³This information can be provided by a table which maps message name to the resources required for processing that message.

Algorithm 2 QoS Policy

```
1: remove expired resources /* a resource expires if it is not used during its available time
   interval */
2: while message queue is not empty do /* there are more requests to process */
3:   dequeue the first message msg
4:   if destination of msg is not local then
5:     calculate deadline d and attach it to msg /* msg is sent to an actor on a remote
       node */
6:     dispatch msg
7:     break
8:   end if
9:   if deadline is null then
10:    calculate deadline d and attach it to msg
11:  end if
12:  if resources available before  $d > \text{msg's required resources}$  then /* msg can be accom-
       modated, see Algorithm 1 */
13:    allocate resources to msg
14:    insert msg to destination actor's queue /* which is an EDF queue */
15:  else /* msg can not be accommodated using available resources */
16:    raise a warning on msg
17:  end if
18: end while
```

sources in the system (stored as resource terms). If there are enough resources available by the deadline, put the message in the recipient actor’s mailbox, allocate resources to the recipient actor. If there are not enough resources available to the message before its deadline, a warning will be raised, indicating a deadline to be missed. Third, if resources are allocated, update the available resources.

- **Outgoing Remote Messages:**

Outgoing remote messages are sent to actors on a remote foundry node by local actors. Upon receiving a reasoning request of an outgoing message, DR only needs to calculate its deadline, attach the deadline to the message, and then return the message to Actor Manager, which will actually send the message to the remote foundry node through the transport layer of the node.

- **Incoming Remote Messages:**

Incoming remote messages are sent to actors on the local foundry node by remote actors. Upon receiving a reasoning request of an incoming message, DR compares the message’s required resources to available resources on the local node. If enough resources are available, DR puts the message to its destination actor’s mailbox, allocates resources to the destination actor, and update resource availability in the system. If available resources are not enough for processing the message, a warning will be raised.

4.3 Experimental Results

Two sets of experiments have been carried out to illustrate the effectiveness and efficiency of the QoS resource coordination policy, including a benchmark example, Threading with deadline constraints, and a example of real-time application – video conference.

4.3.1 Threading Benchmark with Deadline Constraints

Threading is a benchmark developed by Karmani et al. [77] for analyzing performance of actor implementations. In Threading, a number of actors pass a token for a specified number of times. Here in the experiments, I specify the number of actors to be 503, and the number

of token passes to be 2000. Specifically, when the application is initiated, an actor is created, and a start-up message *boot* is sent to it. The *boot* message invokes 503 actor creation calls, and the newly created actors form a thread ring, in the sense that each actor knows the name of its subsequent actor (passed to it as a parameter in creation). Note the subsequent actor of the last actor is set to be the first actor in the array. Then a *passToken* message is sent to the first actor, upon receiving the message, the actor sends the same message to its subsequent. The application terminates when the token has been passed for 2000 times.

In each computation 503 actors pass a token for 2000 times. For each computation with numeric id i ($1 \leq i \leq 10$), I set its deadline to be $i \times t + \Delta_i$, where t is a close approximation of the amount of smallest time in which the hardware can execute one instance of the Threading computation, and Δ_i is an amount of time by which a deadline may be missed. In the experiments, the value used for t was 280ms, and Δ_i is randomly generated within range [0, 400ms].

This set of experiments aims to illustrate the effectiveness of actor scheduling of our approach. The experiments are run on a MacBook Pro laptop with Intel Core Duo CPU @ 2GHz, 2GB RAM and 2MB L2 cache. Specifically, I initiated 10 Threading applications, each of which is associated with its own deadline. I run the same experiment in ActorFoundry, and the extended version of ActorFoundry with DREAM reasoning for providing deadline support (shown as “AF-D” in the Figure). The results are shown in Figure 4.1.⁴

The results show only one out of ten deadlines is missed in AF-D (which is very close to be met though), but ActorFoundry missed 9 deadlines. Obviously, AF does not claim to support deadlines; the comparison with AF here is simply to establish a baseline. The additional overhead of using AF-D (i.e., the cost of supporting the deadlines) included a startup overhead of approximately 60ms, followed by an average of 5ms for each Threading execution completed over the period of approximate 280ms. Discounting the setup cost, the additional overhead of AF-D amounted to 2% above the cost of carrying out the computation using AF.

⁴Note that the experiments were run for 15 times, and the results are similar. Here I only present results from one instance of the runs.

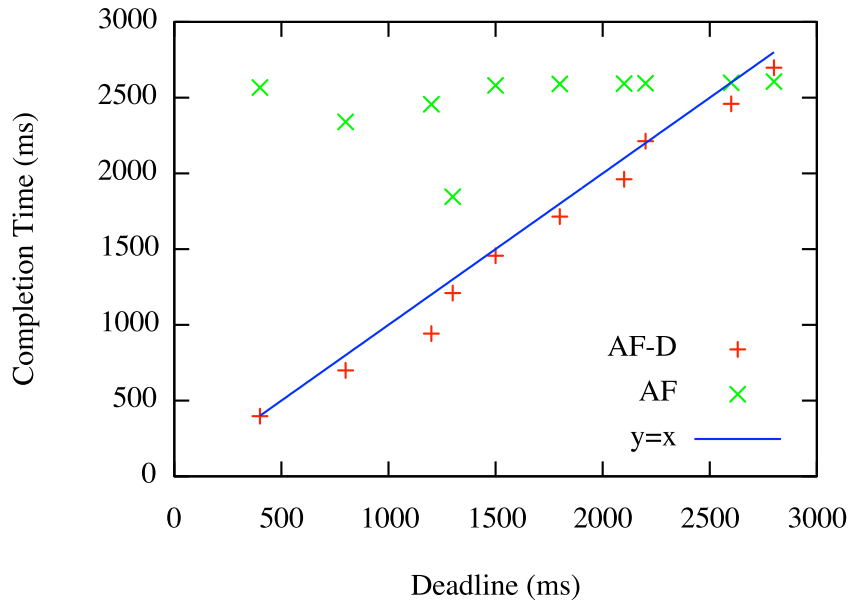


Figure 4.1: Threading Benchmark

4.3.2 Live Video Conferencing

Multimedia applications have been characterized by their strict requirements of Quality of Service, and low fault-tolerance. Because in order to fulfill users' expectations, these applications can only tolerate minor and very infrequent violations of their performance requirements. Therefore, this type of applications present challenges to scheduling and usually require significant size of buffer at the receiver side to smooth jitter, minimize the effects of inter-stream skew, and handle other synchronization issues. Previous work [49, 129] has been done to model multimedia systems and provide synchronization mechanisms using Actor frameworks. However, neither of them has presented an analysis on the cost of the synchronization, comparing to the original Actor framework, not to mention the control over the synchronization cost.

I choose a real-time video conference application to illustrate the power of the approach, because among all multimedia applications, real-time video conference is one of the most demanding applications. In order to guarantee high quality of real-time interaction, it requires very low End-to-End Delay ($< 150\text{ms}$), and low audio/video inter-stream skew ($< 100\text{ms}$).

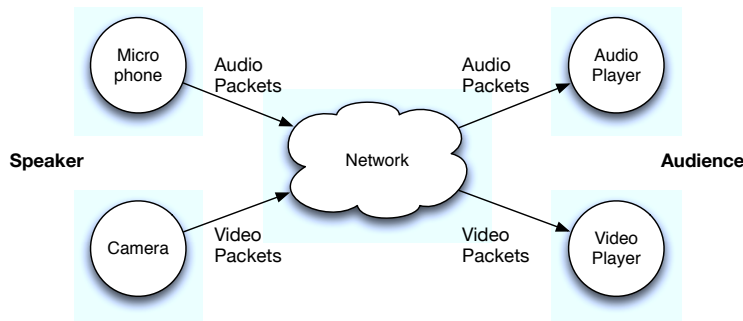


Figure 4.2: Multimedia Application: Video Conference

Figure 4.2 shows the implementation of a video conference session which simulates the commonly used Real-time Transport Protocol (RTP) [116]. RTP is designed for end-to-end, real-time transfer of stream data. As shown in Figure 4.2, a *microphone* actor samples sound signal from the speaker side and encodes it into audio packets. A *camera* actor samples images and encodes them into video packets. Both audio and video packets are transmitted to the audience side through the network. At the audience side those packets are decoded by *audio player* and *video player* actors respectively. When the packets are processed at the receiver side, the video and audio streams should be synchronized, and played at the same frequency as when they are sampled at the speaker side. The acceptable audio/video inter-stream skew is $\pm 100\text{ms}$. Note that the frequencies of audio and video packets are usually 8000Hz and 90000Hz respectively. For convenience, I use the ratio 1:10 in the experiments.

Algorithm 3 Sampling method in Camera Actor

- 1: **for** $i = 0$ to n **do**
 - 2: wait($1/f$) /* f is the frequency of video packets */
 - 3: sample image and encode it into video packet \mathbf{vpk}
 - 4: send \mathbf{vpk} to the remote video player actor /* which will decode and play the packet */
 - 5: **end for**
-

Algorithm 3 shows the **sampling** method in the camera actor. The program code is unchanged except for the fact that a coarse grained deadline is specified. In this example, the deadline is set to be the time when the conference session is to end plus an acceptable

lag. AF-D automatically calculates the deadlines for delivery of each packet – inside separate messages – and then synchronizes the delivery of those messages according to their deadlines. Note that in the implementation, the contents of the video and audio packets are simulated, because in this way the size of the computation can be easily controlled. The simulated data is transmitted over a real network.

In the experiments, for source node (speaker side), I use a MacBook Pro laptop with Intel Core Duo CPU @ 2GHz, 2GB RAM and 2MB L2 cache; for destination node (audience side), I use a Dell XPS laptop with Intel Core Duo CPU @ 2GHz, 3GB RAM and 2MB L2 cache.

Figure 4.3 shows the results of running the example on both ActorFoundry and my approach, AF-D.⁵ In ActorFoundry, because the video and audio packets are not synchronized, packets captured at the same time do not arrive at the same time, and the arrival orders of the packets in each stream are not determined either. Although ActorFoundry has a way to define local synchronization constrains (LSC) [77], it can not specify real-time constraints. The results show that one of the audio packets (ID: 0) is delayed for approximately 1s. If we consider it as a missing packet, the maximum inter-stream skew is still 562ms. Moreover, the playing order of the packets is not properly sequenced.

On the contrary, using AF-D leads to all video and audio streams being properly synchronized, with adequate control of the playing order of packets from both audio and video streams. The maximum audio/video inter-stream skew is 78ms, which is within acceptable range. The extra overhead incurred is 53ms over the course of a computation requiring 1544ms of execution time, which is approximately 3.4%. Note that for these experiments, the run-time system was set up to be self-tuning, so that the balance between the resources consumed by computations and those consumed by resource reasoning was automatically adjusted.

Results from these two sets of experiments illustrate that this approach, although constrained by the opportunities provided by relatively coarse-grained scheduling changes, nevertheless offers a degree of control that is sufficient for applications with timeliness constraints such as Quality of Service requirements. In addition, the approach does not add programming

⁵Note that the experiments were run for 15 times, and the results are similar. Here I only present results from one instance of the runs.

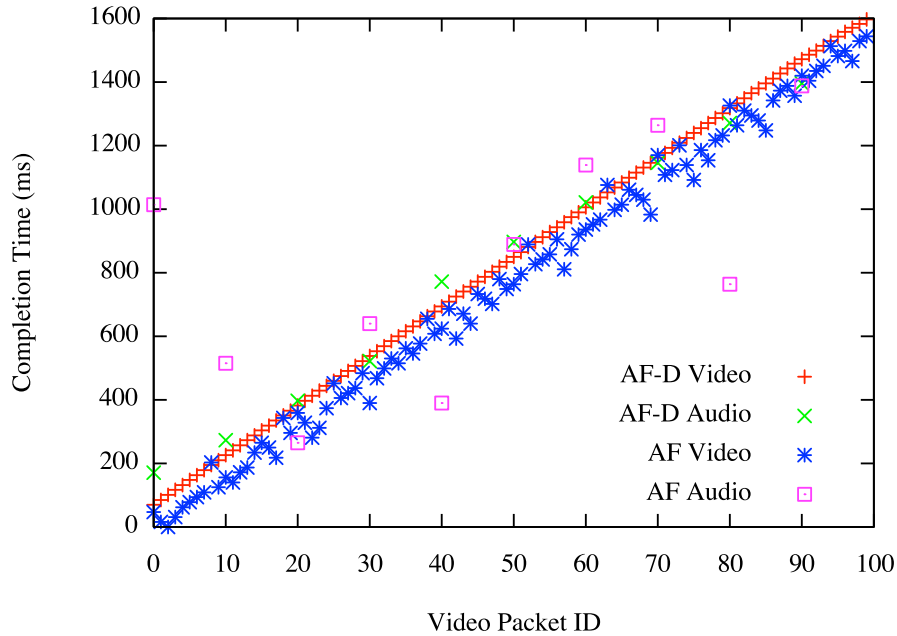


Figure 4.3: Audio/Video Stream Synchronization for Live Video Conferencing

complexity to applications.

4.4 Summary

In this chapter, the resource coordination approach presented in Chapter 3 is adapted for providing QoS support. This is achieved by calculating and enforcing fine-grained deadlines for individual actor messages. A QoS policy is developed and employed by DR for supporting fine-grained deadlines. Experimental results show that the coordination approach can provide effective support for computations with deadline constraints, and QoS requirements. For a class of computations which use pipeline interaction style, the QoS support can be provided efficiently at a relatively modest overhead.

CHAPTER 5

CASE STUDY: POWER-EFFICIENT MULTICORES

There is growing interest in the energy consumed by computer systems, for both individual (battery life) and environmental (global warming) reasons. Multicore architectures offer a potential opportunity for energy conservation by allowing cores to operate at lower frequencies. Most of the previous work on analyzing power consumption of multicores assumes that all cores must run at the same frequency. However, new technologies, such as fast voltage scaling and Turbo Boost, allow cores to operate at different frequencies.

In this chapter, I extend the resource coordination approach presented in Chapter 3 to accommodate energy consumption of computations. This approach provides a flexible way to analyze energy consumption of multicores operating at non-uniform frequencies. This information can then be used to generate a energy-efficient schedule for execution of the computations – as well as a schedule of frequency changes on a per-core basis – while satisfying performance requirements of computations. Experimental results show that the energy savings achieved using this approach far outweigh the energy consumed in the reasoning required for generating the schedules.

The organization of this chapter is as follows. I first present the challenges in Section 5.1, then I review related work in Section 5.2. In Section 5.3, I take two frequency scaling technologies as examples to illustrate the effect of these technologies on energy consumption; Section 5.4 presents the effort on adapting the DREAM model for multicore resource management and energy analysis; experimental results from an example computation are presented in Section 5.5; finally, Section 5.6 summarizes the chapter.

5.1 Challenges

Climate change is one of the major challenges facing the current generation of humanity, and the fact that computers are responsible for 2-3% of greenhouse gas emissions related to human activities has raised interest in power conservation and efficient use of computational resources. As a result, computer architects are shifting hardware design to multicore architectures, and using dynamic voltage and frequency scaling (DVFS) to trade processing speed for power savings. This is motivated by the relationship between a processor's speed and its power requirement: the power consumed by a core is (typically) proportional to the cube of its frequency. In particular, multiple cores running at a lower frequency can deliver the same performance as can be achieved from a single core running at a higher frequency, while consuming less power.

Existing analytical models for power consumption of multicores typically assume that all cores operate at the same frequency [84, 91, 130]. This is a reasonable assumption for current processors which use off-chip voltage regulators (i.e., a single regulator for all cores on the same chip), which set all sibling cores to the same voltage level [106]. However, recent studies [27, 61] based on *Amdahl's law* [11] show that more flexible chip designs, such as asymmetric multicore chips, and dynamic multicore chips, can potentially improve both the speedup and energy efficiency. In addition, new hardware advances have emerged, which enable similar flexibility in frequency scaling. First of all, for off-chip regulators, even though cores on the same chip must operate at the same frequency, if there are multiple chips in the system, cores on different chips may operate at different frequencies, i.e., per-chip frequency [136]. Furthermore, even for cores that are required to execute on a uniform frequency, there are ways to aggressively adjust the frequency for all cores. An example is the Turbo Boost [127] technology, which provides flexibility of frequency control by boosting all cores to a higher frequency to achieve better performance when necessary and possible. Note that the frequency can be increased only when the processor is operating below rated power, temperature, and current specification limits.

In addition, the most recent technology, on-chip switching regulators [81] enable cores on the same chip to operate on different frequencies, providing greater flexibility for fre-

quency scaling. Studies have shown that per-core voltage control can provide significant energy-saving opportunities compared to traditional off-chip regulators [82]. Furthermore, it has been shown recently [80] that on-chip multicore voltage regulators (MCVR) can be implemented in hardware. Essentially a DC-DC converter, the MCVR can take a 2.4V input and scale it down to voltages ranging from 0.4 to 1.4V. To support efficient scaling, MCVR uses *fast voltage scaling* to rapidly cut power according to CPU demands. Specifically, it can increase or decrease the output by 1V in under 20 nanoseconds.

To fully exploit the potential of these technologies, a finer-grained model for power consumption and management is required. Because the frequency of a core represents the available CPU resources in time (cycles/second), it can naturally be treated as a computational resource, which makes it possible to address the problem of power consumption from the perspective of resource management. In this case study, I illustrate that if per-core control is available, the fine-grained resource coordination approach can be used to support power-efficient execution of computations on multicore processors. For this, I adapt the DREAM model for reasoning about energy consumed by concurrent computations executing on multicore processors, and develop mechanisms involved in creating schedules – of resource usage as well as frequencies at which processor cores should execute – for completing computation in an energy-efficient manner.

5.2 Related Work

Although *Moore's Law* has long predicted the advance in processing speeds, the exponential increase in corresponding power requirements (sometimes referred to as the *power wall*) presented significant challenges in delivering the processing power on a single processor. Multicore architectures emerged as a promising solution [1]. Since then, power management on multicore architectures has received increasing attention [76], and power consumption has become a major concern for both hardware and software design for multicore.

Li et al. [91] were among the first to propose an analytical model which brought together efficiency, granularity of parallelism, and voltage/frequency scaling, and to establish a formal relationship between the performance of parallel code running on multicore processors and

the power they would consume. They established that by choosing granularity and voltage/frequency levels judiciously, parallel computing can bring significant power savings while meeting a given performance target.

Wang et al. [130] have analyzed the performance-energy trade-off. Specifically, they have proposed different ways to deploy the computations on the processors, in order to achieve various performance-energy objectives, such as energy or performance constraints. However, their analysis is based on a particular application (matrix multiplication) running on a specific hardware (FPGA based mixed-mode chip multiprocessors). A more general quantitative analysis has been proposed by Korthikanti et al. [84], which is not limited to any application or hardware. They propose a methodology for evaluating energy scalability of parallel algorithms while satisfying performance requirements. In particular, for a given problem instance and a fixed performance requirement, the optimal number of cores along with their frequencies can be calculated, which minimize energy consumption for the problem instance. This methodology has then been used to analyze the energy-performance trade-off [86] and reduce energy waste in executing applications [85].

These analytical studies make an assumption that all cores operate at the same frequency because of the hardware limitation of traditional off-chip regulators – a limitation that is about to be removed by recent hardware advances.

There are a number of scenarios where finer-grained control is possible. Even when off-chip regulators are used, if there are multiple chips, cores on different chips can be operating at different frequencies. For example, Zhang et al. have proposed a *per-chip adaptive frequency scaling*, which partitions applications among multiple multicore chips by grouping applications with similar frequency-to-performance effects, and sets a chip-wide desirable frequency level for each chip. It has been shown that for 12 SPECCPU2000 benchmarks [119] and two server-style applications, per-chip frequency scaling can save approximately 20 watts of CPU power while maintaining performance within a specified bound of the original system.

However, two recent advances in hardware design promise even greater opportunities. The first of these is Turbo Boost [127], which can dynamically and quickly change the frequency at which the cores on a chip are operating during execution. Specifically, depending on the performance requirements of the applications, Turbo Boost automatically allows processor

cores to run faster than the base operating frequency if they are operating below power, current, and temperature specification limits. Turbo Boost is already available on Intel's new processors (codename Nehalem). The second, and perhaps more important, is the emergence of on-chip switching regulators [81]. Using these regulators, the different cores on the same chip can operate at different frequencies. Studies [82] have shown that the energy savings made possible by using on-chip regulators far outweigh the overhead of having these regulators on the chip.

As for commercial hardware, the first generation of multicore processors which support per-core frequency selection are the AMD family 10h processors [10], but the energy savings on these processors are limited, because they still maintain the highest voltage level required for all cores. Most recently, it has been shown that the on-chip multicore voltage regulator together with the fast voltage scaling can be efficiently implemented in hardware [80], which can rapidly cut power supply according to CPU demand, and perform voltage transition within tens of nanoseconds.

These new technologies provide opportunities for energy savings on multicore architectures. However, a flexible analytical model is required to analyze power consumption on multicores with non-uniform frequency settings. Cho et al. addressed part of the problem in [27] by proposing an analysis which can be used to derive optimal frequencies allocated to the serial and parallel regions in an application, i.e., non-uniform frequency over time. Specifically, for a given computation which involves a sequential portion and a parallel portion, the optimal frequencies for the two portions can be derived, which can achieve minimum power consumption while maintaining the same performance as running the computation sequentially on a single core. However, this work is a coarse-grained analysis, and it does not consider non-uniform frequencies for different cores.

Besides theoretical model and analysis, significant work has been done to optimize power consumption at run-time through software-controlled mechanisms, or knobs. Approaches include dynamic concurrency throttling (DCT) [24], which adapts the level of concurrency at runtime based on execution properties, dynamic voltage and frequency scaling (DVFS) [68], or a combination of the two [30]. Among these [68] is particular interesting, because it considers per-core frequency. Specifically, a global multicore power manager is employed which

incorporates per core frequency scaling. Several power management policies are proposed to monitor and control per-core power and performance state of the chip at periodic intervals, and set the operating power level of each core to enforce adherence to known chip level power budgets. However, the focus of this work is on passively-monitoring power consumption, rather than modelling power and resource consumption at a fine grain, and actively deploying computations power-efficiently.

I address the problem from a different perspective: the resource management point of view. The DREAM model presented in Chapter 3 models resources and computations at a fine grain, and models the evolution of the system as the process of resource consumption. In this chapter, I adapt the DREAM model to accommodate energy consumption as the cost/consequence of a specific CPU resource allocation. The adapted model is energy-aware, and can be used to generate an energy-efficient resource allocation plan for a given computation.

5.3 Effect of Frequency Scaling on Energy Consumption

Consider a computation consisting of a sequential part s , followed by a parallel part p ,¹ so that the sequential part must be executed on a single core, and the parallel part can be (evenly or unevenly) distributed over multiple cores,² the analysis carried out in [27] shows how to optimize processor frequency for the case when the parallel part can be evenly divided between a number of cores, as follows.

In order to present the derivation in an intuitive way, we normalize the sequential execution time of the computation to be 1. Similarly, we normalize the amount of work to be 1, i. e., $s + p = 1$. Suppose we want to achieve the same performance in the parallel execution of the computation (the total execution time is 1, as in the sequential execution), and in the parallel execution, the execution time for the sequential part s is t . The frequencies for the sequential and parallel parts in the computation can be calculated as follows:

¹ s and p are amounts of work in the two portions of the computation, i.e., number of CPU cycles.

²Note that although we consider the case where all parallel computation happens in one stretch, this can be easily generalized to a case where sequential and parallel parts of the computation take turn, by having a sequence of sequential-parallel pairs.

$$f_s = \frac{s}{t} \quad (5.1)$$

$$f_p = \frac{1-s}{(1-t) \times N} \quad (5.2)$$

A well accepted formula for calculating energy consumption of a processor/core is as follows [27] :

$$E \propto T_{busy} \times f^\alpha + \lambda \times T \quad (5.3)$$

where T_{busy} is the time during which the computation is carried out; α is the exponential factor of power consumption; ³ λ is a hardware constant which represents the ratio of the static power consumption to the dynamic power consumption at the maximum processor speed. The first term in the formula corresponds to energy consumed for carrying out the computation (dynamic power), and the second term represents energy for the static power consumption during the entire period of execution. Processor temperature is not considered; therefore, energy for static power consumption is only related to λ and T .

For a given computation, s is fixed, and for a given architecture, N and λ are fixed. Hence, according to Equation 5.1, 5.2, and 5.3, for the above computation, the energy consumption can be expressed as a function of t , as follows:

$$E(t) \propto t \cdot f_s^\alpha + N \cdot (1-t) \cdot f_p^\alpha + N \cdot \lambda \quad (5.4)$$

where the first term is the energy consumed by the sequential part of the computation, the second term is the energy consumed by the parallel part of the computation, and the third item represents the static power consumption.

According to Equation 5.1 and 5.2, Equation 5.4 can be rewritten as

$$E(t) \propto t \cdot \left(\frac{s}{t}\right)^\alpha + N \cdot (1-t) \cdot \left(\frac{1-s}{(1-t) \cdot N}\right)^\alpha + N \cdot \lambda \quad (5.5)$$

³In other words, the power consumption of a core running at frequency f is proportional to f^α . As suggested by literature, α is a number between 2 and 3, typically 3.

Next, we obtain the derivative of $E(t)$ with respect to t ,

$$\frac{dE(t)}{dt} \propto \frac{-(\alpha - 1) \cdot s^\alpha}{t^\alpha} + \frac{(\alpha - 1) \cdot (1 - s)^\alpha}{(1 - t)^\alpha \cdot N^{(\alpha-1)}} \quad (5.6)$$

Then, we compute the value of t which minimizes $E(t)$ by setting $dE(t)/dt$ to 0,⁴ and obtain:

$$\frac{t}{1 - t} = \frac{s}{1 - s} \cdot N^{(\alpha-1)/\alpha} \quad (5.7)$$

$$t^* = \frac{s}{s + p/N^{(\alpha-1)/\alpha}} \quad (5.8)$$

Now we can compute the values of f_s and f_p which minimize $E(t)$ using Equation 5.1, 5.2, and 5.8. Therefore, to achieve minimum energy consumption while maintaining a performance identical to running the computation sequentially on a single core processor, the optimal frequencies for executing the sequential and parallel parts (f_s^* and f_p^* , respectively) are:

$$f_s^* = \frac{s}{t^*} = s + \frac{p}{N^{(\alpha-1)/\alpha}} \quad (5.9)$$

$$f_p^* = \frac{1 - s}{(1 - t^*) \cdot N} = \left(s + \frac{p}{N^{(\alpha-1)/\alpha}}\right) \cdot N^{-\frac{1}{\alpha}} = \frac{f_s^*}{N^{\frac{1}{\alpha}}} \quad (5.10)$$

In the following sections, I illustrate the effects of non-uniform frequency scaling on multicore energy consumption. Particularly, I extend the analysis in [27] to consider two specific technologies: per-core frequency, and Turbo Boost.

5.3.1 Per-Core Frequency

It turns out that when parallel workload cannot be evenly distributed among multiple cores, per-core frequency scaling can be used to achieve energy savings. This has been enabled by the latest technologies which support per-core frequency setting in multicore architectures [80]. I illustrate this for a simple case involving only 2 cores. Let us say that the ratio of the workloads on the 2 cores is ($q > 1$). The performance requirement for the computation is 1,

⁴Note that the second derivative is positive.

i.e., the computation must be completed in time $T = 1$. If the two cores must run at the same frequency, the optimal frequency is:

$$f_{uniform} = s + \frac{q}{1+q} \times p$$

If the cores can operate at different frequencies, i.e., using non-uniform frequency scaling, the optimal frequencies are:

$$\begin{aligned} f_1 &= s + \frac{q}{1+q} \times p \\ f_2 &= f_1/q \end{aligned}$$

Obviously, the frequency at which the core executing the sequential part of the computation executes, remains unchanged regardless of the whether uniform or non-uniform frequencies are employed. I assume that the same core carries out the heavier of the two uneven workloads to be carried out in parallel. Any energy savings to be achieved from non-uniform frequency scaling are therefore on the other core operating at a lower frequency.

I first calculate the time period for the parallel part (let us call it T_p) of the computation, which is the focus of our attention:

$$T_p = \frac{p \times q / (1+q)}{s + p \times q / (1+q)}$$

Recall that p is the normalized size of the parallel part of the computation ($p = 1 - s$), and $q > 1$ is the ratio of the two uneven workloads. Next, I calculate the energy savings ΔE :⁵

$$\begin{aligned} \Delta E &= E_{uniform} - E_{non-uniform} \\ &\propto \frac{T_p}{q} \times f_1^3 - T_p \times f_2^3 \\ &\propto T_p \times \left(\frac{1}{q} - \frac{1}{q^3} \right) \times f_1^3 \end{aligned} \tag{5.11}$$

For a given computation, the right hand side is a function of s and q . Figure 5.1 illustrates the energy savings which result from using per-core frequency scaling for the two cores.

⁵We use Equation 5.3 to calculate energy consumption, and use the value of 3 for α , as is typical in the literature.

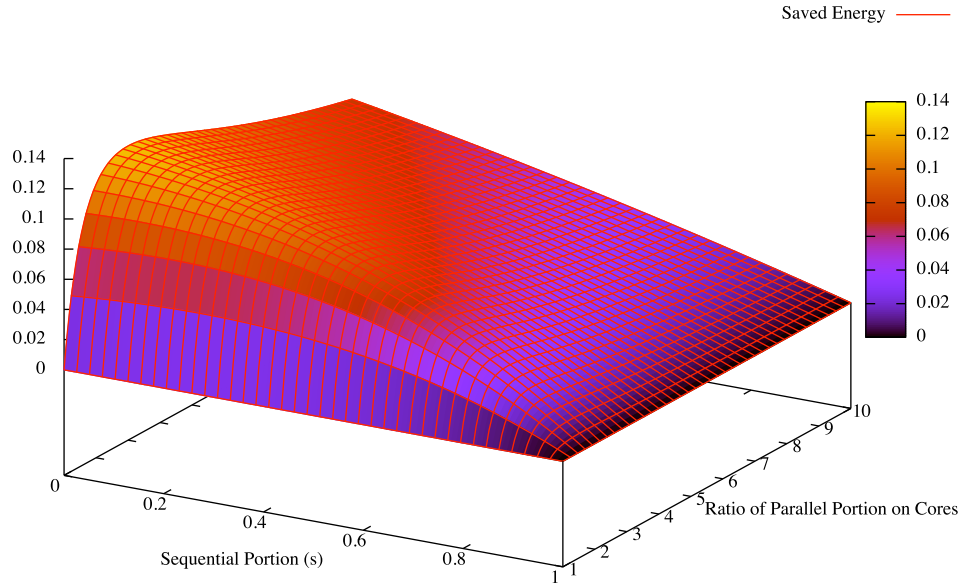


Figure 5.1: Saved Energy on Non-uniform Per-Core Frequency Technology

5.3.2 Turbo Boost

When per-core frequency scaling is not available, turbo boost enables cores to vary their frequency during a computation; the boost is only for a short duration for now to avoid overheating. I now examine the opportunity for energy saving by using this facility. Consider N cores. If all cores must execute at the same frequency over the course of a computation, the frequency required for achieving the same performance can be computed as follows:

$$f_{uniform} = s + \frac{1-s}{N}$$

The time required for completion of the parallel part of the computation would be:

$$T_p = \frac{p/N}{s + p/N} = \frac{p}{s \times N + p}$$

Because static power consumption does not change (by definition), we only consider the energy for dynamic power consumption of the two frequency scaling approaches. Energy required for the computation using uniform frequency is:

$$E_{uniform} \propto f_{uniform}^3 + (N-1) \times T_p \times f_{uniform}^3 \quad (5.12)$$

As derived in [27], the optimal energy consumption when using non-uniform frequency scaling (using the optimal frequencies f_s^* and f_p^* shown in equation 5.9 and 5.10) is:

$$E_{non-uniform} \propto \left(s + \frac{1-s}{N^{2/3}}\right)^3 \quad (5.13)$$

The energy saved by utilizing turbo boost technology is:

$$\begin{aligned} \Delta E &= E_{uniform} - E_{non-uniform} \\ &\propto \left(s + \frac{1-s}{N}\right)^3 \times (1 + (N-1) \times T_p) - \left(s + \frac{1-s}{N^{2/3}}\right)^3 \end{aligned} \quad (5.14)$$

The above formula is a function of s and N , as plotted in Figure 5.2. It shows that using turbo boost can save energy comparing to using uniform frequency for all cores.

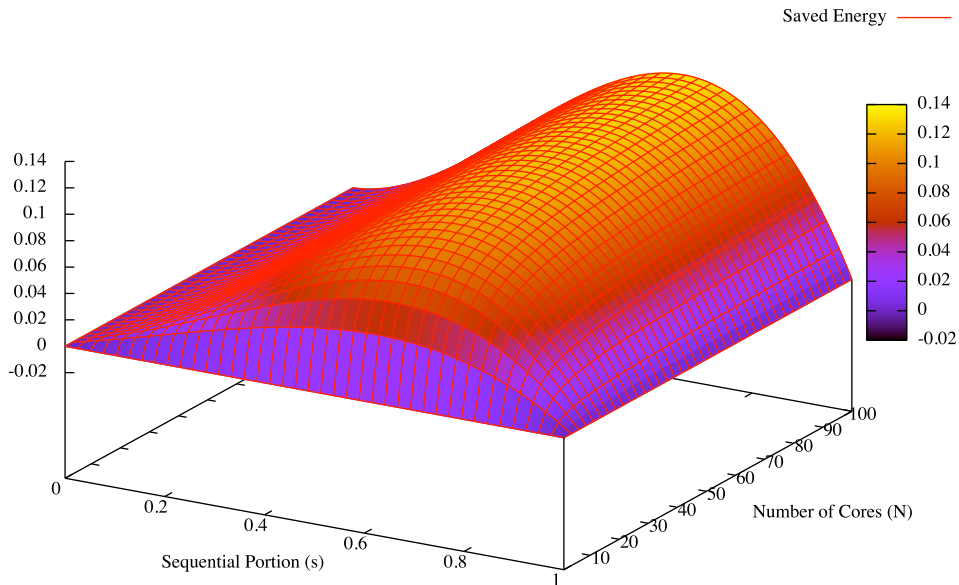


Figure 5.2: Saved Energy on Turbo Boost Technology

The analysis thus far has shown that energy savings can be achieved by using non-uniform frequency technologies. However, the scenario in the analysis is simple: only one computation is considered, and workload and structure of the computation is well known. Next I address the problem of finding the optimal *frequency schedule* for a complex computation, with frequencies varying multiple times over the course of the computation's execution.

5.4 Reasoning about Multicore Energy Consumption

In Chapter 3, I have introduced DREAM [137] and related mechanisms [139] for reasoning about scheduling of deadline constrained concurrent computations over parallel and distributed execution environments. Fundamental to this work is a fine grained accounting of available resources, as well as the resources required by computations. Here, I connect the use of resources by computations to the energy consumed in their use, leading to a specialized model, called DREAM-MCP (*DREAM for Multicore Power*). DREAM-MCP defines multicore processor resources over time and space, and represents them using *resource terms*. A resource term specifies values for attributes defining a resource: specifically, maximum available frequency, time interval during which the resource is available, and location of existence for the resource (i.e., the core id). Computations are represented in terms of the resources they require. System state at a specific instant of time is captured by the resources available at the instant and the computations which are being accommodated. Labeled transition rules are used to represent the progress in the system, and a utility function is associated with each transition rule to indicate the energy required for carrying out the transition.

5.4.1 Representing Multicore Resources and Computations

In DREAM-MCP model, multicore processor resources are represented using *resource terms* in the form of $\llbracket \mathbf{r} \rrbracket_{\xi}^{\tau}$, where \mathbf{r} represents the maximum available frequency of the specific core (in *cycles/time*), τ is the time interval during which the resource is available ($\mathbf{r} \times \tau$ is the number of CPU cycles over interval τ), and ξ specifies the *location* of the available resource, which is the *id* of the specific core.

As in the original DREAM model, Interval Algebra [9] is used for representing relations between time intervals. Each time interval τ has a start time t_{start} , and an end time t_{end} . Note that we also use (t_{start}, t_{end}) as an alternative notation for time interval τ .

Resources in a multicore system can be represented by a set of resource terms. The computations between two resource sets, including union, relative complement, and the inequality operator, are defined as same as in the original DREAM model, presented in Chapter 3.

A computation consumes resources at every step of its execution. As in DREAM, I

abstract away what a distributed computation does and represent it by the sequence of its resource requirements for each step of execution.

In this case study, as the first step towards reasoning about resource / energy consumption of computations, I assume that computations only require CPU resources. I represent a computation using a triple (Γ, s, d) , where Γ is a representation of the computation, s is the earliest start time of the computation, and d is the deadline by which the computation must complete. Particularly, the computation does not seek to begin before s and seeks to be completed before d . I assume the resource requirement of a computation Γ can be calculated by function ρ , as follows:

$$\rho(\Gamma, s, d) = [\mathbf{q}]^{(s,d)}$$

where \mathbf{q} represents the CPU cycles the computation requires.

We say that the above resource requirement is a *simple resource requirement*, and it can be satisfied if there exists a core ξ , such that for all ξ -related resource terms which are during (s, d) $[[\mathbf{r}_i]_{\xi}^{\tau_i}$:

$$\sum_i (\mathbf{r}_i \times \tau_i) \geq \mathbf{q}$$

The above formula specifies the total available CPU cycles during (s, d) are more than the resource requirement \mathbf{q} , and it can be used to determine whether a computation (Γ, s, d) can be accommodated using available resources in the system.

Note that for a computation which is composed of sequential and parallel portions, its resource requirement can be represented by several simple resource requirements which should be simultaneously satisfied.

5.4.2 DREAM-MCP

For a computation that can be accommodated, different scheduling schemes result in different levels of energy consumption. To model all possible system evolution paths and the effects they have on overall energy consumption, DREAM-MCP models system evolution as a sequence of states connected by labeled transition rules, which specify multicore resource allocation, and represents energy consumption as a utility function associated with each transition rule.

We define \mathcal{S} , the state of the system as $\mathcal{S} = (\Theta, \rho, t)$, where Θ is a set of resource terms, representing future available resources in the system, starting from time t ; ρ represents the resource requirements of the computations that are accommodated by the system at time t ; and t is the point in time when the system's state is \mathcal{S} .

The evolution of a multicore system is denoted by a sequence of states $(\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n)$, and the progress of the system is regulated by a labeled transition rule:

$$\mathcal{S}_i \xrightarrow{f, \xi \rightarrow \Gamma} \mathcal{S}_{i+1}$$

where ξ is a core id, f is the utilized frequency for ξ , and Γ is a computation. The transition rule specifies that the utilization of CPU resource on core ξ – which is operating at frequency f – for computation Γ makes the system progress from state \mathcal{S}_i to the next state \mathcal{S}_{i+1} . If we replace the states in the above transition rule with the detailed (Θ, ρ, t) format, the transition rule can be written as:

$$(\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}, \Theta\}, \{\llbracket \mathbf{q} \rrbracket^{(t,t'')}, \rho\}, t) \xrightarrow{f, \xi \rightarrow \Gamma} \{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t+\Delta t, t')}, \Theta\}, \{\llbracket \mathbf{q} - f \times \Delta t \rrbracket^{(t+\Delta t, t'')}, \rho\}, t + \Delta t$$

where $\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}$ is the available resource of core ξ , $\llbracket \mathbf{q} \rrbracket^{(t,t')}$ is the resource requirement of Γ , and Δt is the smallest time slice that the system can account for. Every time a transition rule is applied to the system, the system progresses one step further by time Δt (which can be set for the desired control granularity). Here, the transition rule states that during the time interval $(t, t + \Delta t)$, the available resource ξ is used to fuel computation Γ . As a result, by time $t + \Delta t$, the computation Γ 's resource requirement will be $f \times \Delta t$ less than it was at time t .

Note that the actual frequency for core ξ , f , may be different from the maximum available frequency \mathbf{r} ($f \leq \mathbf{r}$). This enables cores to operate at lower frequencies for saving power.

The energy consumption associated with the above transition rule is represented by a utility function u :

$$u = \Delta t \times f^3 + \lambda \times \Delta t$$

where the first term represents energy for dynamic power consumption and the second represents energy for static power consumption, λ is a hardware constant.

Note that if certain resource becomes available, yet no computations require that type of resource, the resource expires. The resource expiration rule is defined as follows:

$$(\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t,t')}, \Theta\}, \rho, t) \xrightarrow{\xi} (\{\llbracket \mathbf{r} \rrbracket_{\xi}^{(t+\Delta t,t')}, \Theta\}, \rho, t + \Delta t)$$

The energy consumption for an expired resource only includes static power: $u = \lambda \times \Delta t$.

If there are multiple cores in the system, and during a time interval $(t, t + \Delta t)$, some resources are consumed, while others expire, we use a more general *concurrent* transition rule to represent this scenario:

$$(\{\bigcup_{i=1}^m \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t,t'_i)}, \Theta\}, \{\bigcup_{i=1}^n [\mathbf{q}_i]^{(t,t''_i)}, \rho\}, t)$$

$$\xrightarrow[\xi_{n+1} \rightarrow \dots \xi_m \rightarrow]{f_1, \xi_1 \rightarrow \Gamma_1, \dots, f_n, \xi_n \rightarrow \Gamma_n} (\{\bigcup_{i=1}^m \llbracket \mathbf{r}_i \rrbracket_{\xi_i}^{(t+\Delta t,t'_i)}, \Theta\}, \{\bigcup_{i=1}^n [\mathbf{q}_i - f_i \times \Delta t]^{(t+\Delta t,t''_i)}, \rho\}, t + \Delta t)$$

The utility function for the above concurrent transition rule is:

$$u = \sum_{i=1}^n (\Delta t \times f_i^3) + m \times \lambda \times \Delta t$$

where the first term represents energy for dynamic power consumption, and the second represents energy for static power consumption. Note that non-uniform frequency scaling allows f_i to have different values for different “ i ”, where uniform frequency requires them to be the same.

DREAM-MCP represents all possible evolutions of the system as sequences of system states connected by transition rules. Energy consumption of an evolution path can be calculated using the utility functions associated with the transition rules on that path; consumptions of these paths can then be compared to find the optimal schedule. In addition to exploring heuristic options, our ongoing work is also aimed at explicitly balancing the cost of reasoning against the quality of solution (see Section 5.6).

5.5 Experimental Results

A prototype of DREAM-MCP has been implemented for multicore processor resource management and energy consumption analysis. The prototype is implemented by extending

ActorFoundry [77]. A key component of DREAM-MCP is *Reasoner*, which takes as parameters the resource requirements of a computation and its deadline, and decides whether the computation can be accommodated using available resources in the system. For computations that can be accommodated, the *Reasoner* generates a fine-grained schedule, as well as a *frequency schedule* which instructs the system to perform corresponding frequency scaling.

To illustrate the prototype, I implement the Gravitational N-Body Problem (GNBP) as an example. GNBP is a traditional physical problem which aims to predict the motion of a group of celestial objects which exert a gravitational pull on each other. I choose GNBP as an example in this case study, because it represents a large class of computations which consist of a sequential portion and a parallel portion, which can be naturally distributed on a number of cores. The way I implement GNBP is as follows. A *manager* actor sends the information about all bodies to the actors corresponding to the bodies, which use the information to calculate the forces, velocities, and new positions for their bodies, and then send their updated information to the *manager*. In this computation, the sequential portion is that the *manager* gathers all information about bodies, and sends it to all bodies, and the parallel portion is that each individual body calculates its new position, and sends a reply message to the *manager*.

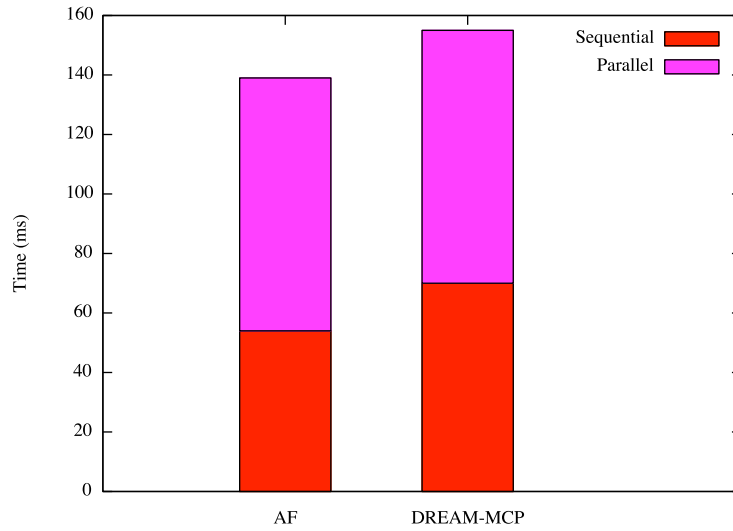
To evaluate the computational overhead caused by DREAM-MCP, experiments were carried out to assess the execution times of the computation on DREAM-MCP and ActorFoundry (AF) respectively. I measured the extra computation time (overhead) that is taken by DREAM-MCP, comparing to AF. The hardware I use is an Xserve with 2×Quad-Core Intel Xeon processors (8 cores)@ 2.8GHz, 8GB memory and 12MB L2 cache. For the same GNBP application, the execution times for the two systems, DREAM-MCP and ActorFoundry (AF), are shown in Table 5.1, and Figure 5.3. ⁶ Note that the processors run at maximum frequency.

As illustrated in Table 5.1, the extra overhead caused by the reasoning is 16ms, which is approximately 11.5%. Because *Reasoner* is implemented as a single Java native thread which is scheduled to execute exclusively, the overhead it causes is in the form of sequential computation. We then normalize the GNBP execution time to 1, and we can calculate

⁶The results are average execution times from 15 runs. The detailed data is presented in Appendix B.

Table 5.1: GNBP: Execution Time at Maximum Frequency

System	Sequential portion (ms)	Parallel portion (ms)	Overhead (%)
DREAM-MCP	68	83	9.4%
AF	55	83	0

**Figure 5.3:** GNBP: Execution Time at Maximum Frequency

energy for dynamic power consumption of the two systems using Equation 5.12 and 5.13 in Section 5.3. We also calculated the extra energy consumption by reasoning itself. As shown in Figure 5.4, by consuming extra 1.9% of the energy requirement of the computation, DREAM-MCP can achieve approximately 21.6% of energy saving. The saving is calculated using the computation schedule and per-core frequency schedule the reasoner generates, and we assume that the hardware supports per-core frequency control.⁷

Note that since the reasoning increases the total execution time of the computation, energy for static power consumption also increases. From Equation 5.3 in Section 5.3 (assuming we

⁷I did not have access to the actual hardware, so the savings have been estimated analytically. Actual savings may be different if measurements were made on actual hardware because of factors such as temperature.

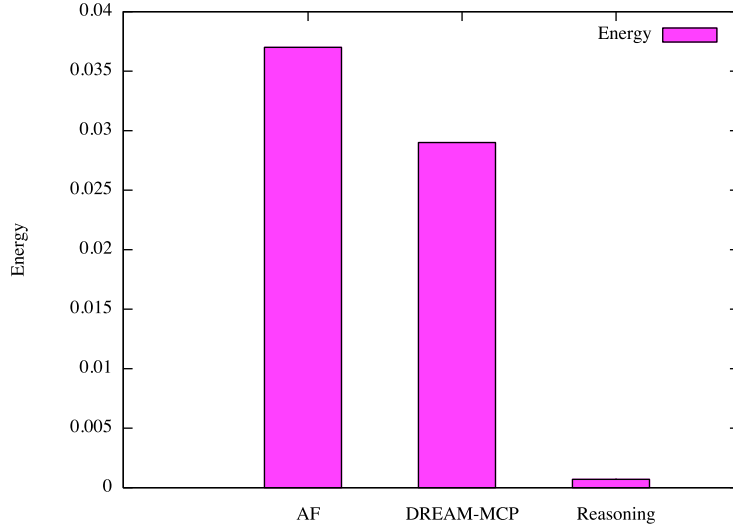


Figure 5.4: GNBP: Energy Saving and Cost

ignore processor temperature), it is only related to λ (hardware constant) and T (execution time), i.e. $E_{static} \propto \lambda \times T$. Because the computational overhead of using DREAM-MCP is 9.4%, extra energy for static power consumption is also 9.4% of the total static energy required by the computation. Depending on the hardware constant λ , the total energy saving (or loss) by using DREAM-MCP for executing GNBP problem can be calculated.

5.6 Summary

Power consumption of multicore architectures is becoming important in both hardware and software design. Existing power analysis approaches have assumed that all cores on a chip must execute at the same frequency. However, emerging hardware technologies, such as fast voltage scaling and Turbo Boost, provide opportunities for energy conservation by selecting different frequencies for individual cores on a chip at a finer grain. Deciding what these frequencies should be – the next challenge – is non-trivial.

Here, after an analysis of the energy conservation opportunities presented by two important recent hardware advances, I extended the DREAM model on fine-grained scheduling of resource use, in order to support reasoning about energy consumption. This reasoning

enables creation of fine-grained schedules for the frequencies at which multiple cores should operate for energy-efficient execution of concurrent computations, without compromising on performance requirements. The experimental evaluation shows that the cost of the reasoning is well worth it: it requires only a fraction of the energy it helps save.

In addition to exploring more efficient reasoning, I am working on explicitly rationalizing the cost of reasoning. Particularly, I hope to customize the tuner that was previously developed – for balancing computational cost of creating fine-grained resource use schedules versus cost of carrying out the actual computation – to enable similar choices in terms of energy consumption.

CHAPTER 6

CASE STUDY: DYNAMIC LOAD BALANCING

Dynamic load balancing is critical in achieving high performance in parallel systems. Traditional approaches for dynamic load balancing often assume that the computations being redistributed have the same size. These approaches do not scale when the computations have non-uniform sizes. In this chapter, I adapt the resource coordination approach presented in Chapter 3 for providing dynamic load balancing support. This is achieved by reasoning about resource utilization, and controlling resources at a fine grain. The approach is implemented using actors for providing clear separation of concerns, code modularity and reusability. In addition, I propose a Unbalanced Cobwebbed Fruit Tree (UCFT) benchmark, which addresses the diversity of computations, in terms of size and priority. Experimental results show that this approach can effectively balance the workload, while promptly responding to computations with higher priority. When the computations' sizes are highly variant, this approach comes close and even outperforms the traditional work sharing and work stealing approaches, which do not address separation of concerns, or the priority of computations.

The organization of this chapter is as follows. I first give an introduction and motivation of the work in Section 6.1, then I review related work in Section 6.2; Section 6.3 presents UCFT tree, the new benchmark application for evaluating the approach; Section 6.4 describes the coordination scheme which integrates DREAM into ActorFoundry, followed by a detailed description of the implementation in Section 6.5; experimental results are discussed in Section 6.6; and finally, Section 6.7 summarizes the chapter and proposes some future directions of the research.

6.1 Introduction

Developments in computer hardware present opportunities for solving large problems in parallel in order to achieve high performance. In a parallel system, how the jobs are scheduled on the processors is critical and has a significant impact on the overall performance. To maximize the overall performance, ideally the idle time of processors should be kept minimum. However, the task of assigning work to processors is not trivial. For applications with constant workload, static load-balancing can be used as a pre-processor of computations, which attempts to evenly distribute workload to processors. On the contrary, for applications which have unpredictable or continuously changing workload, substantial dynamic load-balancing is required. For such applications, achieving load-balancing is even more challenging because of the uncertainty. Traditional approaches for dynamic load-balancing completely focus on partitioning the workload as computation proceeds, and they try to redistribute subcomputations – which are often assumed to have the same size – among processors. However, on the one hand, they are not suitable for computations which are non-uniform in terms of the size; on the other hand, they do not consider the cases where certain computations require prompt responses.

In this chapter, I propose a novel approach for dynamic load-balancing based on fine-grained resource-reasoning. Instead of solving the problem from the perspective of partitioning computations and assigning workload to processors, I view the problem from the perspective of resource management and control, and achieve dynamic load-balancing by matching available resources with computations efficiently, and minimizing the amount of resources that are expiring.

A number of benchmark applications have been developed for evaluating dynamic load-balancing approaches. The commonly-used one is the *unbalanced tree search* (UTS) benchmark [109], the task of which is to count the number of nodes in an unbalanced tree, starting from the root node. Visiting a tree node instantly reveals its children nodes. The imbalance of the tree demonstrates the need for balancing workload. UTS represents a class of applications with highly unpredictable workload requiring continuous dynamic load balancing. However, unlike the UTS benchmark, where the size of the computation on each node is

identical, in reality, the sizes of computations are usually non-uniform. In addition, despite the performance concern, some computations may require more prompt response comparing to others, such as computations in a natural-disaster monitoring and detecting system [125]. Based on these observations, I have developed a new benchmark, *Unbalanced Cobwebbed Fruit Tree* (UCFT), which is similar to UTS, but is adapted to address the above concerns. I have evaluated our resource-reasoning based approach of dynamic load-balancing using the new benchmark, and the experimental results show that this approach can achieve comparable or even better performance comparing to the traditional work sharing and work stealing approaches, while reserving the flexibility of taking into account the priorities of computations. Especially when the sizes of computations are highly variant, the resource-reasoning based approach shows better scalability and even outperforms the traditional work stealing approach. In addition, implementing the approach using actors can take advantage of the programmability, modularity and reusability of actor systems.

6.2 Related Work

In this section, related work in dynamic load-balancing, as well as benchmark applications for evaluating those approaches are reviewed.

6.2.1 Dynamic Load Balancing

Load balancing is the process of redistributing workload of computations among available processors, and it is a key factor in improving the performance of parallel systems [13]. Load balancing has recently received more and more interest, because of the emerging multicore architectures which dramatically increase the scale of parallelism [62]. Generally, two different load-balancing strategies can be utilized depending on the nature of the computations. If the amount of computation is known or can be accurately estimated (e.g., based on average), static load-balancing can be used, which is usually implemented as a pre-processor for computations and employs simple algorithms, such as Round Robin, and Randomized algorithms [104]. Static load-balancing is easy to implement; however, its effectiveness requires accurate estimates of the computations' sizes. In reality, accurate estimates about

computations may not be possible. The uncertainty of computations poses challenges for load balancing. If the amount of computation can not be determined or predicted before the computation is actually carried out, redistribution of the workload is required throughout the course of the computation. Dynamic load-balancing can be applied in this case, which is the focus of this work.

Dynamic load-balancing takes into account the system status and redistributes the workload as the computation proceeds. It has been shown that even very simple dynamic-load-balancing strategies which only collect small amounts of system state information can lead to significant performance gains [40]. Two different strategies are commonly used: work sharing [90, 93] and work stealing [21].

In the work sharing strategy, processors share a global FIFO queue, which is used to place unassigned tasks. An under-utilized processor which requests workload is assigned the first available task in the queue. Work sharing is a centralized approach which utilizes a shared queue to facilitate load balancing. Therefore, it is believed to be suitable for shared memory systems [37].

Unlike the work sharing strategy, the work stealing [21] strategy takes a distributed approach. Each processor maintains a local task queue, and processes the tasks in the queue. An under-utilized processor attempts to get workload from other processors' queues. Work stealing is a decentralized approach, and it is well suited for distributed systems. In addition, because the burden of finding and migrating workload is on under-loaded processors, the added overhead on the loaded processors which are making progress is minimized. Therefore, the work stealing strategy is considered to be more stable comparing to work sharing. The scalability of the approach has been investigated in [36], and it has been shown that work stealing strategy can be implemented efficiently. Recently different variants of work stealing algorithms are developed and utilized in large-scale distributed systems [36], grids [114], and multi-core clusters [115].

More detailed comparisons of work sharing and work stealing strategies including performance analysis can be found in [39], [105], and [131].

Most of the existing approaches on dynamic load-balancing focus on partitioning and distributing uniform-sized workload to processors. In this chapter, I present a different

approach based on fine-grained resource coordination and control.

6.2.2 Benchmarks for Dynamic Load-Balancing

A number of benchmark applications have been proposed for evaluating the performance of approaches for dynamic load-balancing. The NAS parallel benchmark [17] is a set of numerical aerodynamic simulations, which are developed for performance evaluation of highly parallel supercomputers. NAS consists of five parallel kernel benchmarks and three simulated application benchmarks. These benchmarks simulate the computation and data movement characteristics of large scale computational fluid dynamics applications. Biomolecular simulations have also been used as load-balancing benchmarks because of their dynamicity and the large scale of computations involved, such as NAMD [113] (nanoscale molecular dynamics), and Mol3D [111]. Both of them are molecular dynamics programs that simulate biomolecular systems. LBTest [140] is a synthetic benchmark. In LBTest, a collection of objects are created based on customizable parameters, and these objects communicate with each other, obtain data and carry out computations.

A benchmark which is more relevant to the work presented here is the unbalanced tree search (UTS) benchmark [109]. UTS provides a function to create an unbalanced tree using a set of parameters, such as shape, depth, size, and imbalance (a measure of the variation in the size of its subtrees). The benchmark problem is the parallel exploration of the created tree. The performance of a load balancing scheme can be evaluated by measuring the computation time of performing an exhaustive search on the tree. The tree is implicitly constructed, that is, each node contains all information necessary to construct its children. However, a parent node must be visited before its children nodes. Since the tree could be highly unbalanced, there is a high variation in the sizes of the subtrees of a specific parent node. The variation presents challenges in performing an efficient search, and requires continuously dynamic load balancing. The UTS benchmark is a representation of applications that must enumerate a large state space of unknown or unpredictable structure, such as many search and optimization problems. The UTS benchmark has recently been widely used and implemented using difference languages and architectures [37, 110].

In this chapter, for the purpose of simulating real world computations, I adapt the UTS

benchmark and develop a new benchmark application, unbalanced cobwebbed fruit tree (UCFT), for evaluating dynamic load balancing approaches.

6.3 Unbalanced Cobwebbed Fruit Tree

UTS is an effective benchmark for evaluating dynamic load balancing approaches because it nicely captures the uncertainty of the workload of an application: visiting a parent node may reveal an unknown number of children nodes, each of which may connect to a subtree of unknown size. However, UTS does not take into consideration two common characteristics that real world computations usually have. First, the sizes of real world computations are usually non-uniform. Second, besides performance, under some circumstances, the prompt response is also critical for a computation. For example, in natural disaster monitoring and detection, such as earthquake simulation [125] and distributed weather forecasting [72], a disaster may be detected at anytime, and prompt responses are critical for the region where the disaster is detected. When no disaster is detected, performance is the main concern. Based on these observations, I have adapted UTS benchmark and developed a new benchmark, unbalanced cobwebbed fruit tree (UCFT), to address these two characteristics for simulating the real world workload.

An UCFT tree is an implicitly constructed unbalanced tree, as in the UTS benchmark. An example UCFT tree is shown in Figure 6.1.

Each node in an UCFT tree contains a task, instead of a data structure, as was the case in UTS. Each task has a pre-defined size, shown as the number in each node in Figure 6.1, representing the computation time it requires to carry out the task. The tasks are the “cobwebs” which slow down the traversal process. Note that the task on a tree node is indivisible while performing load balancing. The size of the tasks follows a normal distribution,¹ with predefined mean and standard deviation. These values are specified as parameters when the tree is created. Besides the computation size, another newly added attribute is the *priority* of tasks. Some nodes are defined as *urgent* nodes, shown in red in Figure 6.1, which represent tasks that require prompt response. The urgent nodes are the “fruit” on the tree. The

¹This is a truncated normal distribution, so that it does not go negative.

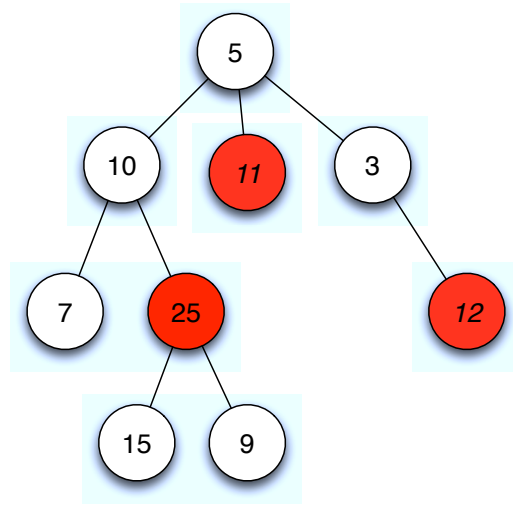


Figure 6.1: Unbalanced Cobwebbed Fruit Tree

number of urgent nodes is defined as a parameter when the tree is created, in the form of a percentage of the total number of nodes in the tree, i.e., size of the tree. Therefore, in addition to the original parameters for creating a UTS tree, three new parameters are added: mean and standard deviation of the computation sizes, as well as the percentage of urgent nodes.

The benchmark problem is to traverse the UCFT tree, and carry out the task at each node when the corresponding node is visited. As in the UTS benchmark, the nodes in the tree can be traversed in parallel and in any order, as long as a parent node is visited before its children nodes.² The goal is to complete all computations in the tree as fast as possible, and in the meantime, process urgent computations as quickly as possible once they are identified.

The unbalanced cobwebbed fruit tree benchmark aims to simulate real world computations, where limited information about computations (i.e. size) is available while uncertainty is also present. When a node is visited, the information of its computation size is known, but visiting the node may reveal an unknown number of children nodes which may connect to subtrees of highly variant sizes.

When a load-balancing approach is evaluated using the UCFT benchmark, besides the

²The child nodes can be enqueued before the computation associated with the parent node is complete.

overall performance of the traversal, how sensitive the approach is in terms of responding urgent computations is another measurement.

6.4 Resource-Coordination Framework for Load Balancing

The ultimate goal of load balancing is to improve the overall system performance by utilizing as much available resources as possible. Therefore, it can be viewed as a resource management problem. Here I present a novel approach for dynamic load balancing based on fine-grained resource coordination and control.

As presented in Chapter 3, the framework of ActorFoundry with DREAM reasoning can provide deadline support and power-efficiency support for distributed computations by matching available resources with computations' requirements. The matching is carried out by Dream Reasoner (DR), and the purpose of the matching is to meet computations' deadlines, and to minimize energy consumption, respectively, instead of increasing the overall performance. Here in this chapter, I modify the framework and develop a new resource coordination policy, which aims to improve performance, as well as provide prompt response for computations with high priorities.

6.4.1 Resources and Computations

I use resource terms for representation of resources, as in the DREAM framework. An available resource is represented by its density (rate), associated with a time interval during which the resource is available, and a located type, which specifies the type and location of the resource.

Computations are represented by their resource requirements, as in the original framework. However, since the deadline is not a concern in this case, I remove the time related elements (earliest start time and deadline) from the computation requirements, and only keep the quantities of required resources, and their located types. In addition, to model the priority of computation, a new attribute, *urgent*, is added, which is a flag indicating whether

the computation requires prompt response.

When the available resources are matched against the computations' requirements, one objective is to utilize all the available resources at the full capacity, for achieving high performance; another objective is to respond to urgent computations as quickly as possible, once they are discovered.

6.4.2 Resource-Coordination Policy for Dynamic Load-Balancing

In Actor systems, computations are carried out by actors when they process their messages. Processing a message may result in new messages being sent to other actors, which eventually trigger new computations. Therefore, scheduling the processing of actor messages is essential and has great impact on the overall performance. Because in the original DREAM framework, every actor message must go through DREAM Reasoner (DR), which attempts to allocate resources against the message's resource requirements, DR is the key component where we can reschedule actor messages by installing a *dynamic-load-balancing policy*. Figure 6.2 shows the composition of modified DREAM Reasoner.

As shown in Figure 6.2, DREAM Reasoner is a meta-actor, in which the state represents available resources in the system (in the forms of resource terms), and the methods are resource coordination policies, in this case dynamic-load-balancing policy (DLBPolicy). As regular actors, DREAM Reasoner contains a thread which processes received messages from the message queue (MsgQ), which is a FIFO queue. All actor messages in the system are captured by DREAM Reasoner, and released after being processed according to the DLBPolicy.

Under the circumstances where computations' workload is unpredictable and dynamically created, computations which create sub-computations are critical, because they may become a bottleneck in achieving high overall performance. A common reason for processors being under-utilized is that these bottleneck computations are not processed early enough to reveal adequate workload for all processors.

In actor systems, an actor message which contains a *send* operation will eventually trigger new computations at the destination actor of the message. Therefore, they are the potential

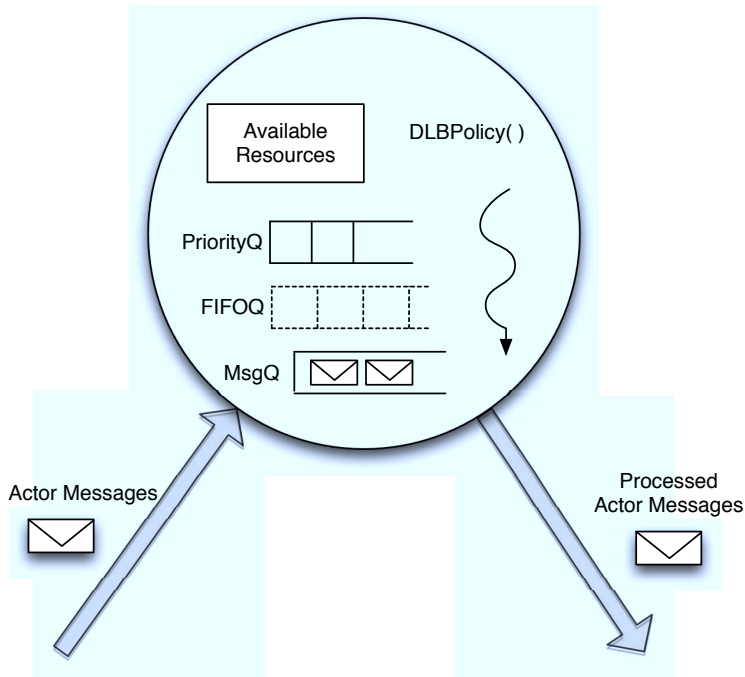


Figure 6.2: DREAM Reasoner

bottleneck messages that need to be processed at higher priority comparing to regular messages which only contain pure computation. In DREAM Reasoner, I use a priority queue (PriorityQ in Figure 6.2) to store these bottleneck messages which are to be released by DR, and the messages are sorted according to their resource requirements – messages which require less resources will be released earlier in order to expose more workload as quickly as possible. Another FIFO queue (FIFOQ in Figure 6.2) is used to store regular computation messages which do not result in more workload. When it is necessary to release messages, messages from the PriorityQ are released first, and then messages in the FIFOQ are released if necessary.

The algorithm of the DLBPolicy in DR is shown in Algorithm 4.

As shown in Algorithm 4, urgent messages, which represent computations requiring prompt reply, are processed and released by DR immediately. Other messages will be placed to one of the waiting queues. If processing the message results in creating more workload, i.e., the message has a *send* operation, the message is placed in the priority queue, which

Algorithm 4 DLBPolicy in DR

```
1: while not all MsgQ, PriorityQ, FIFOQ are empty do
2:   while MsgQ is not empty do /* there are unprocessed messages */
3:     dequeue the first message from MsgQ msg
4:     if msg is an urgent message then
5:       release msg /* msg requires prompt response */
6:       update available resources
7:     else
8:       if msg has send operation then /* processing msg creates new workload */
9:         insert msg in PriorityQ
10:      else
11:        insert msg in FIFOQ
12:      end if
13:    end if
14:    Release Messages /* as shown in Algorithm 5 */
15:  end while
16:  Release Messages /* as shown in Algorithm 5 */
17:  if MsgQ is empty and there are no resources available in next scheduling cycle then
18:    break
19:  end if
20: end while
21: wake up actor computations /* terminate DR cycle */
```

is sorted according to the resource requirements of messages.³ On the other hand, if the message is a pure computation message which does not create new workload, it will be placed in the FIFO queue. The non-urgent messages wait in the queues until they are released by DR. The system schedules DR and worker threads (which are responsible for processing actor messages) in a schedule cycle with predefined length, according to the ratio specified by the tuner. DR only releases enough messages to keep the processors busy for the next scheduling cycle. The algorithm of releasing messages is shown in Algorithm 5. Messages waiting in the priority queue will be released before the messages in the FIFO queue, and messages are released until all the available resources in the next scheduling cycle are allocated.

Algorithm 5 Release Messages

```

1: while there are resources available in next scheduling cycle do
2:   if PriorityQ is not empty then
3:     dequeue the first message priorityMsg
4:     release priorityMsg
5:     update available resources
6:   else
7:     if FIFOQ is not empty then
8:       dequeue the first message regularMsg
9:       release regularMsg
10:      update available resources
11:    else
12:      break
13:    end if
14:  end if
15: end while

```

Holding messages at DR decreases the possibility of resources being under-utilized, because in this way the system can make decisions about which messages to release, in order to generate enough workload to keep all worker threads busy. The bottleneck messages, i.e.,

³Messages which require less resources are processed earlier, in order to reveal more workload as quickly as possible.

messages which create more workload, should be processed as early as possible, because processing those messages results in more workload being created, which can be scheduled on idle processors. Holding messages at DR can help minimize the uncertainty of the workload when a decision has to be made. In addition, holding messages at DR does not affect the overall performance, because enough messages have been released to keep the processors busy until the next scheduling cycle when DR is scheduled again.

6.5 Benchmark Implementation

I have implemented the unbalanced cobwebbed fruit tree (UCFT) benchmark described in Section 6.3, using actors.

6.5.1 Tree Generation

As described in Section 6.3, each node of a UCFT tree contains a computation of predefined size. Since actors are essentially computations, a convenient way of implementing the UCFT tree benchmark is to create a hierarchy of actors according to the parameters specified at creation.

At the time of creating a UCFT tree, three types of parameters need to be specified: tree parameters, computation parameters, and priority parameters. Tree parameters include *size* of the tree, which specifies the number of nodes in the tree, and *shape* of the tree. Note that there are several shapes of the tree, such as *binomial* trees, and *geometric* trees [109]. In these experiments, I use binomial trees, in which each node may have either no children, or m (a predefined number) children. Tree parameters specify the attributes of the tree. Computation parameters include information about the distribution of the computations' sizes. In the implementation, I assume the sizes of computations follow a normal distribution, with predefined mean and standard deviation parameters. Priority parameters specify how many computations have higher priority and require prompt responses, i.e., urgent computations. In the implementation, there is a single priority parameter, which is the *percentage* of urgent computations in the tree.

The trees are generated in the following manner. First, the *root* actor is created, and its

computation size is generated by a number generator following a normal distribution with predefined mean and standard deviation. Then a *CreateTree* message is sent to the root actor, which triggers creation of the tree.

Algorithm 6 Create Tree

```

1: determine whether this actor has children /* based on the tree parameters */
2: if hasChildren then
3:   determine number of children  $m$  /* based on the tree parameters */
4:   for  $i = 1$  to  $m$  do /* create children nodes */
5:     determine size of the computation on node  $i$  /* based on the computation parameters
      */
6:     determine whether node  $i$  is urgent /* based on priority parameters */
7:     create actor  $i$ 
8:     record information of child node  $i$ 
9:     send CreateTree message to actor  $i$  /* tell child  $i$  to create more nodes */
10:  end for
11: end if

```

The algorithm of the *CreateTree* message is shown in Algorithm 6. Upon receiving the message, a node actor first determines whether it has children nodes; if it does, the node actor tries to create children nodes based on the creation parameters. Then the actor records the information about its children, and finally sends the *CreateTree* message to all its child nodes. The creation process terminates when the tree reaches the predefined size.

6.5.2 Traversal

The benchmark application of the UCFT tree is to carry out all the tasks contained in tree nodes while performing a traversal of the tree. The node tasks can be carried out in parallel. However, a parent node must be visited before its children nodes.

To initiate the traversal, a *Traverse* message is sent to the root actor of the tree. The algorithm of the *Traverse* message is shown in Algorithm 7. First, the computation with predefined size is carried out, following which a *Traverse* message is sent to each of the children

nodes, if there are any. Sending *Traverse* message to children nodes triggers computations contained in the children nodes. The traversal completes when all the messages are processed.

Algorithm 7 Traverse

```
1: carry out the computation /* based on the computation size */
2: if hasChildren then
3:   for  $i = 1$  to  $m$  do
4:     send Traverse message to child  $i$  /* traverse child  $i$  */
5:   end for
6: end if
```

Note that in the current implementation, the size of the computation at each node is specified in the form of the computational time it takes to complete the task, which is a dummy function performing certain mathematical calculations.

6.6 Experimental Results

Experiments have been carried out to evaluate the dynamic-load-balancing approach based on resource coordination, using the UCFT benchmark presented in Section 6.3.

6.6.1 Experimental Design

The approach for dynamic load-balancing is implemented by installing the dynamic-load-balancing policy in the resource coordination approach presented in Chapter 3.

For evaluation, I compare the performance of four approaches, including the resource coordination based approach AF-D, ActorFoundry, work sharing [93] and work stealing [21]. The former two are actor systems, and the latter two are traditional approaches for dynamic load-balancing.

I evaluate the approach in two aspects: performance of carrying out the traversal computation for UCFT trees with different creation parameters, and the ability of promptly responding to tasks of high priority.

6.6.2 Baseline Approaches Implementation

Work sharing and work stealing are two existing and commonly-used approaches for dynamic load-balancing. I choose them as baseline approaches for comparison purposes.

I implemented the UCFT tree benchmark in Java, without using actors. The parameters are as same as in the actor tree described in Section 6.5, but each node of tree is a data structure which contains the information about its own computation, as well as its children.

The work sharing and work stealing approaches are implemented using Java threads. In both approaches, the threads (workers) get tree nodes, read the information, and carry out the computations according to the information.

Work Sharing

In the work sharing approach, load balancing is achieved through a globally shared task queue. The algorithm of a worker thread in work sharing approach is shown in Algorithm 8.

Algorithm 8 Worker Thread in Work Sharing

```
1: while true do
2:   if taskQ is not empty then
3:     dequeue the first node myTask from taskQ
4:     if myTask has children then
5:       enqueue all the children nodes
6:     end if
7:     carry out the computation /* according to computation size specified in myTask */
8:   else /* taskQ is empty */
9:     sleep for some time /* wait for other workers to generate more nodes */
10:  end if
11: end while
```

Each worker thread tries to get a node from the task queue. Once a node is dequeued, the worker thread enqueues the children nodes, if there are any, and then carries out the computation specified in the node. Note that if the task queue is empty, the worker thread

will sleep for some time, and then check if other workers have generated more nodes in the queue.

The benchmark computation is initiated by enqueueing the root node of a UCFT tree to the task queue, and creating a predefined number of worker threads. The computation terminates when the task queue is empty and all worker threads are sleeping.

Work Stealing

The worker stealing approach is different from work sharing. In the work stealing approach, there is no global task queue shared by all workers. Instead, each worker has its own task queue, and processes the tasks in the local queue until it is empty. Once completing all the tasks in the local queue, a worker thread tries to steal some tasks from the queues of other workers, and then process these tasks. The algorithm of a worker thread in the work stealing approach is shown in Algorithm 9.

Algorithm 9 Worker Thread in Work Stealing

```
1: while true do
2:   if myQ is not empty then
3:     dequeue the first node myTask from myQ
4:     if myTask has children then
5:       enqueue all the children nodes to myQ
6:     end if
7:     carry out the computation /* according to the computation size specified in node
      myTask */
8:   else /* taskQ is empty */
9:     keep checking other queues and try to steal a node
10:    enqueue the stolen node to myQ
11:  end if
12: end while
```

The benchmark computation is initiated by creating a predefined number of worker threads and their local task queues, and enqueueing the root node of a UCFT tree to the

task queue of one of the workers. The computation terminates when all worker threads are trying to steal tasks from others, and all the task queues are empty.

Discussion

Note that in both work sharing and work stealing approaches (shown in Algorithms 8 and 9), the code for the computations is mixed with the code for load balancing, which increases the complexity of programming. However, the two are separated in the resource coordination based approach: the code for computations is shown in Algorithm 7, and the code for load balancing is shown in Algorithm 4. The fact that computation code is well separated from that of load balancing simplifies the task of programmers, enhances modularity, and supports reusability of the code.

6.6.3 Experimental Results

Experiments were carried out to evaluate the approach of dynamic load balancing using fine-grained resource coordination. I evaluated the approach in two aspects: the effectiveness of dynamic load-balancing, and the ability of promptly responding to computations which require quick responses. I used the UCFT described in Section 6.3 as the benchmark application for evaluation. I compared four approaches, including two actor systems (AF-D and AF), and two traditional dynamic load-balancing approaches (work sharing and work stealing). The experiments were carried out on an Xserve with $2 \times$ Quad-Core Intel Xeon processors (8 cores) @ 2.8GHz, 8GB memory and 12MB L2 cache. The number of worker threads in all approaches were set to be 8. I created binomial trees with parameter $m = 5$; that is to say, each node in the tree either has no children, or 5 children. All four approaches were executed on the same set of UCFT trees for comparison.

The first set of experiments evaluated the performance of the four approaches with increasing average computation size in the tree. In these experiments, the size of the tree is set to be 100, and the standard deviation of computation sizes is 50. The results is shown in Figure 6.3. The computation time for traversing a UCFT tree is linear with respect to the average computation size in the tree, for all four approaches. Comparing to pure Java implementation of the work sharing and work stealing approaches, actor systems have extra

overhead, which is caused by extra workload for supporting actors, such as maintaining the actor platform, handling actor messages, and the scheduler service. However, the overhead appears to be constant when the average computation size increased. In addition, AF-D even outperforms ActorFoundry when the average computation size is larger than 600ms, which shows the resource reasoning performed by AF-D helps balance the workload, and the performance gain exceeds the extra overhead caused by the reasoning. AF-D shows better scalability than ActorFoundry.

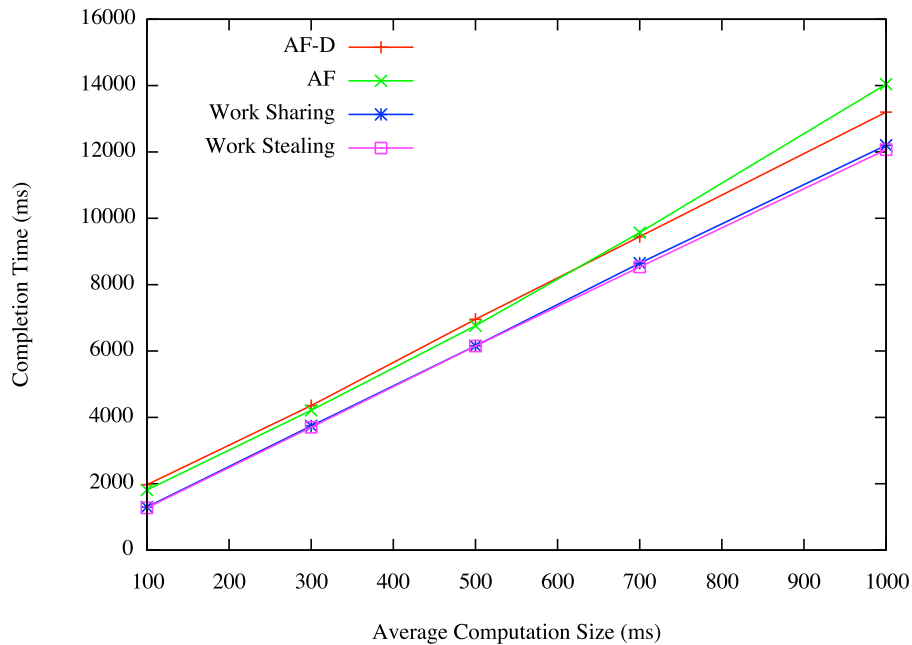


Figure 6.3: Performance Vs. Computation Size (TreeSize=100, StdDev=50)

The second set of experiments investigated the relationship between the overall performance and the standard deviation of the computation sizes in the tree. In these experiments, the size of the tree is set to be 100, and the average computation size in the tree is set to be 500ms. The results are shown in Figure 6.4. The work sharing approach outperforms the work stealing approach when the standard deviation of computation size exceeds 300ms. A large standard deviation indicates that the size of computations in the tree is highly variant. Highly variant computation size presents challenge for load balancing. When bottleneck presents, in the work sharing approach, the worker threads only need to check the global task

queue, while in the work stealing approach, each work threads needs to continuously check all task queues and tries to steal tasks from other threads. Therefore, extra overhead may be present in the work stealing approach.

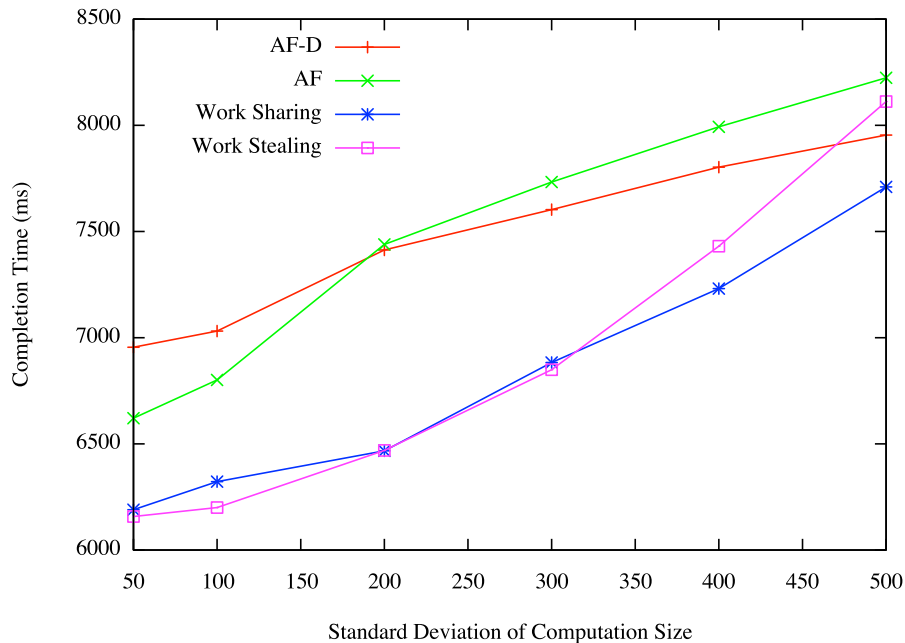


Figure 6.4: Performance Vs. Standard Deviation (TreedSize = 100, AveCompSize = 500)

AF-D outperforms ActorFoundry when the standard deviation exceeds 180ms. More interestingly, when the standard deviation exceeds 470ms, AF-D even outperforms the work stealing approach, which does not provide support for actors, and does not consider urgent computations. When the standard deviation increases, the performance of AF-D comes close to the work sharing approach. Note the work sharing approach is believed to be best suitable for shared memory parallel systems, which is the case in our experiments. These results demonstrate the effectiveness of the resource coordination based approach in providing support for dynamic load-balancing.

The third set of experiments investigated the scalability of the four approaches in terms of the tree size. In these experiments, the average computation size is 500ms, and the standard deviation is 100. The experimental results are shown in Figure 6.5. All four approaches

show similar scalability, and the computation time is linear to the size of the tree. As expected, actor systems have extra insignificant overhead, and AF-D slightly outperforms ActorFoundry for a computation tree whose size is larger than 700 nodes.

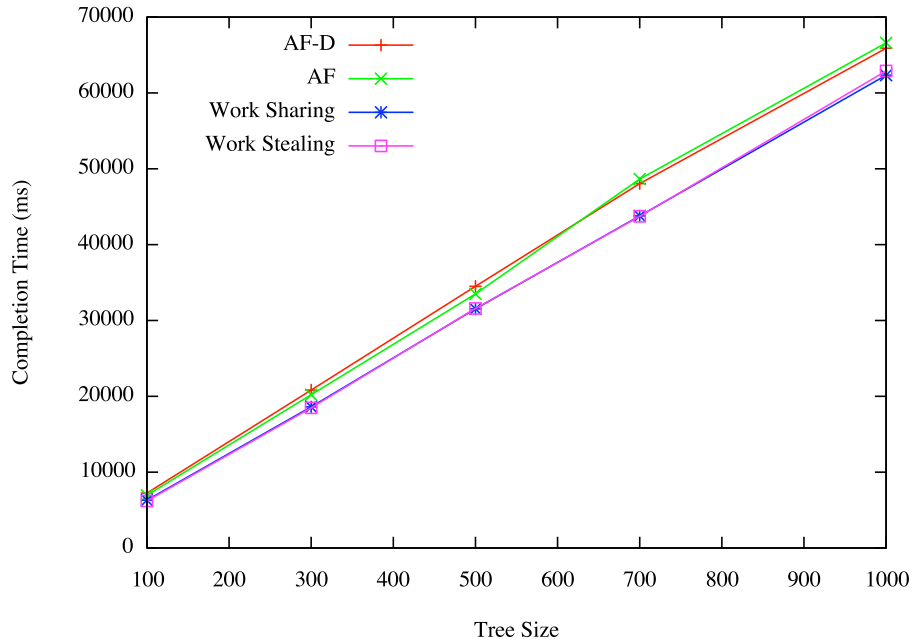


Figure 6.5: Performance Vs. Tree Size (AveCompSize = 500, StdDev = 100)

Besides performance, prompt response for computations with high priority is also critical for certain applications. In the last set of experiments, I evaluated the ability of each system to respond to urgent computations. In these experiments, the tree size is 100, the average computation size is 500ms, with standard deviation of 100. I carried out the experiments for all four approaches, and record the time when they complete the computation at each of the tree node. The experimental results are shown in Figure 6.6. Note that in order to obtain a readable figure, I only plotted the results for two actor approaches, AF-D and AF.

As shown in the figure, in AF-D, all urgent computations are completed earlier than in the ActorFoundry. These experiments indicate that AF-D provides effective support for promptly responding to computations with high priority, while still providing dynamic load balancing.

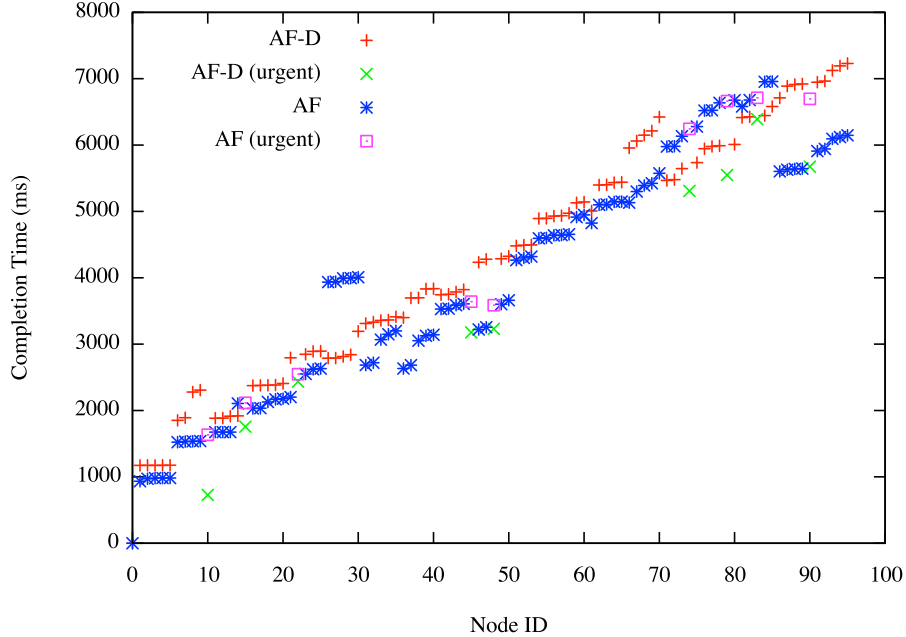


Figure 6.6: Responding to Urgent Computations (TreeSize = 100, AveCompSize = 500, StdDev = 100)

6.7 Summary

Dynamic load-balancing is a key factor in achieving high performance in parallel systems. Computations with unpredictable workload present challenges for dynamic load balancing. Traditional approaches for dynamic load balancing assume that computations can be partitioned into uniform-sized subcomputations, and try to redistribute these subcomputations among processors. However, real world computations often are non-uniform in terms of size, and even partition is rarely feasible. When the sizes of computations are highly variant, traditional approaches do not scale well. In addition, besides performance, responding time is also critical for some types of applications which requires prompt responses. Based on these observations, I develop a benchmark application, unbalanced cobwebbed fruit tree (UCFT), which addresses these challenges. UCFT is developed by adapting the traditional unbalanced tree search (UTS) benchmark, and it enclose substantial computation to each tree node, and adds a priority attribute to the computations. The UCFT benchmark simulates real world

computations to a great extent, and can be used for evaluating dynamic load balancing approaches.

I then present a novel approach for dynamic load balancing based on fine-grained resource reasoning. In this approach, I view the problem from the perspective of resource control, keep track of available resources in the system, and try to allocate the resources in a way that the amount of resources that are expiring is minimized. This approach is implemented by extending ActorFoundry, a Java based framework for actors. Using actors clearly separates the concerns of computations and dynamic load balancing, provides modularity and reusability of the code, and greatly simplifies programmers' tasks. I added a reasoner to ActorFoundry to provide resource reasoning, and installed a dynamic load balancing policy to the reasoner. In addition, I implemented a tuner to control the overhead caused by the reasoning system by providing meta level resource control. I evaluated this approach using the UCFT tree benchmark, and the results show that it can effectively provide dynamic load-balancing, while reserving the flexibility of responding to urgent computations promptly. An interesting observation is that when the size of computations is highly variant, this approach outperforms the work stealing approach, and comes close to the work sharing approach, which is believed to be best suitable for shared memory systems, without providing support for code modularity, reusability, and computation's priorities.

Work is ongoing in a number of directions. First, I am exploring the effectiveness of the approach in parallel systems with distributed memory, where communication presents more challenges, as well as multicore HPC clusters where a mix of shared memory and distributed memory present. Second, I will investigate the scalability of the approach when a larger number of processors/cores are involved. Third, I will explore the possibility of dynamically control the number of worker threads using the tuner, for conserving power. Since worker threads which are not making progress consumes power, a greener solution is possible if the number of workers is dynamically changed as needed, according to the available resources and current workload of the system. I will build on the previous work of power efficient multicores [138], and extend the work for supporting green dynamic load-balancing.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this chapter, I present the conclusion of the thesis in Section 7.1, and future directions of the research in Section 7.2.

7.1 Conclusion

In an open distributed system, computations can be carried out without statically-owned resources, harnessing the collective compute power of the resources connected by the Internet. However, realizing this potential requires efficient and scalable resource discovery, coordination and control, which present challenges in a dynamic, open environment. The three major challenges are: *uncertainty* about resource availability, *overhead* of resource control, and *programming complexity* of resource coordination. In this thesis, I addressed these challenges using a resource-coordination approach.

7.1.1 Resource Coordination Approach

First, the *uncertainty* challenge is addressed by DREAM, a Distributed Resource Estimation and Allocation Model, which is developed for reasoning about resource availability and consumption, and making decisions of matching resources to computations. DREAM defines resources in time and space, and represents distributed computations in terms of the resources they require and the intervals during which they would like to use them. Given the state of resource availability at a point in time, DREAM tells us whether a particular deadline-constrained distributed computation can be accommodated by the system.¹ This

¹I assume that the resource requirements of the computation are provided by the the programmer/user.

can be useful for computations choosing between various courses of action, allowing them to avoid attempting infeasible pursuits.

Second, the challenge of high *overhead* of resource control is addressed using the resource coordination scheme, which integrates DREAM reasoning into ActorFoundry, an optimized implementation of the Actor model. The resource related decision making is carried out by a reasoner alongside the actual computations, and the resource allocation decisions are enforced by properly scheduling sub-computations. The overhead caused by the decision making process is controlled by a meta-component, tuner, which dynamically adjusts the resource division between the reasoner and the computations.

Third, the *programming complexity* challenge is addressed by the way the resource coordination is implemented. In particular, DREAM abstracts away the functionality of computations, and represents computations by their resource requirements. This type of abstraction enables resource coordination to be implemented separately from the computations. In this approach, DREAM Reasoner is the resource coordination component, while computations are carried out by regular actors. Separating resource coordination concern from that of computations greatly simplifies the programming complexity of resource coordination, and also enhances code readability, modularity, and reusability. This is also illustrated in the case studies.

7.1.2 Case Studies

The effectiveness and efficiency of the resource coordination approach were evaluated using three different types of applications, as the case studies presented in the thesis.

In the first case study, the resource coordination approach is used for providing QoS support. This is achieved by prioritizing individual message deliveries and method executions involved in a distributed computation, based on the calculated deadlines by which each must be completed. Fundamental to this approach is an identification of the proper granularity of control, efficient calculation of fine-grained deadlines, and effectively enforcement of those deadlines. I identified that message-level deadlines are effective in providing QoS support. Although the calculation of fine-grained deadlines for actor messages is non-trivial for a computation with sufficiently complex interaction between actors, for classes of computations

which use the pipeline communication style, this can be efficiently achieved. Experimental results from a live video conferencing application illustrated the effectiveness of this approach.

In the second case study, the resource coordination approach is adapted for supporting power-efficient multicore execution of actor computations. The DREAM model is adapted to be an energy-aware resource management model, DREAM-MCP, for providing a flexible way to analyze energy consumption of multicores operating at non-uniform frequencies. This information can then be used to generate a energy-efficient schedule for execution of the computations – as well as a schedule of frequency changes on a per-core basis – while satisfying performance requirements of computations. Since existing approaches for analyzing multicore power consumption assume all cores must operate the the same frequency, our approach is particular promising in fully exploiting the potential of the newly emerging hardware technologies, which enable per-core frequencies.

In the third case study, the resource coordination approach is used for providing dynamic load balancing support. This is achieved by reasoning about resource utilization, generating computation schedules which can utilize CPU resources at their full capacity, and controlling resources at a fine grain. This approach provides more flexibility than the existing dynamic load balancing approaches because of two reasons. First, it does not assume computations being redistributed have the same size, as the case in the existing approaches. Second, it provides the flexibility of dealing computations of different priorities. A new benchmark application, unbalanced cobwebbed fruit tree (UCFT), is developed to simulate real world computations. This benchmark was used to evaluate our approach, and the experimental results illustrated that the resource-coordination approach is effective and shows better scalability when the size of the computation is highly variant. Besides, it offers better programming complexity for coding the both the coordination and the computations.

These case studies also illustrated the flexibility of the resource coordination approach, which can be adapted for providing support for different types of coordination. For example, dynamic load balancing requires computations being executed as *fast* as possible, power and energy concerns require computations being executed as *slow* as possible (as long as the performance requirements are satisfied), and the QoS requires computations being executed at an even speed, which demands precise resource delivery at the right time. It has

been shown in this thesis that although sometimes constrained by certain limitations, our resource coordination approach can satisfactorily provide effective support for these different types of coordination, and the overhead caused by the coordination is relatively modest and adjustable. In addition, the adaptations of the resource coordination do not add extra programming complexity to computations.

7.2 Future Directions

Work is ongoing in a number of directions, for enhancing and improving the approach, as well as exploring more applications. Potential future work is presented in the following sections.

7.2.1 Multi-Dimension Tuning

The tuner provides meta level resource control by dynamically balancing the use of resources between reasoning and carrying out the computations. In the current implementation, this is achieved by adjusting the resource allocation ratio between the reasoning component and the computations. I am exploring different dimensions along which the tuner might try to balance the resource use.

One promising opportunity appears to lie in controlling how far in the future the reasoning mechanism looks in search of needed resources. Since the complexity of matching resources to computations is linearly related to the number of resource terms, the size of the time frame in which resources are searched is a key factor. Tuning the size of the time frame can adjust the overhead caused by the reasoning mechanism.

7.2.2 Smart Space: Spatial-Temporal Coordination

Besides temporal constraints such as deadlines, computations often have *spatial constraints*, which can be represented as spatial relationships between computations. For example, it is more efficient for a computation to be collocated with its data source at the stage of taking input data, and collocated with collectors of results when it finishes the computation. A data backup application should store the copy of data at a different location from its original.

To support separation of concerns [35], dealing with spatial constraints should not be mixed with computations. In particular, programmers only need to program functionality of the computations and specify their temporal and spatial constraints, and satisfying those constraints is taken care of by a *middleware*.

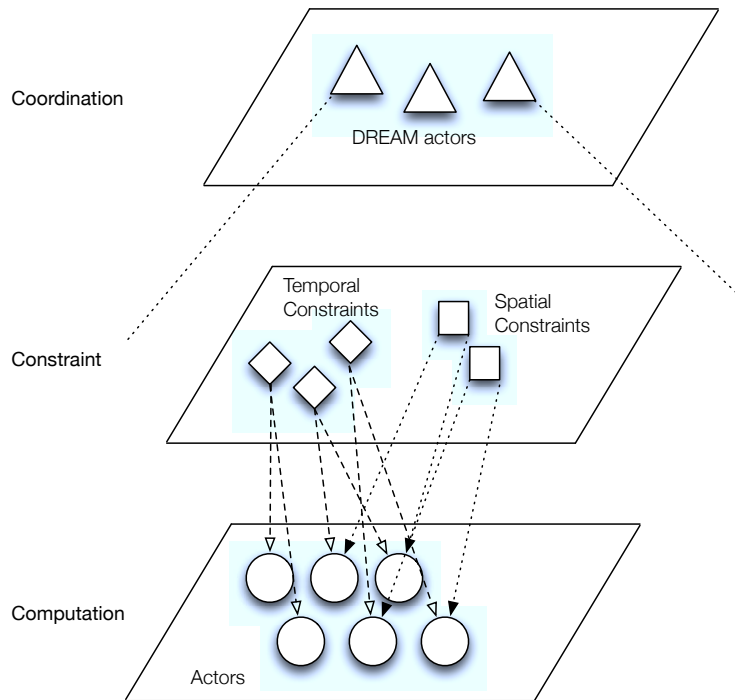


Figure 7.1: Smart Space

I propose such a middleware, namely a smart space, as a layered approach for coordinating both temporal and spatial constraints of computations. As shown in Figure 7.1, the lowest layer is computation layer, which contains actors (computations). Both temporal and spatial constraints of computations are specified at the second layer, constraint layer. The third layer, coordination layer is responsible for scheduling the computations in a way that both temporal and spatial constraints are satisfied. Coordination layer consists of a number of DREAM reasoning actors, which attempt to satisfy those constraints by coordinating resource use. Separating what the constraints are and how they should be satisfied from the computations enhances modularity and reusability, simplifies the programmers' task, and supports a clear design.

7.2.3 Phone Grid

The population of mobile device holders is growing rapidly. As shown in *Global Mobile Statistics* [101], there are 5.3 billion mobile subscriptions as of the end of 2010. In addition, mobile devices sales rose in 2010, while smartphone sales showed the strongest growth (74.4% of annual sales growth). These mobile devices form a highly dynamic network of distributed resources. Given the fact that usually mobile devices are not used all the time, they create opportunities for computations making use of the idle resources. However, the dynamicity of the network presents challenges.

Inspired by SETI@HOME [117], a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence, I propose a Phone Grid approach, which can synthesize distributed resources of mobile devices to carry out computations. By subscribing to Phone Grid, mobile device holders make their resources available on the network, and receive rewards which entitle them to use resources on the grid later, or to free air time. The DREAM-based resource coordination scheme can be installed to effectively schedule computations on the grid.

7.2.4 Power-Efficient HPC Clusters

In Chapter 6, I adapted the coordination approach for providing dynamic load balancing support. The work is implemented on a shared memory multicore system. I am extending the work to a HPC cluster, which consists of a number of multicore computers connected by a high speed network. Since an HPC cluster has a mix of shared memory and distributed memory, communication presents challenges for load balancing.

Power consumption is a major concern in the current HPC clusters. I propose to address this concern in two levels. First, on each processor, a fine-grained per-core frequency control can be applied, for determining frequencies on a per-core basis, as described in Chapter 5. Second, on the whole system, a coarse-grained processor control can turn off/on individual processors according to the power consumption and performance needs. The tuner can be used for balancing performance against power consumption by tuning the number of active processors.

7.3 Concluding Remarks

In a resource constrained world, fine-grained control on resources is becoming necessary and critical. In this thesis, I present a resource coordination approach, which enables fine-grained resource control with manageable overhead. The approach is implemented in a way that resource coordination concerns are separated from computational concerns, which enables reusability of the approach. I have shown that the approach can be easily adapted for supporting different types of coordination requirements, without adding extra programming complexity to the computations. I am following up the work in both directions of enhancing the approach and applying it to more applications.

REFERENCES

- [1] T. Agerwala and S. Chatterjee. Computer Architecture: Challenges and Opportunities for the Next Decade. *IEEE Micro*, 25:58–69, 2005.
- [2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [3] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, 1997.
- [4] R. Al-Ali, K. Amin, G. von Laszewski, O. Rana, and D. Walker. An OGSA-Based Quality of Service Framework. In M. Li, X.-H. Sun, Q. Deng, and J. Ni, editors, *Grid and Cooperative Computing*, volume 3033 of *Lecture Notes in Computer Science*, pages 529–540. 2004.
- [5] A. Albore, N. Alechina, P. Bertoli, C. Ghidini, and B. Logan. Bounded-Resource Reasoning as (Strong or Classical) Planning. In M. Fisher, F. Sadri, and M. Thielscher, editors, *Computational Logic in Multi-Agent Systems*, pages 77–96. 2009.
- [6] N. Alechina and B. Logan. A Logic of Situated Resource-Bounded Agents. *Journal of Logic, Language and Information*, 18:79–95, 2009.
- [7] N. Alechina, B. Logan, N. H. Nga, and A. Rakib. Verifying Time, Memory and Communication Bounds in Systems of Reasoning Agents. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, pages 736–743, 2008.
- [8] N. Alechina, B. Logan, and M. Whitsey. A Complete and Decidable Logic for Resource-Bounded Agents. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 606–613, 2004.
- [9] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [10] AMD Corporation. AMD BIOS and Kernel Developers Guide (BKDG) for AMD Family 10h Processors, 2009. <http://developer.amd.com/documentation/guides/>.
- [11] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS)*, pages 483–485, 1967.

- [12] B. An, V. Lesser, D. Irwin, and M. Zink. Automated Negotiation with Decommitment for Dynamic Resource Allocation in Cloud Computing. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 981–988, 2010.
- [13] G. Andrews, D. Dobkin, and P. Downey. Distributed Allocation with Pools of Servers. In *Proceedings of the 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 73–83, 1982.
- [14] A. Appel and T. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 293–302, 1989.
- [15] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- [16] M. Ashley. The Actor Foundry: A Java-based Actor Programming Environment. Technical report, Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998.
- [17] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, and S. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on SuperComputing*, pages 158–165, 1991.
- [18] D. Batista and N. da Fonseca. A Survey of Self-Adaptive Grids. *IEEE Communications Magazine*, 48(7):94–100, 2010.
- [19] W. Binder, J. Hulaas, and A. Villazon. Portable Resource Control in Java: The J-SEAL2 Approach. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, 2001.
- [20] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1995)*, pages 207–216, 1995.
- [21] R. Blumofe and C. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- [22] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.
- [23] A. Cesta and C. Stella. A Time and Resource Problem for Planning Architectures. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning*, pages 117–129, 1997.

- [24] K. Chakraborty. A Case for an Over-Provisioned Multicore System: Energy Efficient Processing of Multithreaded Programs. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, 2007.
- [25] F. Chang, J. Ren, and R. Viswanathan. Optimal Resource Allocation in Clouds. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD 2010)*, pages 418–425, 2010.
- [26] K. Chard, K. Bubendorfer, and P. Komisarczuk. High Occupancy Resource Allocation for Grid and Cloud Systems, a Study with DRIVE. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 73–84, 2010.
- [27] S. Cho and R. Melhem. Corollaries to Amdahl’s Law for Energy. *Computer Architecture Letters*, 7(1):25–28, 2008.
- [28] W. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, 1981.
- [29] R. Cox. Probability, Frequency and Reasonable Expectation. In G. Shafer and J. Pearl, editors, *Readings in Uncertain Reasoning*, pages 353–365. Morgan Kaufmann Publishers Inc., 1990.
- [30] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008)*, 2008.
- [31] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A Resource Management API for Java Platform. *Software: Practice and Experience*, 35(2):123–157, 2005.
- [32] G. Czajkowski and T. von Eichen. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998)*, pages 21–35, 1998.
- [33] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proceedings of IFIP Congress*, pages 807–813, 1974.
- [34] E. Dijkstra. Cooperating Sequential Processes. *Programming Languages*, pages 43–112, 1968.
- [35] E. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. 1982.
- [36] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

- [37] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2007)*, pages 1–8, 2007.
- [38] J. Drapkin and D. Perlis. A Preliminary Excursion into Step-Logics. In *Proceedings of the ACM SIGART International Symposium on Methodologies for Intelligent Systems*, pages 262–269, 1986.
- [39] D. Eager, E. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. In *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–3, 1985.
- [40] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.
- [41] A. El-kholy and B. Richards. Temporal and Resource Reasoning in Planning: the ParcPLAN Approach. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996)*, pages 614–618, 1996.
- [42] J. Elgot-Drapkin, M. Miller, and D. Perlis. Memory, Reason and Time: the Step-Logic Approach. *Philosophy and AI: Essays at the Interface*, pages 79–103, 1991.
- [43] J. Elgot-Drapkin and D. Perlis. Reasoning Situated in Time i: Basic Concepts. *Journal of Experimental & Theoretical Artificial Intelligence*, 2:75–98, 1990.
- [44] B. Esbaugh and A. Sodan. Coarse-Grain Time Slicing with Resource-Share Control in Parallel-Job Scheduling. In *Proceedings of the 3rd International Conference on High Performance Computing and Communications (HPCC 2007)*, pages 30–43, 2007.
- [45] D. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. 1995.
- [46] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
- [47] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 238–261, 1997.
- [48] M. Fisher and C. Ghidini. Exploring the Future with Resource-Bounded Agents. *Journal of Logic, Language and Information*, 18(1):3–21, 2009.
- [49] G. Fortino and L. Nigro. Modeling, Analysis and Implementation of Actor-Based Multimedia Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPTA 1999)*, pages 489–495, 1999.

- [50] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-allocation. In *Proceedings of the 7th International Workshop on Quality of Service (IWQoS 1999)*, pages 27–36, 1999.
- [51] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture That Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service (IWQoS 2000)*, pages 181–188, 2000.
- [52] S. Ghosh, R. R. Rajkumar, J. Hansen, and J. Lehoczky. Scalable Resource Allocation for Multi-Processor QoS Optimization. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 174–183, 2003.
- [53] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.
- [54] M. Gothe, D. Wengelin, and L. Asplund. The Distributed ADA Run-time System DARTS. *Software - Practice and Experience*, 21(11):1249–1263, 1991.
- [55] J. Grant, S. Kraus, and D. Perlis. A Logic for Characterizing Multiple Bounded Agents. *Autonomous Agents and Multiagent Systems*, 3(4):351–387, 2000.
- [56] I. Greif. Semantics OF Communicating Parallel Processes. Technical report, Massachusetts Institute of Technology, 1975.
- [57] F. Guim, I. Rodero, and J. Corbalán. The Resource Usage Aware Backfilling. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 59–79. 2009.
- [58] P. Haller and M. Odersky. Actors that Unify Threads and Events. In A. Murphy and J. Vitek, editors, *Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. 2007.
- [59] C. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. PhD thesis, Massachusetts Institute of Technology, 1971.
- [60] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
- [61] M. Hill and M. Marty. Amdahl’s Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.
- [62] S. Hofmeyr, C. Iancu, and F. Blagojević. Load Balancing on Speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 147–158, 2010.
- [63] E. Horvitz. Reasoning about Beliefs and Actions Under Computational Resource Constraints. *International Journal of Approximate Reasoning*, 2(3):337–338, 1988.

- [64] E. Horvitz, D. Heckerman, and C. Langlotz. A Framework for Comparing Alternative Formalisms for Plausible Reasoning. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 210–214, 1986.
- [65] S. Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 27–40. 1996.
- [66] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy Project. Technical report, University of California, Berkeley, 2003.
- [67] O. Ibarra and C. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM (JACM)*, 24:280–289, 1977.
- [68] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2006)*, pages 347–358, 2006.
- [69] N. Jamali. *CyberOrgs: A Model for Resource Bounded Complex Agents*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.
- [70] N. Jamali and C. Liu. Reifying Control of Multi-Owned Network Resources. In *Proceedings of the 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 1–8, 2007.
- [71] N. Jamali, P. Thati, and G. Agha. An Actor-based Architecture for Customizing and Controlling Agent Ensembles. *IEEE Intelligent Systems*, 14(2):38–44, 1999.
- [72] N. Jamali and X. Zhao. A Scalable Approach to Multi-Agent Resource Acquisition and Control. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 868–875, 2005.
- [73] N. Jamali and X. Zhao. Self-Adapting Resource Bounded Distributed Computations. In *Proceedings of the 1st IEEE International Conference on Self-Adapting and Self-Organizing Systems*, pages 311–314, 2007.
- [74] M.-W. Jang. The Actor Architecture Manual. Technical report, Open Systems Laboratory, University of Illinois at Urbana-Champaign, 2004.
- [75] D. Kafura. ACT++: Building a Concurrent C++ with Actors. *Journal of Object-Oriented Programming*, 3:25–37, 1990.
- [76] K. Kant. Toward a Science of Power Management. *IEEE Computer*, 42:99–101, 2009.
- [77] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (PPPJ 2009)*, 2009.

- [78] G. E. Kersten, W. Michalowski, S. Szpakowicz, and Z. Koperczak. Restructurable Representations of Negotiation. *Management Science*, 37(10):1269–1290, 1991.
- [79] W. Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
- [80] W. Kim, D. Brooks, and G.-Y. Wei. A Fully-Integrated 3-Level DC/DC Converter for Nanosecond-Scale DVS with Fast Shunt Regulation. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC 2011)*, 2011.
- [81] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. Enabling OnChip Switching Regulators for Multi-Core Processors Using Current Staggering. In *Proceedings of the Workshop on Architectural Support for Gigascale Integration*, 2007.
- [82] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 123–134, 2008.
- [83] W. Kornfeld and C. Hewitt. The Scientific Community Metaphor. In A. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 311–320. 1988.
- [84] V. Korthikanti and G. Agha. Analysis of Parallel Algorithms for Energy Conservation in Scalable Multicore Architectures. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP 2009)*, pages 212–219, 2009.
- [85] V. Korthikanti and G. Agha. Avoiding Energy Wastage in Parallel Applications. In *Proceedings of the International Conference on Green Computing*, pages 149–163, 2010.
- [86] V. Korthikanti and G. Agha. Energy-Performance Trade-off Analysis of Parallel Algorithms. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HotPar 2010)*, 2010.
- [87] P. Krueger, T. Lai, and V. Dixit-Radiya. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5:488–497, 1994.
- [88] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.
- [89] P. Lee and G. Necula. Research on Proof-Carrying Code for Mobile-Code Security. In *Proceedings of the Workshop on Foundations of Mobile Code Security*, 1997.
- [90] W. Leinberger, G. Karypis, V. Kumar, and R. Biswas. Load Balancing Across Near-Homogeneous Multi-Resource Servers. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 60–72, 2000.
- [91] J. Li and J. F. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2:397–422, 2005.

- [92] D. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.
- [93] H.-C. Lin and C. Raghavendra. A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC). *IEEE Transactions on Software Engineering*, 18(2):148–158, 1992.
- [94] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [95] C. Manning. *ACORE: The Design of a Core Actor Language and Its Compiler*. PhD thesis, Massachusetts Institute of Technology, 1987.
- [96] R. McAfee and J. McMillan. Auctions and Bidding. *Journal of Economic Literature*, 25:699–738, 1987.
- [97] P. Messina. The Concurrent Supercomputing Consortium: Year 1. *IEEE Parallel Distributed Technology*, 1:9–16, 1993.
- [98] Microsoft Corporation. Axum Programming Language, 2008.
- [99] R. Milner. *Communication and Concurrency*. Prentics Hall, 1989.
- [100] R. Milner. *Communicating and Mobile Systems: The π -Calculus*. Cambridge University Press, 1999.
- [101] Global Mobile Statistics, 2011. <http://mobithinking.com/>.
- [102] L. Moreau and C. Queinnec. Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing. In *Proceedings of the Usenix Conference on Domain-Specific Languages (DSL 1997)*, pages 183–197, 1997.
- [103] L. Moreau and C. Queinnec. Distributed and Multi-Type Resource Management. In *Proceedings of the ECOOP 2002 Workshop on Object-Oriented Technology*, 2002.
- [104] R. Motwani and P. Raghavan. Randomized Algorithms. *ACM Computing Surveys*, 28(1):33–37, 1996.
- [105] R. Mukhopadhyay, D. Ghosh, and N. Mukherjee. A Study on the Application of Existing Load Balancing Algorithms for Large, Dynamic, Heterogeneous Distributed Systems. In *Proceedings of the 9th International Conference on Software Engineering, Parallel and Distributed Systems*, pages 238–243, 2010.
- [106] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and Thermal Management in the Intel Core Duo Processor. *Intel Technology Journal*, 10(2):109–122, 2006.
- [107] J. Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

- [108] M. Nirkhe, S. Kraus, and D. Perlis. Thinking Takes Time: a Modal Active-Logic for Reasoning in Time. Technical report, University of Maryland at College Park, Report No. UMIACS-TR-94-39, 1994.
- [109] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In G. Almsi, C. Cascaval, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. 2007.
- [110] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP 2008)*, pages 123–131, 2008.
- [111] D. Pattou and B. Maigret. MOL3D, A Modular and Interactive Program for Molecular Modeling and Conformational Analysis: I – Basic Modules. *J. Mol. Graph.*, 6(2):112–121, 1988.
- [112] J. Pearl. Fusion, Propagation, and Structuring in Belief Networks. *Journal of Artificial Intelligence*, 29:241–288, 1986.
- [113] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kalé, and K. Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.
- [114] J.-N. Quintin and F. Wagner. Hierarchical Work-Stealing. In *Proceedings of the 16th International European Conference on Parallel and Distributed Computing (EuroPar 2010): Part I*, pages 217–229, 2010.
- [115] K. Ravichandran, S. Lee, and S. Pande. Work Stealing for Multi-Core HPC Clusters. In *Proceedings of the 17th International European Conference on Parallel and Distributed Computing (EuroPar 2011): Part I*, pages 205–217, 2011.
- [116] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *Internet Engineering Task Force (IETF) RFC1889*, 1996.
- [117] SETI@HOME. <http://setiathome.berkeley.edu/>.
- [118] L. Shi. A New Algorithm for Stochastic Discrete Resource Allocation Optimization. *Discrete Event Dynamic Systems*, 10(3):271–294, 2000.
- [119] SPEC CPU 2000 Benchmarks. <http://www.spec.org/benchmarks.html>.
- [120] S. Srinivasan. A Thread of One’s Own. In *Proceedings of the Workshop on New Horizons in Compilers (NHC 2006)*, pages 1–11, 2006.
- [121] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, pages 104–128, 2008.

- [122] B. Srivastava. Realplan: Decoupling Causal and Resource Reasoning in Planning. In *Proceedings of the 17th National Conference on Artificial Intelligence and the 12th Conference on Innovative Applications of Artificial Intelligence*, pages 812–818, 2000.
- [123] D. Sturman and G. Agha. A Protocol Description Language for Customizing Failure Semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 148–157, 1994.
- [124] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.
- [125] I. Takeuchi. Towards an Integrated Earthquake Disaster Simulation System. In *Proceedings of the 1st International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster*, 2003.
- [126] The E Language, 2000. <http://www.erights.org/elang>.
- [127] Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. White paper, Intel, 2008. <http://www.intel.com/technology/turboboost/>.
- [128] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36:20–34, 2001.
- [129] N. Venkatasubramanian and G. Agha. An Actor Based Framework for Managing Multimedia QoS. In *Proceedings of the 3rd International Workshop on Multimedia Information Systems*, pages 54–62, 1999.
- [130] X. Wang and S. Zivarras. Performance-Energy Tradeoffs for Matrix Multiplication on FPGA-Based Mixed-Mode Chip Multiprocessors. In *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED 2007)*, pages 386–391, 2007.
- [131] Y.-T. Wang and R. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, 1985.
- [132] A. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proceedings of the International Symposium on Parallel Processing*, pages 542–546, 1998.
- [133] J. White. Telescript Technology: The Foundation for the Electronic Marketplace. *Technical report, General Magic Inc., Mountainview, CA*, 1994.
- [134] M. Wooldridge. Computationally Grounded Theories of Agency. In *Proceedings of the 4th International Conference on MultiAgent Systems (ICMAS 2000)*, pages 13–20, 2000.
- [135] X. Zeng and A. Sodan. Job Scheduling with Lookahead Group Matchmaking for Time/Space Sharing on Multi-Core Parallel Machines. In *Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2009)*, pages 232–258, 2009.

- [136] X. Zhang, K. Shen, S. Dwarkadas, and R. Zhong. An Evaluation of Per-Chip Nonuniform Frequency Scaling on Multicores. In *Proceedings of the USENIX Annual Technical Conference*, pages 19–24, 2010.
- [137] X. Zhao and N. Jamali. Temporal Reasoning about Resources for Deadline Assurance in Distributed Systems. In *Proceedings of ICDCS Workshop on Assurance in Distributed Systems and Networks (ADSN 2010)*, 2010.
- [138] X. Zhao and N. Jamali. Fine Grained Per-Core Frequency Scheduling for Power Efficient Multicore Execution. In *Proceedings of the 2nd IEEE International Green Computing Conference (IGCC 2011) [Work-in-Progress Workshop]*, pages 1–8, 2011.
- [139] X. Zhao and N. Jamali. Supporting Deadline Constrained Distributed Computations on Grids. In *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (Grid 2011)*, pages 165–172, 2011.
- [140] G. Zheng, E. Meneses, A. Bhatel, and L. Kal. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW 2010)*, pages 436–444, 2010.
- [141] M. Zhu and S. Martínez. On Distributed Optimization under Inequality Constraints via Lagrangian Primal-Dual Methods. In *Proceedings of the 2010 American Control Conference*, pages 2434–2439, 2010.

APPENDIX A

RAW DATA FROM CASE STUDY 1

Raw data from the experiments in case study 1, QoS support, is presented in this chapter.

Table A.1 shows the raw data from one set of the ThreadRing benchmark experiments. These results have been shown in Figure 4.1.

Table A.1: Raw Data from the ThreadRing Experiments

Computation ID	Completion Time (ms)	
	AF-D	AF
1	397	2567
2	699	2340
3	942	2456
4	1210	1846
5	1456	2581
6	1715	2591
7	1961	2593
8	2213	2595
9	2459	2598
10	2698	2606

Table A.2 shows the number of tuning events triggered in the AF_D experiment.

Table A.2: Tuning Events Triggering from the AF_D Experiment

Events	Reasoning too fast	Reasoning too slow	Reasoning too costly
Number	5	2	0

Table A.3 shows the raw data from one set of the live video conferencing experiments. Note packet IDs starting with a “v” represent video packets, and those starting with an “a” represent audio packets. These results have been shown in Figure 4.3.

Table A.4 shows the number of tuning events triggered in the AF_D experiment.

Table A.3: Raw Data from the Live Video Conferencing

Packet ID	Completion Time (ms)	
	AF-D	AF
v0	70	47
a0	158	1014
v10	226	156
a1	273	515
v20	382	359
a2	397	265
v30	538	390
a3	522	640
v40	694	624
a4	772	390
v50	850	764
a5	897	889
v60	1066	936
a6	1021	1139
v70	1162	1170
a7	1146	1264
v80	1318	1326
a8	1271	764
v90	1472	1420
a9	1387	1398

Table A.4: Tuning Events Triggering from the AF_D Experiment

Events	Reasoning too fast	Reasoning too slow	Reasoning too costly
Number	3	1	0

APPENDIX B

RAW DATA FROM CASE STUDY 2

Raw data from the experiments in case study 2, power-efficiency support, is presented in this chapter.

Table B.1 shows the raw data from the N-Body Experiments. These results have been shown in Figure 5.3. The experiments are carried out for 15 times, and mean and standard deviation are shown in Table B.1.

Table B.1: Raw Data from the N-Body Experiments

System	Sequential portion (ms)		Parallel portion (ms)	
	Mean	StdDev	Mean	StdDev
DREAM-MCP	68	3.7	83	3.2
AF	55	2.5	83	4.1

Table B.2 shows the number of tuning events triggered in the DREAM_MCP experiments.

Table B.2: Tuning Events Triggering from the N-Body Experiments

Events	Reasoning too fast	Reasoning too slow	Reasoning too costly
Mean	1.3	0	0
StdDev	0.46	0	0

APPENDIX C

RAW DATA FROM CASE STUDY 3

Raw data from the experiments in case study 3, dynamic load balancing support, is presented in this chapter.

Table C.1 shows the raw data from the UCFT Experiments, which investigated the performance of the four dynamic load-balancing approaches vs. the computation size. For each parameter setting, I ran the experiment for 15 times, for each of the approaches. The mean and standard deviation are shown in Table C.1. These results have been shown in Figure 6.3.

Table C.1: Raw Data from the UCFT Experiments (Performance Vs. Computation Size; TreeSize=100, StdDev=50)

Computation Size	AF-D		AF		WSH		WST	
	Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev
100	1970	12.1	1814	10.9	1297	21.5	1274	44.2
300	4354	20.4	4211	19.1	3743	35.2	3700	36.2
500	6955	29.9	6763	30.1	6155	35.3	6151	40.2
700	9449	49.1	9570	39.1	8648	41.1	8533	52.1
1000	13200	49.3	14041	52.1	12199	67.1	12070	65.1

Table C.2 shows the number of tuning events triggered in the AF_D experiments.

Table C.2: Tuning Events Triggering from the UCFT Experiments

Computation Size	Reasoning too fast		Reasoning too slow		Reasoning too costly	
	Mean	StdDev	Mean	StdDev	Mean	StdDev
100	15.9	4.9	7.8	4.3	0	0
300	17.9	3.5	5.2	3.9	0	0
500	28.3	8.1	11.8	6.3	0	0
700	44.9	11.4	27.8	9.3	0	0
1000	48.7	19.9	33.5	14.5	0	0

Table C.3 shows the raw data from the UCFT Experiments, which investigated the performance of the four dynamic load-balancing approaches vs. the standard deviation of the computation size. For each parameter setting, I ran the experiment for 15 times, for each of the approaches. The mean and standard deviation are shown in Table C.3. These results have been shown in Figure 6.4.

Table C.4 shows the number of tuning events triggered in the AF_D experiments.

Table C.3: Raw Data from the UCFT Experiments (Performance Vs. Standard Deviation; TreedSize = 100, AveCompSize = 500)

Standard Deviation	AF-D		AF		WSH		WST	
	Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev
50	6955	29.9	6763	30.1	6155	35.3	6151	40.2
100	7031	34.5	6801	38.9	6322	51.3	6200	41.2
200	7413	38.1	7439	34.5	6467	43.4	6469	44.9
300	7603	29.9	7733	34.1	6883	44.3	6849	46.9
400	7803	43.5	7993	44.5	7231	45.0	7431	47.9
500	7954	39.0	8224	49.2	7710	52.4	8112	54.8

Table C.4: Tuning Events Triggering from the UCFT Experiments

Standard Deviation	Reasoning too fast		Reasoning too slow		Reasoning too costly	
	Mean	StdDev	Mean	StdDev	Mean	StdDev
50	28.3	8.1	11.8	6.3	0	0
100	27.8	6.5	11.2	6.6	0	0
200	28.4	9.2	14.5	5.9	0	0
300	34.9	12.4	27.8	9.3	0	0
400	38.7	15.3	23.5	9.5	0	0
500	35.6	18.5	20.4	10.5	0	0

Table C.5 shows the raw data from the UCFT Experiments, which investigated the performance of the four dynamic load-balancing approaches vs. the size of the UCFT tree. For each parameter setting, I ran the experiment for 5 times, for each of the approaches. The mean and standard deviation are shown in Table C.5. These results have been shown in Figure 6.5.

Table C.5: Raw Data from the UCFT Experiments (Performance Vs. Tree Size; AveCompSize = 500, StdDev = 100)

Tree Size	AF-D		AF		WSH		WST	
	Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev
100	7031	34.5	6801	38.9	6322	51.3	6200	41.2
300	20822	58.6	20211	80.1	18610	83.4	18478	89.0
500	34507	123.1	33500	178.4	31534	161.2	31579	189.1
700	48051	235.4	48645	277.1	43788	401.4	43733	479.3
1000	65899	427.4	66600	480.9	62340	598.6	62898	616.2

Table C.6 shows the number of tuning events triggered in the AF_D experiments.

Table C.6: Tuning Events Triggering from the UCFT Experiments

Tree Size	Reasoning too fast		Reasoning too slow		Reasoning too costly	
	Mean	StdDev	Mean	StdDev	Mean	StdDev
100	27.8	6.5	11.2	6.6	0	0
300	17.9	3.5	15.2	5.5	0	0
500	25.3	8.1	17.8	8.3	0	0
700	39.9	16.4	27.8	9.3	0	0
1000	50.7	15.7	33.5	14.5	0	0

Table C.7 shows the raw data from one set of the UCFT Experiments, which investigated the ability of responding to urgent nodes. These results have been shown in Figure 6.6.

Table C.7: Raw Data from the UCFT Experiments (Responding to Urgent Computations; TreeSize = 100, AveCompSize = 500, StdDev = 100)

Urgent Node ID	Completion	Time (ms)		
	AF-D	AF	WSH	WST
10	728	1635	1081	750
15	1757	2116	1771	1812
22	2435	2547	2651	2618
45	3178	3641	3049	4349
48	3231	3587	3493	3964
74	5309	6245	5358	5345
79	5549	6663	5630	5819
83	6391	6712	6290	6136
90	5675	6697	6285	6185