Effective cooperative scheduling of task-parallel applications on multiprogrammed parallel architectures

GEORGIOS VARISTEAS

Doctoral Thesis in Information and Communication Technology
Royal institute of Technology, KTH
Stockholm, Sweden, 2015
Abstract

Emerging architecture designs include tens of processing cores on a single chip die; it is believed that the number of cores will reach the hundreds in not so many years from now. However, most common parallel workloads cannot fully utilize such systems. They expose fluctuating parallelism, and do not scale up indefinitely as there is usually a point after which synchronization costs outweigh the gains of parallelism. The combination of these issues suggests that large-scale systems will be either multi-programmed or have their unneeded resources powered off.

Multiprogramming leads to hardware resource contention and as a result application performance degradation, even when there are enough resources, due to negative share effects and increased bus traffic. Most often this degradation is quite unbalanced between co-runners, as some applications dominate the hardware over others. Current Operating Systems blindly provide applications with access to as many resources they ask for. This leads to over-committing the system with too many threads, memory contention and increased bus traffic. Due to the inability of the application to have any insight on system-wide resource demands, most parallel workloads will create as many threads as there are available cores. If every co-running application does the same, the system ends up with threads $N$ times the amount of cores. Threads then need to time-share cores, so the continuous context-switching and cache line evictions generate considerable overhead.

This thesis proposes a novel solution across all software layers that achieves throughput optimization and uniform performance degradation of co-running applications. Through a novel fully automated approach ($DVS$ and $Palirria$), task-parallel applications can accurately quantify their available parallelism online, generating a meaningful metric as parallelism feedback to the Operating System. A second component in the Operating System scheduler ($Pond$) uses such feedback from all co-runners to effectively partition available resources.

The proposed two-level scheduling scheme ultimately achieves having each co-runner degrade its performance by the same factor, relative to how it would execute with unrestricted isolated access to the same hardware. We call this fair scheduling, departing from the traditional notion of equal opportunity which causes uneven degradation, with some experiments showing at least one application degrading its performance 10 times less than its co-runners.
Acknowledgements

I would like to give special thanks to...

*Mats Brorsson*
for giving me clear perspective

*Karl-Filip Faxén*
for being patient and helpful with all sorts of engineering troubles

*Christian Schulte*
for giving me the initial inspiration to follow systems engineering

*Mothy Roscoe, Tim Harris, and Dorit Nuzman*
for support that gave me confidence to come through the other end.

*Sandra Gustavsson Nylén and Marianne Hellmin*
for handling all procedure headaches on a moment’s notice any given day

*Ananya Muddukrishna and Artur Podobas*
for accompanying me on this difficult ride

*Georgia Kanli*
for being my sounding board daily, my biggest supporter and second harshest critic

everyone who listened to my questions and helped me along the way

and last but not least

*Avi Mendelson, Willy Zwaenepoel, Mats Dam, Håkan Grahn, and Markus Hidell*
for giving this journey its deserving ending
# Contents

- **Contents** v
- **List of algorithms** ix
- **List of Figures** x
- **List of Tables** xiii

## I Prologue

1. **Introduction and Motivation** 3
   1.1 Motivation 3
   1.2 Background 5
      1.2.1 Application level 6
      1.2.2 System level 7
   1.3 Problem statements 8
   1.4 Research methodology 9
   1.5 Thesis Organization 9

## 2 Parallel Architectures

2.1 Introduction 13
2.2 Anatomy of Multi-Core Chips 15
2.3 Many-core architectures 20
2.4 Cache Hierarchies 20
2.5 Interconnect Networks 22

## 3 Parallel Programming Models

3.1 Introduction 27
3.2 Task-centric programming models 27
3.3 Characterization of task-based applications 29
3.4 Scheduling task-parallel applications 31
   3.4.1 Work-distribution overview 32
3.5 Work-stealing 33
## CONTENTS

3.5.1 Task queues ........................................... 34  
3.5.2 Scheduling Policies ................................. 35  
3.5.3 Victim Selection ................................. 35  
3.5.4 Discussion ........................................ 37  
3.5.5 Existing work-stealing task schedulers .......... 38  

4 Operating System Scheduling ......................... 41  
4.1 Introduction ...................................... 41  
4.2 Basic concepts .................................... 42  
4.2.1 Resource partitioning strategies .......... 42  
4.2.2 Interrupts ..................................... 43  
4.2.3 Context switching ............................ 43  
4.2.4 Thread management ......................... 43  
4.3 System-wide scheduling issues .................. 44  
4.3.1 Selecting resources .......................... 44  
4.3.2 Fairness ....................................... 45  
4.3.3 Efficiency ..................................... 47  
4.3.4 Requirements estimation trust ............. 48  
4.3.5 Scalability: computability and overhead .... 48  
4.3.6 Use case 1: a first attempt on the Barrellfish OS 49  
4.3.7 Use case 2: The evolution of the Linux scheduler 51  
4.4 System-wide scheduling principles ............ 53  

II Solution Space ....................................... 57  

5 Adaptive Work-Stealing Scheduling ................. 59  
5.1 Introduction ...................................... 59  
5.2 Parallelism quantifying metrics ................. 60  
5.3 Deciding allotment size satisfiability ........ 61  
5.3.1 ASTEAL ...................................... 61  
5.4 Computing estimation requirements ............ 62  
5.5 DVS: Deterministic Victim Selection ........... 63  
5.5.1 Description .................................. 63  
5.5.2 Formal definition ........................... 66  
5.5.3 Implementation .............................. 68  
5.5.4 Discussion and Evaluation .................. 72  
5.6 Palirria: parallelism feedback through DVS .... 79  
5.6.1 Description .................................. 79  
5.6.2 Definition and discussion ................. 82  
5.6.3 Implementation .............................. 93  
5.6.4 Discussion and Evaluation .................. 97  

6 Pond: Exploiting Parallelism Feedback in the OS ... 105
## CONTENTS

6.1 Introduction .................................................. 105
6.2 Restructuring the problem .................................... 106
6.3 Process model ................................................... 108
6.4 Pond-workers: Threads as a service ........................ 109
   6.4.1 Pond over a monolithic kernel ....................... 110
   6.4.2 Pond over a micro-kernel ............................. 111
6.5 Scheduling ..................................................... 112
   6.5.1 Architecture modeling ................................. 113
   6.5.2 Space sharing ........................................... 117
   6.5.3 Time Sharing ............................................ 119
6.6 Experimental Methodology .................................... 120
6.7 Evaluation ..................................................... 122
   6.7.1 Adversarial Scenario ................................. 124
   6.7.2 Trusted Requirements Estimation ...................... 127
   6.7.3 Limitations ............................................. 130
6.8 Related Work .................................................. 130

III Epilogue .......................................................... 133

7 Summary and Future Work ....................................... 135
   7.1 Thesis summary ............................................. 135
   7.2 Contributions ............................................... 136
   7.3 Future work ................................................ 137

Bibliography ...................................................... 141
List of Algorithms

1 Sequential Fibonacci ........................................ 28
2 Task parallel Fibonacci ...................................... 28
3 DVS Classification: Main function to construct the victims sets of all nodes. 69
4 DVS Outer Victims: Find a node’s outer victims. .................. 70
5 DVS Inner Victims: Find a node’s inner victims. .................. 70
6 DVS source Victims set: Construct the source node’s victims set. ..... 71
7 DVS Core X victims set: Construct the core X victims set extension. 71
8 DVS Z Class victims set: Construct the Z class victims set extension. 71
9 Palirria Decision Policy: Check DMC and change allotment respectively. 93
10 Incremental DVS Classification: Adapted function to construct the victims sets of nodes from some zones. Code segments identical to algorithm 3 are omitted. ......................................................... 96
11 get_distance_of_units: implementation for a mesh network. .... 114
12 get_units_at_distance_from: implementation for a mesh network. 115
13 get_distance_of_units: implementation for a Point-to-point link network. ...................................................... 116
14 get_units_at_distance_from: implementation for a Point-to-point link network. ...................................................... 116
15 get_distance_of_units: implementation for a ring network. ...... 116
16 get_units_at_distance_from: implementation for a ring network. 117
17 GetNextSource: Pick worker in biggest idle area as next source. 118
18 MinWeightCore: Select worker furthest away from all existing sources. 118
19 Increase: Add workers, most idle and closest to the source .................. 119
20 Decrease: Remove workers furthest from the source ................. 119
21 Run: Decides next application to run. .......................... 121
# List of Figures

1.1 Avoiding contention and over-commitment through space sharing OS scheduling  

2.1 Combined results from SPEC CPU 1995, 2000, and 2006  
2.2 The free lunch is over  
2.3 Diagram of a generic dual core processor  
2.4 Block diagrams of Intel’s Core 2 Duo and Pentium D, and AMD’s Athlon 64 X2  
2.5 Example SMT and CMP block diagrams  
2.6 Block diagram of the Sandy Bridge microarchitecture, Intel’s first mainstream SMT CMP.  
2.7 Block diagram of the Xeon E5-2600 chip, implementing the Sandy-Bridge microarchitecture  
2.8 Block diagram of a 4 module, 8 core, AMD Bulldozer microarchitecture, aimed to compete with Sandy-Bridge.  
2.9 Cache hierarchy implementations for various processor designs  
2.10 AMD Opteron series interconnect designs  
2.11 Simplified diagrams of Intel’s QuickPathInterconnect (QPI) implementations in variable sized Xeon based systems.  
2.12 Block diagram of EZ Chip’s 100 core Tile architecture.  
2.13 Simplified block diagram of Intel’s Xeon Phi interconnect network.  
3.1 Recursive Fibonacci parallelized with tasks  
3.2 Task tree of $FIB(5)$ as shown in algorithm 2  
3.3 A task-parallel applications complete task-tree can quantify parallelism properties  
3.4 A subtree can visualize the maximum potential for sequential work on the core where the root was placed.  
3.5 Visualization of *task granularity* and *parallelism potential* onto a task tree  
3.6 Work distribution strategies  
3.7 Simple visualization of sequential selection  
3.8 Visualization of leapfrogging and transitive leapfrogging  
4.1 Performance degradation variance using non-adaptive runtime on Linux  
4.2 Performance degradation variance using adaptive runtime on Linux
List of Figures

4.3 A snapshot of the BIAS system-wide resource manager over Barrellfish

4.4 Co-running processes A and B over CFS

4.5 A snapshot of the BIAS system-wide resource manager over Barrellfish

4.6 Co-running processes A and B over CFS

5.1 DVS virtual mesh grid topology for 64 physical cores

5.2 DVS worker placement on the virtual topology

5.3 DVS distance definition through hop-count

5.4 25 worker DVS classification

5.5 Task relocation as load flow through work-stealing

5.6 Two example allotments on two different architectures, classified as per the DVS rules

5.7 Workload input data sets for DVS and Palirria evaluation on Barrellfish and Linux

5.8 First successful steal latency evaluation

5.9 Evaluation of successful steals with DVS on Barrellfish and Linux

5.10 Evaluation of per worker useful execution time for Sort on real hardware

5.11 Standard deviation of useful time among workers with DVS

5.12 DVS execution time evaluation

5.13 Outer victims

5.14 Workers of class Z can be characterized as peak and side, based on their location relative to the topology’s diamond shape

5.15 Ideal DVS model distance constraints

5.16 Results on execution time, wasted cycles, and adaptation on Barrellfish (simulator)

5.17 Useful and non-useful normalized time per worker ordered by zone on Barrellfish (simulator)

5.18 Results on execution time, wasted cycles, and adaptation on Linux (real hardware)

5.19 Useful and non-useful normalized time per worker ordered by zone on Linux (real hardware)

6.1 Visualization of Pond as a departure from the traditional process model

6.2 Pond consists of four modules

6.3 The proposed threading model could be considered as either 1:1 or N:M

6.4 Action sequence for loading and running applications over Pond

6.5 Pond over Barrellfish

6.6 Two examples of modeling a typical quad-core, 8 hardware thread, Intel iCore processor

6.7 State machine for time-sharing a worker between 2 applications

6.8 Performance degradation variance using non-adaptive runtime on Linux and Pond

6.9 Performance degradation variance using adaptive runtime on Linux and Pond

6.10 Performance degradation variance and relative worst degradation on Linux and Pond with dishonest estimations

6.11 Per-application performance with dishonest requirements estimations on Linux and Pond
6.12 Performance degradation variance and relative worst degradation on Linux and Pond ........................................ 128
6.13 Per-application performance on Linux and Pond ....................................................................................... 129
List of Tables

2.1 Cross-node communication latencies in cycles for a 8-node AMD Opteron 6172 system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 Cross-node communication latencies in cycles for a 2 node Intel Xeon E5-2697 system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
6.1 Application input data sets for the evaluation of Pond . . . . . . . . . . . 121
6.2 Application ideal allotment size in number of worker threads . . . . . . . . 122
Part I

Prologue
Chapter 1

Introduction and Motivation

1.1 Motivation

Two-level-Scheduling — cooperative scheduling between operating system and applications — has emerged as a viable solution for dynamically adaptive placement of parallel applications on shared memory platforms, like modern multi- and many- core architectures also called Chip Multiprocessors (CMPs). However, current solutions that address scheduling of such applications on multiprogrammed systems heavily favor overall throughput and not individual application performance. Multiprogramming leads to hardware resource contention and as a result application performance degradation, even when there are enough resources, due to negative share effects and increased bus traffic. Most often this degradation is quite unbalanced between co-runners, as some applications dominate the hardware over others. This thesis proposes a novel solution across all software layers that achieves throughput optimization and uniform performance degradation of co-running applications.

The universal shift to CMP architectures has radically altered the scheduling requirements at both the application and the system level. What used to be a fight for a timeslice on a single CPU, becomes a fight for exclusive access on numerous CPUs. The former situation was conceptually a simple trade-off; the OS would give applications equal access to the processor, delegating solely to the application’s runtime the responsibility of effective use of that CPU time. A single CPU also meant uniform access to memory, and although per application memory footprint was an issue for capacity reasons, it was as manageable as on modern CMPs with banked cache hierarchy levels.

Today parallel applications spread their computation on arbitrarily many processing cores with non-uniform memory access, due to having the cache spread over multiple banks all over the topology; a cache coherency protocol handles replication and synchronization across those banks, while it has been shown that these protocols are hard to scale up. Hence hardware parallelism has created several performance degrading circumstances, such as bus and cache contention.

The aim of current Operating Systems on multiprogrammed CMPs is to partition re-
sources among running applications, in a way that minimizes contention and other negative effects while maximizing throughput. This last statement however is troublesome; throughput maximization does not require the simultaneous optimal execution of all applications. If each application can fully utilize the whole system, it is in principle acceptable to run them one after the other. In this thesis we axiomatically assume that multiprogramming necessitates maintaining \textit{per application performance in par to isolated execution}. In other words, co-runners should be expected to perform as if the system, or a subset of it, was dedicated to them; negative share effects should be minimized; if resources are not enough for all co-runners and performance degradation is unavoidable, they should all degrade uniformly.

Practical example scenarios, where co-runners are run with equal priority and are expected to execute to completion as fast as possible, would be: a multi-user system, where each user is executing parallel workloads expecting the shortest possible execution time, or a multi-tasking single user shifting focus between parallel applications running simultaneously in the background. In each scenario simultaneously running applications are expected to run concurrently thus needing to fight for access to the hardware.

This thesis redefines \textit{fair scheduling} as having all co-runners execute in proportionally the same time relative to each one’s execution time if run in isolation on the same hardware. In other words, every co-runner should degrade its performance by the same positive factor, relative to its execution time with isolated unrestricted access to the same hardware. Of the degradation factor can also be 0 if the available hardware can accommodate the complete requirements of all co-runners\textsuperscript{1}.

Current Operating Systems blindly provide applications with access to as many resources they ask for. This leads to over-committing the system with too many threads, memory contention and increased bus traffic. Due to the inability of the application to have any insight on system-wide resource demands, most parallel workloads will create as many threads as there are available cores. If every co-running application does the same, the system ends up with threads \( N \) times the amount of cores. Threads then need to time-share cores, so the continuous context-switching and cache line evictions generate considerable overhead.

A naive approach to improve this situation, assigns resources proportionally to the parallelism available in the application, avoiding system over-commitment and unnecessary context switching, as shown in figure 1.1.

The naive approach fails to account for certain hardware and software related properties, which can have a significant effect. Application resource requirements are rarely fixed throughout the workload’s execution; most real life algorithms expose fluctuating parallelism. Thus the problem of estimating those requirements is stochastic. Another issue is heterogeneity in both hardware and software; on one hand available cores can offer varying capacity or functionality; on the other hand not all parallel jobs are of equal demands, as there could exist I/O, compute, or memory bound portions.

\textsuperscript{1}We assume that the comparison is done relative to the optimum performance in isolation, hence degradation cannot be negative
I have devised a method to achieve the goal of fair scheduling as defined above via a completely automated method, targeting mainstream CMP systems like workstations, desktops, laptops, and handheld devices, where the user is not expected to tune the workloads, and the vast hardware fragmentation negates any tuning possibility from the developer’s side. This method spans the whole stack, with components in the Operating System scheduler, the application’s runtime, and the application itself. Nevertheless, each component is self-contained, their implementation completely decoupled, all following a generic API.

Prototype implementations of my proposed designs have shown up to 72% reduction in performance degradation variance between co-runners, combined with up to 171% average performance improvement in the same multiple application configurations. This results stem from combining three components: DVS and Palirria provide a novel approach to application driven instant parallelism quantification, and Pond an OS-level scheduler which leverages application provided parallelism feedback to effectively partition available resources between them.

1.2 Background

The efficient placement of parallel application code on the distributed components of modern CMPs, is characterized by the achieved application performance; its execution time, where least is best. Graceful execution of any application depends on the availability of two primary types of hardware resources: CPU time and cache memory. To minimize the execution time, allotted CPU time and memory must be consecutive and uninterrupted; which means no time-sharing of the processor and no sharing of the cache. These are the two major issues that scheduling needs to address for making placement more efficient. For parallel applications there is a third factor which is locality; remote allotted resources must be close together to reduce data transfer time between threads.
CHAPTER 1. INTRODUCTION AND MOTIVATION

When a CMP system is multiprogrammed with many such parallel applications, minimization of each one’s execution time becomes a NP-complete scheduling problem. This problem is the sum of two requirements, the acquirement of which are the two problems stated at the beginning of this section: on one hand identifying the amount of parallelism in the application is necessary for knowing its requirements in terms of hardware resources; on the other hand the system must know at every moment the true available capacity of all physical resources.

The rest of this section gives a brief introduction on the background of these two requirements; the first being a responsibility of the applications, and the second that of the operating system.

1.2.1 Application level

A real-life parallel workload seldom exposes static parallelism throughout its execution. Fully characterizing or profiling such an application for every input is an impossible and tedious task. Alternatively the runtime could employ a recurring online estimation mechanism, in order to dynamically approximate requirements over time. Development of a proper requirements estimation algorithm consists of three components: i) a metric to quantify available parallelism, ii) conditions over that metric to evaluate the suitability of given resources, and iii) a computation method that is cost-effective and accurate.

Plenty of related projects have explored using hardware counters in order to profile application behavior [19, 156, 28, 33, 43, 138]. However there are specific problems with such methods. For one, using hardware counters by definition quantifies past behavior which is not guaranteed to persist. Hardware events are not portable; they are not standard even between processor models of the same series, forcing software libraries that handle them to be platform dependent. Finally and most importantly, mapping raw hardware data to abstract program behavior is not easily generalized; we’ve found that there is no clear correlation between hardware events and contention. One example regards the md bechmark from the SPEC OMP 2012 suite, which run faster when collocated with the art benchmark; moreover md collocated with any other application suffered from almost doubling of the cache miss ratio, without however affecting its overall performance. In those tests, each benchmark was given exclusive access to the same number of cores, both when running in isolation and when collocated. We attributed these results to the co-runner forcing the CPUs to throttle up. Consequently reasoning on application behavior from hardware derived raw data is complicated and needs to be custom tailored to the profile of the monitored application.

A proper parallelism metric must quantify future work independently of hardware architecture and past utilization. Such a metric could be derived from the semantics of the programming model. Finally per thread hardware profiling is not a lightweight method at large scale; although gathering data is quite fast, parsing and decision making must consume CPU time; at very large scale this overhead could prove considerable.

A metric that estimates future requirements without considering past performance can at best be an approximation. Moreover, the error margin can only be evaluated retroactively. Hence the decision policy must be opportunistic at a very fine granularity, in order
1.2. BACKGROUND

to capture sudden fluctuations. A common problem is that absence of enough parallelism can hinder the generation of more parallelism; thus the decision policy must also be optimistic by asking a bit more than what it deems currently sufficient.

Executing a decision policy repeatedly on a rather small interval — a few milliseconds at most — can potentially produce considerable overhead at large scale. It is obvious that as the number of threads grows, so do the amount of profile data, and subsequently the time to process them all. Making the execution time monotonically independent of the size of data requires to either constrain the size of data by profiling only a subset of threads, or to reduce the processing of profiling data as their size increases. In the first case, evaluating the statistical significance of a sample can be performed only retroactively. Then reducing the computation size could potentially increase the estimation’s error margin per thread; less estimation accuracy would potentially lead to loss of either performance (under estimating) or unnecessary costs (over-estimating).

1.2.2 System level

A multiprogrammed system can exploit online parallelism feedback from applications, to assist in improved partitioning of resources. The crucial criteria would be selecting the appropriate resources that benefit each application, while fairly stifling them to control contention. Algorithmically the distribution of resources among many applications can be reduced to 2 dimensional bin-packing, a well established combinatorial NP-hard problem; the reduction makes the problems equivalent and in extent resource partitioning is also NP-hard.

Unfortunately these problems do not enjoy a polynomial-time approximation scheme. In the case of resource distribution on multiprogrammed systems we must accept that the common case is for the available resources to be either just enough or not enough at all. In terms of the bin packing problem that means that the given objects do not all fit in the bin. Thus the amount of space wasted by the imperfect fitting of objects is the primary property for judging solution optimality. Polynomial-time approximation solutions do not provide any guarantee on the optimization of that property, which requires either exhaustive search or heavy computation of complicated constraints; both attributes are not applicable in terms of system scheduling which needs to consume as little resources as possible. A better strategy is to change the problem being solved into something manageable.

There are two primary strategies for partitioning CPU time among running applications: time-sharing and space-sharing. The former implies that a single processing core is shared by many threads, belonging to the same or different applications; each thread is given a time slice, that is a predefined time window of unrestricted access to that core, then it is interrupted and execution switches to another thread. Slices are kept small enough to illude concurrency. However, switching — formally called context switching — has an overhead as it requires saving the state of the currently running thread, restoring the state of the one to run, and flushing the TLB. Space-sharing avoids such overheads by partitioning cores into disjoint sets, although it might constrain applications to fewer than the optimal number of cores. These strategies are not necessarily mutually exclusive and a hybrid approach is quite common.
The schedulers currently implemented in mainstream operating systems are not equipped to handle the issues of multiprogramming with highly parallel applications. They prioritize time-sharing, after providing applications with access to any amount of processors, thus making overcommitment unavoidable. Furthermore their notion of fairness is not truly fair considering the implications; for example the Linux Completely Fair Scheduler will split CPU time uniformly across running applications, regardless of their parallelism. This tips the scale significantly in favour of the less parallel applications. Consequently the potential variance of performance degradation under heavy load is quite high; in other words while one application executes closely to its ideal time (as if running in unrestricted isolation on the same hardware), others will degrade their performance considerably.

Nevertheless there has been extensive prior work on contention aware schedulers. Their shortcoming is non-portability which takes two forms: either by depending on hardware event measurements which are not always mappable to application behaviour, or by exploiting certain non universal hardware properties, like cache organization. In contrast, this thesis aimed from the start for a just-works method that is both hardware agnostic thus portable, and highly scalable by being deterministic with low computation complexity. The unavoidable trade-off is an increased implementation complexity and a requirement for a certain resource abundance that is not yet mainstream.

1.3 Problem statements

In this thesis I explore methods for cooperative scheduling between system and application layers. In doing so it addresses the following problems:

- Collocating multiple parallel applications with diverse and fluctuating hardware resource requirements can cause negative share effects that unpredictably and non-uniformly degrade application performance.
- Current Operating System scheduling favors overall throughput rather than individual application performance
- Applications can provide information on their resource requirements while the OS can monitor system-wide demands and availability. Both information is necessary for the resource partitioning when multiprogramming parallel architectures. However there are currently two issues:
  - Runtime schedulers, and particularly for task-parallelism, fall short in providing a portable and scalable method for extracting resource requirements dynamically without prior profiling.
  - OS schedulers are not yet capable of leveraging application parallelism feedback in a portable and efficient way
  - The currently accepted notion of scheduling fairness penalizes either the less dominant or more demanding applications, instead of uniformly spreading the unavoidable contention penalty across all running processes
• Related work on estimating resource requirements and providing parallelism feedback has not explored methods for handling dishonesty. In effect they freely service requests potentially over-committing the system in adversely competitive environments.

1.4 Research methodology

The contributions presented in this report are the outcome of combining several methodologies. Initially related work was explored, implementing prototypes and experimentally evaluating them in the specific context that is targeted by this thesis. This experimentation allowed to identify inefficiencies and limitations which resulted in the design of a custom model, whose properties matched the desired objectives; the model was first analyzed theoretically, mathematically proving several of its properties. Using the developed theory as foundation, an extensive design was sketched out covering all affected layers in the software stack, then followed by the evolutionary implementation of several prototypes. Each prototype generation was experimentally evaluated, exposing opportunities for further optimization and refinement of both theoretical and engineering aspects.

1.5 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2

The universal turn to multi-core and many-core architectures has shifted software development toward new programming paradigms. This chapter introduces an anatomy of modern processor designs and expands on the implications between hardware components, parallel application development, and scheduling of such applications.

Chapter 3

This chapter discusses the landscape of parallel programming models. It presents their trade-offs and focuses on Task parallelism, which is the focus of this thesis. The properties of the exposed parallelism are identified, as defined by different programming models; this chapter then moves on to discuss how these properties shape the possible runtime scheduling strategies.

Chapter 4

This chapter introduces the issue of resource partitioning on multiprogrammed systems. It starts by identifying what is the conceptual aim of such scheduling tasks, then discusses different approaches, and concludes with a list of specific principles which the optimal OS scheduler should embody. Parts of this work were presented at:

Chapter 5

This chapter expands on adaptive scheduling strategies for task parallel applications as it relates to modern hardware architectures from chapter 2, and programming model properties from chapter 3. It expands on a method for estimating application resource requirements, via a novel non random, online, hardware agnostic algorithm. A novel metric for the quantification of available parallelism, and the adaptation decision policy are formally defined through a rigorous mathematical foundation, and evaluated comparatively to related projects. This work was published in:


• **Georgios Varisteas** and Mats Brorsson, "DVS: Deterministic Victim Selection to Improve Performance in Work-Stealing Schedulers", in *Proceedings of Programmability Issues for Heterogeneous Multicores (MULTIPROG’14)*, Viena, Austria, January 22 2014.


Chapter 6

This chapter focuses on the operating system layer, detailing the requirements for efficient hardware resource partitioning on multiprogrammed systems. The operating system, knowledgeable of resource availability and system wide utilization, requires a method for fair multi-criteria partitioning of said resources among multiple running applications. Such method is dependent on per application feedback of requirements. The potential for feedback honesty is discussed and a method for negating dishonesty potential is detailed. This work is written in the following manuscript:
1.5. THESIS ORGANIZATION


Chapter 7

This chapter summarizes the thesis, emphasizing on important conclusions, and lists prospective future directions.
Chapter 2

Parallel Architectures

2.1 Introduction

In recent years there has been a radical turn in hardware design toward multi-core. The continuous increase in single thread performance hit a dual wall — ILP and power— by the turn of the century, forcing the exploration of alternative methods for performance improvement. Intel around 2004 canceled its single-core projects Tejas and Jayhawk [143, 55], never to revive them, focusing solely on dual-core chips.

As figure 2.1 shows, both integer and floating-point instruction performance improvement was reduced radically with models released during 2004, never to regain its momentum. What changed was the ability to scale instruction level parallelism (ILP), and at the same time power leakages hindered the clock frequency increase.

Until 2004 processor transistor count would double every 18 months, as would clock frequencies and power consumption. This was achieved primarily through miniaturization, consistently moving to ever smaller lithography with every new processor generation. Eventually the Power Wall was hit where the rate of dynamic CMOS power dissipation over leakage current would become far from negligible [88].

Dynamic power is given as $P \propto CV^2Af$. Supply voltage $V$ (usually $V_{dd}$ too) dropped steadily with each processor generation, allowing to push more and more transistors into each chip, without major changes in overall power dissipation.

Leakage power, given by $P_L = V(ke^{-qV_{th}/(ak_aT)})$, used to not be significant with early x86 models. Affected by temperature and density, leakage power increased 7.5 times with every technology generation [27], eventually accounting for more than 20% of total chip power consumption [88], becoming a primary constraint in hardware design.

The power wall resulted in plateauing clock frequency improvements, (as shown in figure 2.2) as the resulting high temperature would have exponential effect on leakage power.

The ILP-wall came as a result of continuously diminishing returns from an ever increasing complexity in the processor issuing logic. After fully exploiting super pipelining, superscalar execution, and eventually out-of-order superscalar execution, every further op-
(a) Integer SPEC CPU adjusted results.  
(b) Floating point SPEC CPU adjusted results.

Figure 2.1: Combined results from SPEC CPU ’95, ’00, and ’06. Images courtesy of [128].

Both the ILP and Power walls contributed in the universal move of the hardware industry toward multi-core processor designs. The extent of this move is such that single core products are no longer available. Multi-core processors transitioned computing from faster to more; instead of faster computing we now only have the option of more computing.
(scale out the problem, maximize throughput) or parallel computing (scale out the solution, reduce execution time). Parallelism came with a variety of hardware design changes that greatly affect programming models.

The rest of this chapter introduces the primary components of modern multi-core chips and how they affect software development.

2.2 Anatomy of Multi-Core Chips

The simplest design of a multi-core chip is shown in figure 2.3. Although multi-core processors did not introduce cache hierarchies — single core Pentium Pro and Pentium II chips implemented multi-level cache banks already — they effectively put them at work. Cache hierarchies comprise of multiple levels of variably configured cache banks. Designs with 3 such levels have been established today: L1 being close to each core, private to it and with very small communication latency, thus necessarily small in size; L2 being significantly larger and further away thus inflicting higher communication latencies; finally there may be a large L3 as a compromise between the exorbitant latencies of reading from main memory and chip area concession. There have been processors with off-chip optional L3 like the IBM Power 5.

Figure 2.3: Diagram of a generic dual core processor. Dedicated L1 cache per core, shared L2 cache and front side bus.

Figure 2.4 shows the designs of a few early mainstream multi-core chips from Intel and AMD. As discussed in [123], these chips explored redesigns of cache architecture and coherency mechanisms, departing from the norms of the time. Cache coherency is the hardware mechanism for maintaining a common view of shared data across cores. Thus it is a mechanism that updates and discards cache lines so as each core sees the latest values.

Intel’s Core 2 Duo incorporated a shared L2 and employed a MESI coherence protocol internally between the two private L1 caches. Pentium D incorporated private L1 and L2
caches, and used a variation of MESI with update based coherency via the off-chip front side bus (FSB). Early AMD multi-core models used the MOESI protocol for coherence between private L1 and L2 caches.

Inter-core data and message exchange, within the same or remote processors, is necessary especially for cache coherency. These communication channels form a network called the Interconnect. Early multi-core chips built upon Simultaneous MultiProcessor (SMP) technologies, using the generic PCI or FSB system buses borrowed from single core systems. However they were quickly replaced by dedicated more efficient buses like Sun Niagara’s crossbar. The AMD Athlon 64X2 processors brought the crossbar into the mainstream and introduced AMD’s HyperTransport technology.

Fast forward a few years and these early models got replaced by much more elaborate and complicated designs. The first implementations of multi-cores, consisting of 2 to 4 single core chips glued together, gave place to architectures specifically designed from scratch as multi-cores. Cache hierarchies became more elaborate, so did coherency protocols, better supporting advanced parallel programming constructs, and sporting up to 16 or even more cores. Specialized interconnect networks replaced system buses for inter-core communication across the board, enabling fast data exchange and more scalable coherency. Intel introduced QuickPathInterconnect (QPI) to compete with AMD’s HyperTransport.

Simultaneous Multithreading (SMT) (figure 2.5a) [56, 142] is another important aspect of multi-core chips, where multiple independent pipelines are implemented over the same issue logic. Each of these pipelines is called a Hardware Thread. In an SMT core, all threads share the same register file — much larger from non SMT designs — and cache hierarchy, among other resources. Unfortunately this scheme is not really scalable, due to severe constraints on cache bandwidth and size. Moreover, distinct pipelines cannot be powered or clocked down independently of others. Hence power consumption could be disproportional to utilization.

Chip MultiProcessors (CMP) (figure 2.5b) [121, 122] addressed the scalability and tuning issues of SMT designs. Maintaining the same transistor count, hardware threads are made independent — called cores — each given its own dedicated L1 cache (if not L2 too).
2.2. ANATOMY OF MULTI-CORE CHIPS

(a) Simultaneous MultiThreading (SMT) allows to multiplex instructions from different processes, each in its own pipeline, sharing registers and cache hierarchy.

(b) Chip MultiProcessors (CMP) addressed the scalability issues of SMT by reducing the hardware threads per core, but increasing the cores per chip.

Figure 2.5: Multi-Core chips expose even more hardware parallelism by providing multiple independent pipelines, in addition to their ILP.

and register file. Apart from reduced latency causing resource sharing, the ability to power or clock each core independently of others maintaining a power budget with improved proportionality to the utilization, is another important benefit of CMP designs. A core within a CMP chip can also implement SMT, including more than one hardware threads per core and more than one core per chip; such design is called SMT CMP.

Intel introduced its SMT technology marketed as HyperThreading (HT); it had 2 hardware threads sharing the execution engine, caches, the system-bus interface, and firmware. Intel’s first SMT CMP was named Sandy Bridge, succeeding Nehalem/Westmere. AMD’s SMT technology, informally named Clustered Multi-Thread (CMT), is similar with the exception of distinct integer clusters and L1 Data-cache per thread. As an example figure 2.8 shows the block diagram of the Bulldozer microarchitecture.

As a use case, figures 2.7 and 2.8 show two chips targeting the server market, both
CHAPTER 2. PARALLEL ARCHITECTURES

Figure 2.6: Block diagram of the Sandy Bridge microarchitecture, Intel’s first mainstream SMT CMP.

Figure 2.7: Block diagram of the Xeon E5-2600 chip, implementing the Sandy-Bridge microarchitecture.
2.2. ANATOMY OF MULTI-CORE CHIPS

released around March 2012 by Intel and AMD respectively. They implemented all technologies discussed so far in this section. The Intel chip had 8 HT cores, each with private L1 and L2 caches, and a large banked L3 cache shared among all cores. A bidirectional full ring interconnect (QPI) handled coherency and data exchange within the chip, including a specialized QPI Agent module for cross-chip coherency and communication. The AMD chip had 4 CMT cores, and a HyperTransport crossbar interconnect for inter-core communication locally and cross-chip.

Multi-core is the prevalent processor design today, adopted for all types of systems from high-demand servers to low power handheld and wearable devices. Every modern multi-core chip is composed of at least one SMT core or two non-SMT cores, an interconnect network, and a cache hierarchy. Developing software that is efficiently parallelized for such architectures requires awareness on the implication of those components.
2.3 Many-core architectures

The term many-core is used to describe not yet mainstream designs comprised of tens of cores. No formal threshold has been defined yet for the separation of multi-core and many-core, although 32 is a good approximation. There have been several prototypes of many-core chips with 64, 72, 100, or more cores. Examples are the 64-core Tile64 [16] and 100-core Tile-MX100 [58] architectures from EZChip (ex-Tilera), the 64-core Epiphany IV architecture [72, 1] from Adapteva, Intel's 48-core Single-chip Cloud Computer (SCC) [76] and the 50-core Xeon Phi [37] from Intel.

The large number of cores in many-core systems requires, among other things, special attention on the interconnect network. At many-core scale inter-core communication latencies can be quite significant, either due to traffic or simply distance. Examples of the elaborate interconnect designs employed to address these issues are mesh networks and bidirectional rings. EZChip's (fig. 2.12) and Adapteva's products are examples of the former, while Intel has chosen to implement the latter (fig. 2.13).

2.4 Cache Hierarchies

Implementing elaborate cache hierarchies was not a new idea introduced by multi-cores. However, with multi-core chips they became the standard for processor design. Each core has at least the L1 cache bank be private; however there have been implementations with private L2 also a common choice for SMT chips, since a single L1 might be constraining for two hardware threads. The most significant aspect of cache hierarchies on multi-cores is cache coherency. Implemented in hardware, coherency is responsible for maintaining a single view of shared datums across multiple private cache banks.

Hardware cache coherency has been shown to be hard to scale over arbitrarily large amounts of cores [35, 152, 31, 99]. Much research has been performed on various methods for improving scalability [36, 112, 132, 108, 31, 82, 89]. Martin et. al has also proposed that the high complexity of scaling up coherency is manageable with simple concessions [108]. Nevertheless, the issue remains that application parallelism over shared memory might be constrained due to the prohibitive complexity to maintain coherence with either hardware or software based protocols.

A second aspect is the non-uniform latencies of accessing shared cache banks (NUMA); for example L3 can be banked (as in fig. 2.7), with varied access latency per core per bank. L3 being shared means that a cache line will have only one copy across all banks. Usually the owner bank will be local to the core that first accessed it from main memory (first-touch policy). As a simplified example, a core accessing a cache line from L3, will copy it from the bank it resides on into its L1 and L2. If that cache line gets reassessed after having been evicted, it will be transferred from the same remote L3 bank again.

Cache organization plays an important role in scheduling parallel computations. Specific criteria are:

- **Capacity contention**: multiple user threads sharing a cache bank might cause excessive evictions due to lack of capacity.
• **False sharing**: multiple threads are changing distinct variables within the same cache line, causing excessive unnecessary invalidations.

• **Data locality**: hardware threads are changing the same datum, cause frequent invalidations and excess traffic for updating values. If these threads are far away from each other — relative to the chip topology — these transfers would be quite costlier than having placed the computations on closely located hardware threads.

Figure 2.9 depicts various cache hierarchy implementations (adapted from [121]). Conventional microprocessors introduced hierarchical caches to leverage the quantity versus latency trade-off (figure 2.9a). Since then, different CMP designs have implemented a variety of shared and private such hierarchies (figures 2.9b – 2.9d). Sharing L2, and even sharing L1 with SMT designs, was a design choice favoring throughput; the execution of as much as possible fine-grained tasks, with small memory footprint, typically encountered in server and network switching settings.

It is the responsibility of software to place computations in such a way that the aforementioned issues are optimized. Such optimizations can be performed by either the operating system scheduler, the application runtime scheduler, the application itself, or by all in
cooperation. Delegating this responsibility to each layer brings forth a different trade-off. Placement algorithms need to be tailored to the specific architecture and topology that they are executed on; hence burdening applications with this responsibility would necessitate a different implementation per available topology, over encumbering development. The operating system is best equipped to analyze the hardware and monitor its resources; however it remains disconnected from applications, unable to identify their actual requirements and inter-thread interaction patterns. On the other hand a application level scheduler faces the exact opposite problem, having access to all application related information, but unable to monitor system-wide resource availability and utilization levels.

2.5 Interconnect Networks

At large scale, the interconnect network of the architecture affects the efficiency of data exchange between threads running on different cores. These effects grow if cores are far apart, sharing no L1 or L2 cache banks. Random placement of computation in respect to data can lead to increased bus traffic, and significant latencies. The non-uniformity in multi-core memory accesses is called NUMA and the property of closeness between data and computation is called data locality. NUMA negative effects are becoming greater as the scale of processors increases, thus preserving data locality is becoming an ever more important attribute of scheduling parallel programs [70, 19, 42, 67].

In recent years the market has seen products implementing a variety of different interconnect networks. Each with its own benefits and limitations. At multi-core scale Intel and AMD have opted for non-uniform networks of variable diameter.

Figure 2.10: Dual processor (2P) blade and quad processor (4P) rack topologies: (a) 2P Max Perf, (b) 4P Max I/O, (c) 4P Max Perf, and (d) 4P Modular 2+2. Image adapted from [41].

Figure 2.10 [41] depicts various implementations for the Opteron processor series, at different scale. Sub-figure (a) shows the optimal configuration for a dual 2-core processor (2P) system, with non-uniform link latencies, heavily dependent on AMD’s XFire technology [40, 90] for uniform traffic distribution. The 2P Max Perf configuration is the only fully connected topology among the given configurations. The quad 2-core processor (4) variants alternate between optimizing I/O and performance using different technology
connectivity schemes, tuning the average and maximum diameter. Either configuration produces variable latencies in data exchange between cores of different processors and sometimes even within the same processors.

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>10</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>22</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>16</td>
<td>16</td>
<td>10</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>16</td>
<td>10</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
<td>16</td>
<td>22</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.1: Cross-node communication latencies in cycles for a 8-node AMD Opteron 6172 system; it has 2 nodes per processor and 6 cores per node. Data gathered using the `numactl` tool.

Table 2.1 presents the latencies in cycles for inter-core communication on a 4 AMD Opteron 6172 processor system. Each processor has 2x6 cores; each node of 6 threads is a fully connected plane, but with variable connectivity across nodes, as in figure 2.10c.

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.2: Cross-node communication latencies in cycles for a 2 node Intel Xeon E5-2697 system; it has 1 node per processor and 6 HT cores (12 threads) per node. Data gathered using the `numactl` tool.

Intel’s solution into point-to-point signaling — called QuickPathInterconnect (QPI) — follows a similar architecture to AMD’s HyperTransport; however QPI has implemented only fully connected topologies even at high scale, with each processor connected to all others, as shown in figure 2.11. Table 2.2 presents the latencies in cycles for inter-core communication on a 2 processor Intel Xeon E5-2697 based system. Each processor has 1 node with 6 HyperThreading cores, making a total of 24 threads. Intra node communication is 10 cycles, while 20 cycles inter-node (off-chip).

The overall inter-core communication latencies for both Intel’s and AMD’s solutions have a variance around 100%, meaning that some link end-points suffer double the latency of others. Such chips, primarily targeted for mainstream usage where performance tuning is minimal or generic, require special attention to data-locality.

Moving on to many-core systems we see more advanced interconnect solutions. Most prominent designs are bidirectional rings and mesh networks. Either design has a much larger average diameter than HT and QPI. A 10x10 2 dimensional mesh would have an
average diameter of $\frac{100}{2} = 5$. The average diameter of a 50 node ring would be even larger at $\frac{50}{4} = 12.5$.

Figure 2.11: Simplified diagrams of Intel’s QuickPathInterconnect (QPI) implementations in variable sized Xeon based systems.

A mesh interconnect resembles a typical large scale network, where each node has a specialized switch engine for forwarding messages appropriately, with packets able to travel on every direction thus reducing traffic per path. The Tile architecture’s implementation [140] — shown in figure 2.12 — comprises of four distinct channels in order to isolate different types of traffic, optimizing efficiency and effectiveness. Furthermore each node has its own private L1 and shared L2 caches. It is important to observe that since inter-core communication is not a direct link between non neighbouring nodes, the possible long distance can incur significant latencies; hence compacted placement of data and computation across multiple nodes becomes a primary scheduling criterion.

Figure 2.12: Block diagram of EZ Chip’s 100 core Tile architecture.
A ring interconnect requires simpler routing of packets. Packets are initially placed on the appropriate ring direction that minimizes the travel distance, and only need to check if the current node is the destination node. On the other hand, traffic worsens since packets follow the same path. Intel released the Xeon Phi [37] series of many-core chips with up to 61 cores, and a total of 244 hardware threads. A simplified block diagram is shown in figure 2.13.
Chapter 3

Parallel Programming Models

3.1 Introduction

Since the introduction of the first SMP systems many different parallel programming models have been proposed [111, 78, 135], with MPI [115] being the most prominent and widely used. The significant latency in exchanging data between processors made sharing of data inefficient. This was the immediate benefit brought by CMPs and shared cache hierarchies. Modern programming models have shifted toward sharing memory between threads [57], enabling much finer grained parallelism [23, 21].

The purpose of a parallel programming model is twofold: on one hand it adheres by legacy standards and practices, while enabling new types of code decompositions as required by emerging parallel architectures. In other words it enables modern hardware in a way that is familiar to the legacy methods.

3.2 Task-centric programming models

To properly compare different methods for building parallel applications, a crucial criterion is how they can express parallelism via the programming model and map it to hardware via the runtime scheduler. For the first part, the most frequent obstacle is overcoming dependencies between parallelizable blocks of code, and therefore require data synchronization. Synchronization inherently serializes the execution as one thread has to wait for others to complete or catch up.

Tasks-centric models [78, 111, 135] have been shown to express parallelism nicely and are capable to handle very fine grained loads. A task is an independent, self-contained, and variably-grained block of code, implemented as a closure: a function coupled with the appropriate input. Thus a set of tasks can and should be executed in parallel. Furthermore, they allow great flexibility in expressing nested parallelism, since it is simple to create a new instance from within another.

Functions are marked as tasks, either through annotations, preprocessor macro directives, or syntactic keywords that extend the programming language used; some implemen-
Algorithm 1: Sequential Fibonacci

1: function FIB(n)
2:     if \( n < 2 \) then
3:         return \( n \)
4:     end if
5:     \( a = FIB(n-1) \)
6:     \( b = FIB(n-2) \)
7:     return \( a + b \)
8: end function

Algorithm 2: Task parallel Fibonacci

1: function FIB(n)
2:     if \( n < 2 \) then
3:         return \( n \)
4:     end if
5:     \( b = \text{spawn } FIB(n-1) \) \( \triangleright \text{create task} \)
6:     \( a = \text{fib}(n-2) \)
7:     \( \text{sync} \) \( \triangleright \text{wait for task} \)
8:     return \( a + b \)
9: end function

Figure 3.1: Recursive Fibonacci parallelized with tasks. Only a few code changes are needed to transform the sequential version into task-parallel.

The API of such a programming model is small and simple. Fundamentally it consists of only two actions: spawn and sync (also called fork and join from the fork/join design pattern). The former creates a new instance of a task. The second action waits for the task to finish with its return value, if any. A sync action can be explicit or implied through the definition of specific code blocks. Many different implementations of the model have their own semantics.

As an example we present the recursive implementation of a Fibonacci solver called fib, presented in pseudocode in figure 3.1. This algorithm is a very inefficient way to calculate Fibonacci numbers in comparison to a loop-based implementation; however, it is a embarrassingly parallel algorithm, with very fine-grained tasks, and no data dependencies. Thus it scales linearly, and is commonly used to evaluate task-parallel runtime schedulers under ideal conditions.

The execution of the task-based FIB program in algorithm 2 will create several instances of the FIB task. Unfortunately the algorithm will spawn many identical tasks, meaning recursive calls to \( \text{fib()} \) with the same value for \( n \). However, each will descend from distinct other tasks. These parent-child relationships between tasks are commonly visualized as a tree graph, called the task tree. The task tree for \( FIB(5) \) is shown in figure 3.2.

A task tree is a useful tool as it reveals several characteristics of a task based application. For one, it quantifies the available parallelism; it shows the task generation potential, and helps identify bounds on the number of tasks available during different points of the execution. For the specific example of figure 3.2, the rate is 2 since every non-leaf task will generate two more; furthermore, the total amount of parallelism is monotonically increasing; thus given appropriate input, FIB can fully utilize any number of processors, which is why it is called embarrassingly parallel.

Other programming constructs, like loops, can also be mapped to tasks indirectly
3.3. CHARACTERIZATION OF TASK-BASED APPLICATIONS

The task-tree is an important tool for the characterization of task-parallel applications. There is a direct correlation between certain properties of the tree and the behaviour of the application, thus it makes an excellent visualization tool for discussing certain aspects of an task-parallel application’s parallelism profile.

At first view the maximum dimensions, width and depth, reveal the maximum simultaneous parallelism of the workload and the critical path respectively, as shown in figure 3.3. The width is a good approximation on the maximum amount of cores that the application can utilize through its execution, using more processing cores than that risks excess synchronization overhead and eventually slowdown. The critical path is the longest path, or maximum depth of the tree; assuming execution of the critical path on a single core, it is the minimum sequentially executable part of the application. Finally the total number of tasks can be used in conjunction with the critical path to approximate the exposed parallelism and calculate a loose upper bound on the execution time, as:

\[ P = \frac{T_1}{T_{\infty}} = \frac{\text{Serial execution time of all tasks}}{\text{Critical path execution time}} \]
CHAPTER 3. PARALLEL PROGRAMMING MODELS

Figure 3.3: A task-parallel applications complete task-tree can quantify parallelism properties

Figure 3.4: A subtree can visualize the maximum potential for sequential work on the core where the root was placed.

The same generalized analysis can be performed on a subtree basis (fig. 3.4). Assuming the root of the subtree is placed for execution on a specific core, then the complete subtree below that root can approximate the potential for that core to run sequentially. The subtree’s critical path is the minimum execution time, while all branches can be assigned for execution in parallel to other cores.

A more elaborate version of the task-tree can also show advanced features of task-parallelism, like data dependencies. Data dependent tasks are spawned and marked non-ready for execution until all their data dependencies are resolved; thus avoiding stalling a processor until the data become available. Indirectly data dependency resolution also helps in placing the computation close to the data, minimizing transfers and contention.

There are more properties which characterize individual tasks, most importantly granularity and parallelism potential. The former is a reference to the task’s execution time relative to an accepted standard or other tasks in the same application, and can be fine (for short) or coarse (for long); for example, a finer-grained task executes in only 30% of the time other tasks need; a fine-grained task has a very short execution time of just a few thousand cycles on modern hardware. A task’s parallelism potential is an approximation or an upper bound on its ability to produce further parallelism by spawning more tasks; a
3.4. SCHEDULING TASK-PARALLEL APPLICATIONS

Figure 3.5: A task has a certain granularity, which is a reference to its execution time relative to the platform standard or other tasks. Its parallelism potential includes the amount of parallelism it produces by spawning more tasks.

A task’s parallelism potential can be the amount of direct descended tasks or the complete size of the subtree rooted at it. Figure 3.5 depicts both these properties on a task tree.

An important task characteristic which cannot be depicted on a task tree, is the task generation rate. This is the rate at which a task’s parallelism potential is materialized, or in other words how quickly a task will make its children available. It could happen that a task employs tail recursion, meaning that its children won’t be available before the end of its execution; for example a task-parallel quicksort implementation will first partition (the computationally meaningful part of a task’s execution), and only after create two new tasks; hence the children cannot be executed in parallel with the parent task and the amount of parallelism is reduced. On the other hand, a task could spawn multiple tasks early on, to segment the load, and then work locally on one of them; for example a task-parallel matrix multiplication algorithm, where the tasks starts by segmenting the matrix, making all segments immediately available as parallelism, then proceeding to work locally on one of those segments. Hence the children tasks can be executed in parallel with the parent task. In that case the maximum parallelism is greater than the tree’s width. The task generation rate can lead to identifying sources of lost performance, but also signify parallelism fluctuations and more specifically increases.

3.4 Scheduling task-parallel applications

In recent years parallel programming has seen a sudden evolution [20]. Existing paradigms where enriched [44, 24, 102, 100] while many new were developed [68, 127, 126]. The underlying runtime schedulers can be separated into two main families: shared memory
and message passing.

Message passing is based on the axiom that no two execution units should share memory. Parallelism is exposed via executing multiple cooperating processes, while all data are copied between them via messages. Data structures are kept synchronized through algorithms adapted from the field of distributed systems. A good analogy of this model is a network of individual systems. Like with any such network, message passing based parallel programs scale well once the proper protocols for data exchange are in place. However, developing such protocols involves a certain design complexity that can be disproportionately larger than the complexity of the problem being solved.

Shared memory based programs are built using independent execution units that access the same memory directly. Data synchronization is constrained by cache coherency, a function often provided by the hardware. The immediate benefit of shared memory models is less code complexity resulting in faster development. However, such models often fall victims of their own design; shared memory read-write accesses require protection in order to avoid data corruption or races. These protection schemes need careful implementation to not decrease parallelism, since they serialize the execution.

This work focuses exclusively on the shared memory model, although the contributions are independent of it; hence the heuristics and algorithms which are presented here could be adapted to message passing also.

3.4.1 Work-distribution overview

Shared-memory programs exploit the construct of threads. In Linux — and most other modern operating systems — a thread is a lightweight process (LWP) implemented on top of a kernel thread. Thus the OS kernel can schedule each thread individually on hardware. All threads share the same address space.

In its most primitive form, a shared memory program would create a new thread for each parallel task. To avoid resource over-commitment and excessive context switching, there would be a cut-off to constrain parallelism disproportionately larger to hardware available concurrency. Unfortunately such mechanisms were hard to decouple from application code and were quickly substituted.

In its more refined form, a shared memory program would create a pool of threads, which are assigned parallel tasks via specific mechanisms. These threads are commonly referred to as workers. The thread that initiates the application and creates the workers is referred to as master. The work distribution techniques can be summed up into two categories: work-dealing and work-stealing.

With work-dealing, new tasks are offloaded to other threads; primary goal is balancing the load across workers. This paradigm can be inefficient; it is probable that the worker spawning new tasks is already overloaded, but is burdened further with having to find the workers to offload these new tasks to. Work-stealing on the other hand, lets the idle threads find new tasks from other sources. Since these threads are idle, the search overhead will have a smaller if any negative effect on application progress, potentially increasing execution time.
3.5. WORK-STEALING

With either distribution technique there can be different implementations on how to store tasks. There could be one central queue storing all newly spawned tasks, minimizing scheduling complexity and synchronization overhead but with poor scalability. I have not considered such scheme in this project at all. On the contrary each worker could have its own queue. Then workers place new tasks in their own queue; if full they could either execute the task immediately as a local function or deal it to an emptier queue. If a worker becomes idle with an empty task queue, it steals tasks from remote queues.

![Diagram of work distribution schemes](image)

Dealing tasks with per-worker queues places the responsibility of work distribution on already overloaded workers. Contrary stealing tasks employs idle workers, letting busy workers focus only on progressing execution of the application. Of course work-stealing is not without its own problems. If the application parallelism is not enough to utilize all workers, the idle ones will be spinning trying to find work when there is none, wasting plenty of CPU time and power. Because most systems run more than one application at a time, such scenarios can cause degradation of performance due to unnecessary contention. This project addresses exactly this issue.

3.5 Work-stealing

The work-stealing methodology comes with some intricate terminology, surrounding the action of stealing. The thief is the worker stealing the task, the task-queue is the queue that holds the task being stolen, and the victim is the worker that owns the task-queue in question.

A work-stealing scheduler’s implementation consists of two main components: a task-queue, and a victim selection protocol. The task-queue must be able to accommodate multiple readers and writers, since thieves must mark tasks as stolen and then write the return value. The victim selection protocol specifies how to pick the next victim; a naive
approach would iterate the list of all workers each time (sequential selection); more advanced techniques employ randomness (random selection); there are further optimizations that exploit the work-stealing scheduling semantics for even more educated selection (i.e. leapfrogging during a blocked SYNC).

A task tree, like the example of figure 3.2, can be split into task subtrees (or just subtrees). When the parallelism of a program is regular and increasing, like FIB, there will be a point during the execution after which each worker will be working on tasks from distinct subtrees. That means that every worker eventually processed a task such that its descendant tasks can keep it busy for the rest of the execution of the program, negating any need to steal. This pattern is referred to as sequential task execution.

3.5.1 Task queues

The task-queue of a worker must accommodate for multiple readers and writers. The owner of the queue will be adding and processing tasks, plus other workers stealing tasks and updating the queue accordingly. Because these two families of actors are independent, task-queues are typically implemented as double-ended-queues (deques); thieves steal from one end, while the owner adds tasks from the other.

An element of the queue is removed only after being synced. Picking up a task either through stealing or by the owner of the queue, needs marking the task’s descriptor as being under processing to avoid double execution. Upon completion the return value must be written back and the descriptor marked as completed. When a worker needs to sync a task that is not completed yet, it can pause the parent task and try to find other work to avoid idleness. Thus the child task won’t be removed from the queue until the worker resumes its parent and completes the syncing. If however, the runtime allows for another worker to resume the parent task there gaps could be created in the task queue.

If the runtime allows migrating paused tasks across workers, elements of a task deque might not be removed in order, even though it is conceptually operating in FIFO order. A task queue can contain tasks from completely independent subtrees due to arbitrary stealing. Even though parent tasks can usually not be synced — thus removed — before child tasks are synced first, tasks of different subtrees produce no ordering constraints between each other. Unordered removal of queue elements will either increase the algorithmic complexity of the deque, or require iterating all elements to find the size of the queue. A task-queue implemented as a non-overwriting array-based circular buffer can optimize unordered removal, with $O(N)$ complexity for the push operation.

To illustrate the two cases above, assume worker $w_1$ executing task $t_1$ which creates child task $t_2$. $t_2$ gets stolen and $w_1$ cannot proceed with syncing. Thus it pauses $t_1$ and proceeds to steal and execute another task $t_3$, $t_2$ and its complete subtree eventually completes and thus $t_1$ can be synced. If task migration is not supported, $w_1$ will resume $t_1$ only after it has completed every other task following $t_2$ in its queue. After syncing, $t_2$ is removed and the tail is effectively $t_3$ again. On the other hand it could be that $t_1$ is resumed by another worker while $w_1$ is busy on an independent subtree and has placed more tasks after $t_2$. Then $t_2$ will be synced, removed from the queue, and a gap will be created.
There have been two prominent strategies on how to handle spawning new tasks. On one hand the work-first policy has the worker switch immediately to executing the newly spawned task, leaving the continuation of the parent as a single stealable task in its task-queue. In contrast with help-first a worker executes each task to conclusion, pushing any spawned tasks onto its task-queue. Each of the two policies has certain benefits and drawbacks.

Work-first employs a depth-first walk over the task-tree, where a worker will advance to the final leaf before backtracking to execute tasks higher up in the tree. This policy will make sure that most of the application’s critical path is executed locally to a worker, also meaning that the stack size that the parallel code has the same stack requirements as the serial version, reducing the possibility to blow it. However, it employs a technique similar to continuation-passing-style transformations by packaging an executing function’s continuation as a stealable task. This technique is not compatible with standard calling conventions and needs compiler support. In effect it requires a special type of stack implementation called a cactus stack [38, 66, 97]. Using a cactus stack makes the work-stealing implementation vulnerable to unordered removal of tasks, increasing the complexity of interacting with the task-queue.

Help-first works the exact opposite way, conducting a breadth-first walk over the task-tree. Every task is executed to completion before the worker picks another task, which means that all children become immediately available. This technique makes concurrency available quickly and allows its faster distribution across workers; with help-first tasks can only be removed in order making task-queue interaction much less complex. On the other hand it can lead to blowing the stack as it allows an execution order that does not follow the serial order.

To exemplify the scheduling differences assume a task $q$ that spawns three children $x$, $y$, and $z$; with help-first they are all pushed onto the task-queue immediately. When $q$ terminates, the worker picks a new task from its task-queue (probably one of $x$, $y$, $z$) or steals if empty. Meanwhile three other workers had the potential able to steal those tasks simultaneously. With work-first, the worker switches immediately to executing $x$ while leaving the continuation of $q$ in its task-queue; tasks $y$ and $z$ were not spawned. Thus only one worker has the opportunity to be utilized. In conclusion, help-first policy reduces per worker latency until first successful steal by accelerating spawns, benefiting wider task-trees; work-first contributes to load-balancing by avoiding mass spawning of tasks by a single worker, benefiting narrow deep trees.

Sequential victim selection is the naive approach to victim selection. There is a shared ordered list of all workers and for selecting a victim a worker iterates that list in sequence. Usually implementations maintain a distinct starting index per-worker $w$; this index could either be fixed to the beginning of the list (always 0), or at $w + 1$, or dynamically set to the latest index that resulted in a successful steal.
Random victim selection was first introduced by Blumofe et. al. in [25] which provided theoretical bounds on its efficiency. In summary if a worker makes $M$ steal attempts then the delay before succeeding is at most $M$. In other words the number of random based steal attempts required for discovering a stealable task is finite. That work eventually led to the Cilk++ programming model [26] described in more detail in section 3.5.5.

A common optimization is to combine random and sequential victim selection to reduce calls to the expensive $\text{rand}()$ function. The starting index is set dynamically at random; workers are then iterated sequentially until a complete circle back to the starting index, when $\text{rand}$ is invoked again and a new sequential session is started.

The $\text{SYNC}$ action is possible to block if the task in question was stolen but not yet completed. To avoid unnecessary idleness, the work stealing scheduler would initiate a steal session and revisit the sync later. Leafpfrogging [149] is an optimization used during a blocked $\text{SYNC}$ phase, where the victim selected is the thief of the stolen task. An extension called Transitive Leapfrogging, implemented in WOOL [59][1], allows a second Leapfrog-
3.5. WORK-STEALING

...ging step, thus a worker transitively selects the thief of the thief when the latter’s task queue is found empty. Figure 3.8 depicts both methods. Intuitively, there is a high chance the worker that stole the task to be synced to have spawned more tasks, thus increasing the chances for finding stealable tasks belonging to the same subtree; both properties, less chance for failed steal attempts and higher chance to steal descendants, can lead to improved performance and saner handling of the stack since workers will be stealing tasks on the same path on the task-tree.

3.5.4 Discussion

There are special considerations regarding the applicability of work-stealing over task-parallel applications. The first has to do with the proportionality of tasks and worker threads. Of course there is no benefit at having more workers than tasks; the excess workers will only spin trying to find tasks while there won't be any. If the amount of tasks vastly exceeds the number of workers then there might be a lost opportunity for transforming concurrency into parallelism and improving performance. A good approximation for deciding the ideal number of workers is the maximal width of the application’s task-tree, which can be different depending on the application’s data input set.

Nevertheless, work-stealing does incur an overhead. Usually through arbitrary transferring of data over different cache banks. Workers placed on cores that do not share the first levels of the cache hierarchy exchange tasks in the same task subtree, resulting in requiring to transfer data back and forth for syncing those tasks, stalling the processor, and suffering unnecessary delays. The tree depth is another important factor to consider. Ideally all workers should, at some point of the execution, be able to work serially over a small task subtree. Thus there is no performance penalty from excessive stealing. It is difficult to achieve balance between these three quantities, namely that task-tree dimensions defining the available parallelism over time, and the best amount of workers for processing that tree.

Dynamic adaptation of the worker pool size reduces the effect of a wrong initial choice. Adaptation could be opportunistic meaning that when the rate of consuming tasks becomes smaller than the rate of producing tasks, it is a good hint to add more workers. Similarly the opposite situation where tasks are consumed much faster than produced, is a good sign to remove workers. When the depth of the tree is unpredictable, one can start with a safe small number of workers and allow the allotment to expand dynamically; this technique will indeed suffer a penalty if the required extra workers are not added quickly enough. Optimally the scheduler would be clairvoyant in identifying imminent increase in parallelism soon enough to have added workers before it becomes available. Certain techniques have moved toward that path by using offline profiling or static scheduling; the impracticality of those methods is that the characteristics of the parallelism profile change with the application’s data input set, and profiling needs to be repeated.

Correlating task-queue size with parallelism fluctuations seems reasonable and has been investigated before [30, 144]. Nevertheless, when reasoning over a pool of workers there has to be a threshold to differentiate between one state and another. Setting such...
a threshold is not simple considering the potential unbalanced state that random victim selection can cause. If workers steal tasks from arbitrary victims, it becomes possible to end up with a worker pool where a few are idling without work and others are overwhelmed with full task queues. Such a scenario poses a scheduling contradiction. Also, it is wrong to disregard distribution disparity and consider only the total amount of tasks; it could be the cause of inaccuracy in the estimations since randomness provides no guarantee on how long it would take to distribute work to underutilized and newly added workers. In conclusion, using the amount of tasks as a metric for the quantification of parallelism requires their uniform distribution across the allotment of workers.

Furthermore, without the use of a clairvoyant scheduler or static analysis it is impossible for the work-stealing runtime to identify any task’s granularity (expected running time); thus a second point of speculation is the threshold for considering a certain amount of available stagnant tasks (spawned but not processed yet) as an indicator for parallelism increase, and even then the correct amount of utilisable extra workers remains unanswered. It can be argued that an allotment of \( X \) busy workers with \( X + Y \) stagnant tasks, can in all cases utilize further \( Y \) workers, including a delay for making those workers available. Because the current workers are already busy, it will take some time before they require any of the \( X + Y \) stagnant tasks; assuming that the extra \( Y \) workers steal \( Y \) of those tasks, there are still \( X \) tasks to utilize the original \( X \) workers. The problem with this scenario is that random victim selection does not provide a guarantee that all \( Y \) workers will be able to discover a stagnant task before all are completed. The double negative outcome will then be that: i) an opportunity for improving performance is lost, and ii) hardware resources where unnecessarily spent without progressing execution.

In summary efficient adaptive work-stealing requires uniform and deterministic placement of tasks in order to effectively determine parallelism fluctuation thresholds and maximally capitalize on the gains of adaptation.

3.5.5 Existing work-stealing task schedulers

This thesis discusses features and capabilities of work stealing runtime schedulers. It does not investigate work-stealing in its entirety, rather specific aspects. All implemented prototypes were built upon already established work-stealing schedulers, with documented performance profiles. This section presents these schedulers.

WOOL

WOOL is a tasking library developed by Karl-Filip Faxén [61]. The programming model involves defining functions and loops to be executed in parallel and includes three operations, call, spawn and sync. The first two call a task, the former sequentially like any function, the latter will place it in the queue to be stolen or executed later. Sync, retrieves the return value of the latest spawned task; if the synced task is stolen but its execution is not finished, the worker will *leapfrog*, try to steal a task from the thief of the task to-be-synced.
3.5. WORK-STEALING

WOOL spawns and steals tasks at very low cost, resulting in very good overall performance; with fine-grained parallel workloads it can achieve linear speedup [60].

This runtime has been the foundation of the experimental implementations in this thesis.

Cilk++ and Intel Cilk Plus

Cilk++ is a tasking library developed by R.D. Blumofe et.al. [24], originating from the Cilk-5 runtime scheduler [66]. Its programming model is very similar to WOOL, although it employs a different scheduling strategy. Upon spawning a task, it is executed immediately by the worker. A continuation to the task that made the spawn is put in the task queue and can be stolen. To that end it uses a cactus stack and requires a custom compiler, because the continuation-passing-style does not adhere to standards.

Recently Intel acquired Cilk++. They developed a new version called Cilk Plus which abandoned the cactus stack for standards compliance. Support for Cilk Plus has been integrated into the Intel C++ Compiler (ICC) with the 2010 iteration and into the GNU C Compiler (GCC) suite as of version 4.8.

Cilk++ and Cilk Plus have been investigated to apply the proposed algorithms in this thesis, but no implementation has been completed as of this writing. However, it is the foundation of other closely related projects.

OpenMP tasks

OpenMP (Open Multi-Processing) is a compiler aided language extension supported by library routines; it provides shared memory parallel programming constructs in C, C++, and Fortran. It has enjoyed much commercial adoption and multi-platform integration primarily due to the primary focus on providing a simple and flexible programming interface rather than performance. OpenMP supports many different constructs for parallel execution, with tasks being one of them since version 3.0. Other include work-sharing and nested-parallelism.

OpenMP task implementation in GCC does not implement a work-stealing scheduler but a centralized queue scheme. Nevertheless, several projects have proposed alternative schedulers [9, 10, 51] as well as work stealing and hierarchical scheduling [120]. The ICC implementation uses such a work-stealing runtime.

The OpenMP tasking implementation supports several quite advanced features which are not straightforward implementable on a work-stealing scheduler due to lack of control. Among them tied and untied tasks, where a task can be tied to a specific thread making it non-stealable even if switched out, while an untied task can be resumed on a different thread. In comparison, WOOL tasks are tied and Cilk++ tasks are untied.

Intel Threading Building Blocks (TBB)

Intel’s TBB is a C++ template library primarily for scalable data parallel programming and nested parallelism through tasks [92, 124, 130, 129]. It employs work-stealing similarly
to Cilk, and some advanced features like data dependency resolution with the use of algorithmic skeletons, flow graphs for improved task placement, speculative locks, a custom distributed memory allocator, and others.

TBB was released in 2006, much before C++11 implemented parallel execution features, and thus enjoys a vibrant community still. It was also one of the first parallelism libraries to fully support C++, resulting in commercial adoption in the software industry.

Other languages

Many established programming languages have included constructs for task-parallelism recently. Since version 1.7, Oracle Java supports the fork/join pattern, using classes in the java.util.concurrent package, namely the abstract ForkJoinTask class; RecursiveAction and RecursiveTask classes are two implementations of the abstract class, differing only in that the former cannot provide a return value. These tasks are processed by a pool workers created from a ForkJoinPool class, employing work-stealing for work distribution.

Microsoft C# also supports task parallelism through the Task Parallel Library (TPL), as of version 4.5. The System.Threading.Tasks package provides the Task and Parallel classes for explicit and implicit execution of tasks respectively. The default scheduling in TPL is work-stealing, although several schedulers are available through the System.Threading.Tasks.TaskScheduler class. Tasks in TPL are also by default untied (borrowing the OpenMP term).

Erlang, a language designed for building distributed software, also uses work-stealing for load-balancing threads. With version OTP R11B, Erlang’s SMP scheduler moved from a single global run-queue to per scheduler run-queues. In Erlang terminology a scheduler is conceptually similar to a worker thread. Erlang supports up to 1024 schedulers active simultaneously. A scheduler will steal work from remote run-queues when idling.
Chapter 4

Operating System Scheduling

4.1 Introduction

This section discusses placement of task parallel processes from the perspective of the Operating System. In the context of two-level scheduling, having the theoretically perfect requirements estimation runtime is pointless if the other side is unable to efficiently utilize said estimations.

The subject of partitioning resources among processes by the operating system has been studied extensively [47, 48, 110, 154, 114, 131, 109, 53, 54, 98]. However, much of that work expects either too much information or too little in the context of shared memory parallel processes. An example of information that is unrealistic to expect is per job execution time; in this project’s scope, jobs are assumed to be very fine-grained tasks and not whole processes; profiling at that granularity would impose prohibitive overhead either as online computation or offline training. Furthermore, two level scheduling can potentially provide information previously considered unattainable, such as the amount of a process’ current or future parallelism.

Dynamic online information on a process’ parallelism is characterized as Instantaneous Parallelism. There has been prior work on scheduling based on instantaneous parallelism, as it has been shown it can lead to misallocation of resources [134] if the requirements estimation mechanism is not tuned adequately. Even assuming an optimally accurate such mechanism, there are several points of inherent uncertainty which can lead to misallocation. Most prominent among them is the estimation interval coupled with the non definiteness in approximating a task’s execution time. A very large interval could potentially miss significant changes in parallelism, a very short interval could force a system to react on insignificant parallelism changes. A second point of uncertainty is the suitability of the resources allocated; it could happen that the memory footprint of a process fluctuates between dominating caches and minimally using them at all. Such scenarios would result in different strategies for selecting task placement.

Threads were invented to distinguish independent activities within a single application; those being either multiple instances of the same code, or separate modules with interleav-
ing execution. However, the recent abundance of processing cores in multicore and future manycore architectures bestows hardware related connotations to threads as well. The operating system’s scheduler will ideally try to place one thread per core; thus the number of threads spawned is indirectly transformed into a request for a certain number of cores.

Current OS schedulers treat application threads as black boxes scheduling-wise [43]; they provide each with equal opportunity to hardware resources, regardless of the application’s actual requirements. Moreover, most parallel applications exhibit suboptimal inter-thread synchronization, due to hardware latencies and contention. Consequently the current process model allows an application to dictate the terms of hardware resource utilization — possibly overcommitting the system and increasing contention — forcing the OS to act only retroactively to hazardous situations, often unsuccessfully [157]. Employing such a scheduling scheme on multicore systems can cause loss of performance or under-utilization of resources; most importantly though, it can lead to arbitrarily non-uniform loss of performance on multiprogrammed systems.

4.2 Basic concepts

This section discusses basic concepts surrounding a system-level scheduler and primary responsibilities therein.

4.2.1 Resource partitioning strategies

The first and foremost responsibility of a system level scheduler is the partitioning of available hardware resources between running processes. There are two primary strategies for this: time-sharing and space-sharing.

Time-sharing is when processes share the same hardware resources consecutively, each for a set interval. This strategy first came to be to provide an illusion of concurrency on single processor systems. Each concurrent computation is given a time-slice — an amount of CPU time — to run before being replaced by another computation for the next slice and so forth. Time sharing is still employed in modern CMP systems to accommodate for the fact that there are usually much more processes and even more threads running than there are physical cores. This imbalance is not going to change even with the introduction of many-core systems. The negative aspect of time-sharing is that switching between processes has a certain overhead which if incurred often enough could be non negligible.

Alternatively, space-sharing places each process on a disjoint subset of hardware resources. Under the assumption that modern CMP systems have more cores than the parallel scaling capabilities of many algorithms, its arguably sufficient to split resources among time critical workloads, leaving non-critical applications to share a bare minimum amount. With exclusive space sharing distribution, processes run uninterrupted as if in isolation on a machine smaller than the one actually present. Apart from a rather reduced amount of application switching on the same core, space sharing cuts down on contention of secondary resources too like the interconnects and caches.
4.2. BASIC CONCEPTS

Time- and space- sharing strategies are not necessarily mutually exclusive. As hinted above modern operating systems employ both heuristically to maximize performance, based on the computational intensiveness of the running processes.

4.2.2 Interrupts

The system scheduler is activated and run periodically. At the point of its activation the running user process is paused. The system scheduler then decides which application to run next and activates it from a similar paused state. This sequence of steps is orchestrated using interrupts, special signals handled by an interrupt handler. For example on a time-shared single processor system, the hardware clock would issue an interrupt at the end of each time-slice; the handler would then pause the running user process and activate the OS scheduler which then issues another interrupt to wake up the selected user process to run. The function of forcibly switching processes is called preemption.

A similar interrupt based approach has been implemented on modern CMP systems to time-share each core. However, on such systems there is a second layer of scheduling which decides placement of threads on cores. This scheduler runs on one of the available cores like any other process, and wakes up on a fixed interval using the same interrupt method.

4.2.3 Context switching

Switching between threads to run on a processing unit incurs a context switch which means removing and storing the state data of the exiting thread, and restoring the state of the entering thread. A context switch has a cost of performance due to running the scheduler and handling the TLB, but also from cache contention.

Elaborate cache hierarchies on modern CMP chips necessitate the use of a translation lookaside buffer (TLB), a fixed-size table of correspondences from virtual to physical or segment addresses, that greatly improves virtual memory address translation. Upon executing a context switch, this table needs to be flushed; the new process starts with an empty table which means that first occurrence of every subsequent translation will miss, resulting in a page table walk either in hardware or software. Both the flush and the subsequent misses generate a performance loss for the application. Context switching between threads of the same process does not require a TLB flush due to them sharing the same virtual address space; the previous entries of the table would be valid for the new thread also.

4.2.4 Thread management

An operating system might support two different types of threads: kernel and user threads. The former type is recognizable by the OS kernel and consequently schedulable by its scheduler. Moreover, userspace implements what is called a user thread. This is visible only in userspace. There can be multiple user threads spawned over a single kernel thread; then the application’s runtime is responsible for scheduling them and handling their state. A user thread cannot be placed on a processor unless a kernel thread is scheduled on it.
already. Hence kernel threads are a representation of hardware contexts in the kernel, while user threads are application contexts running over whatever allotment of processors the application received from the OS through kernel threads. In reality things are not that fluid nor dynamic. In total there can be three different implementation combinations:

- **1:1**: 1 kernel thread for each 1 user thread. This is the default threading implementation on modern mainstream operating systems. Called a *Lightweight Process (LWP)*, each thread can be scheduled and placed on hardware independently by the OS scheduler. Each process starts as a single LWP and every thread spawned by it thereafter is its own LWP. In this configuration the notion of *process* exists only for administrative purposes, primarily to signify that a group of LWPs use a single address space, used in memory protection and management. Since threads are dependent of a kernelspace counterpart, all management actions need to inform the kernel requiring a system call and incurring some overhead.

- **N:1**: N user threads over 1 kernel thread was the default implementation on single processor systems. The application was able to split computation contexts among virtual entities, it then had to manage itself, run over a single processor. The absence of true parallelism made the requirement for multiple kernel threads redundant, as it was only used to store kernel sensitive state. A benefit of this configuration is that thread management tasks happen solely in userspace, without any system call.

- **N:M**: N user threads over M kernel threads is a combination of the above solutions. Such a scheme increases implementation complexity but enjoys the benefit of both parallelism (M kernel threads allow execution on M physical processors) and lightweight user thread management without system calls. This configuration was first introduced by Anderson et. al through *Scheduler Activations* [7].

The fourth possible combination 1:M is of course nonsensical since a single userspace computation context cannot be split among M kernel threads — and in effect M physical processors — at the same time.

### 4.3 System-wide scheduling issues

#### 4.3.1 Selecting resources

The task of the operating system scheduler is to divide resources between all running processes. These resources are assigned directly or indirectly. For example a processing unit is directly assigned by placing computation on it; on the other hand cache banks are indirectly assigned as a consequence of processor assignment. An ideal OS scheduler must take into consideration both groups of resources. In this project we considered only processing cores and caches as schedulable resources, because of targeting primarily compute bound, non I/O bound processes.
Scheduling for the directly assignable resources is simple due to the fact that performance is easily quantifiable and monitored. Processing cores are distinct entities with utilization status measurable in isolation per process. It is relatively easy to count the amount of cycles a thread consumed on a specific core, even if said core is shared by many threads of the same or other processes.

Furthermore, think of CPU time as a brokered commodity; consuming CPU time is a stochastic process happening progressively during the execution. A thread is placed on a core with a promise of a time slice; in other words the commodity is not consumed in advance. Dynamically changing scheduling does not incur having to rearrange CPU time other than notionally switching ownership of the forthcoming slice. Assuming that data are readily available in the cache, context switching incurs relatively low overhead.

Indirectly assignable resources are not as easy to reason about, especially since they are commonly shared by many directly assignable resources and consumed in advance. That means that scheduling mistakes increase contention with potentially a high reparation cost. Once a thread is given access to a certain amount of cache, the scheduler must assume that it is fully utilized and that the owning process depends on it. If a co-tenant is introduced later on — either due to time-sharing or sharing the cache hierarchy from other cores — cache lines will be evicted to clear up space. If the newer co-tenant has passive cache usage it might result in continuous misses if the older tenant thrashes it due to how the replacement policies work.

It becomes apparent that resource partitioning concerns significantly the indirectly assigned resources such as caches. As a result architecture topology should be a major concern.

### 4.3.2 Fairness

Traditionally scheduling fairness in Operating Systems has been defined as *equal opportunity*. Stemming from single processor machines where there was a direct correlation between time-slicing and total execution time, CPU time was divided into equal slices and processes alternated between them to provide an illusion of parallelism. Varying the size of the slice did not have any effect on the performance of the process other than the aggregated overhead of context switching.

On multicore architectures *equal opportunity* induces unfairness. A direct result of how parallel processes synchronize and concurrent entities collaborate is that one process can dominate over the performance of another if both are given uniform access to the hardware. Under the assumption that all running processes are of equal importance, collocating a highly parallel process with a less parallel one means that, if both are given access to the same amount cores, the former is constrained and the latter potentially wasteful. Comparing individual performance degradation, relative to running in isolation, finds the highly parallel process suffering much more than the not so parallel. Alternatively the resources given to each process should be sized proportionally to its true requirements given the constraints of system-wide demand. Such a configuration would result in relatively equal performance degradation among them, arguably a much fairer partitioning.
Our own experiments have shown variable and unfair performance degradation when processes are multiprogrammed on a multicore processor, even when using dynamic adaptive resource management. Figures 4.1 and 4.2 show the percentage performance degradation variance in log-10 scale (Y axis) and the factor of group performance slowdown (X axis) for groups of 2, 3, or 4 collocated processes on a multicore processor. We run 50 total combinations from a set of 6 processes. If the variance is 0, it means that all co-runners degraded their performance equally, which is the desired case here. The group slowdown factor on the X axis refers to the worst slowdown within each group.

We consider high variance as unfairness in resource management. It can be the case where a high variance does not translate into poor performance; thus we also use the worst slowdown within each group as another primary metric for evaluating the effectiveness of resource management. The desired outcome of this work is to provide low variance and negligible slowdown.

For figure 4.1 we used a non-adaptive runtime scheduler. It is obvious that for Linux unfairness is quite high, when several groups resulted in close to 12x process slowdown while variance was around 300%. Using a adaptive runtime as in figure 4.2 drastically improved results, although there is still significant unfairness; there are many groups which suffer a 5x slowdown and a variance of 100%.
4.3.3 Efficiency

Allocation of hardware resources to processes needs to be efficient. If a process is given access to a set of resources, which will then be powered accordingly, the process must be able to sufficiently utilize them. This sufficiency can vary between different systems and use cases. For example, x86 processors could benefit from the use of dynamic frequency scaling, and accept 50% core utilization as sufficient. It is beyond the scope of this project to iterate and define all the possible scenarios; thus for simplicity sufficient utilization is defined as being simply non-zero.

Talking about just utilization is not ideal also. A parallel process can be utilizing a core fully without providing progress to the application’s execution. An example of that is when a worker thread is attempting to steal and choosing victims whose task-queues are empty, maybe because the application’s parallelism is not enough to accommodate the total number of workers.

Consequently, an OS scheduler needs to be able to cost effectively remove resources from processes which are not able to productively utilize them. It is not necessary at this point to define a method for achieving this, but it suffices to accept that awareness of requirements through parallelism feedback is the only requirement. In an arbitrary scenario where a process’ parallelism fluctuates continuously, the OS scheduler should give and remove access to a certain number of hardware resources proportionally to the fluctuation. This would result in the following alternative situations:

1. Threads are destroyed upon losing resources, and new ones created and initialized upon gaining access.
2. Threads are migrated between time-sharing and space sharing configurations based on the amount of available resources.
3. Threads are put to sleep and woken up again when needed.

The above options are sorted by amount of overhead they produce based on their typical implementation as it’s available in modern standard libraries. Thread creation has a significant cost which in some cases might exceed the running time of fine-grained tasks. Migrating threads does not solve the problem of lack of parallelism, as the feedback the process sends is actually in terms of threads and not processing cores; when such feedback says the parallelism has decreased then that means the process can no longer efficiently utilize the amount of threads it currently has. Thus the best solution is to simply put a thread to sleep.

How about the creation of threads? When should a thread be created? When a process is started with $X$ amount of threads only to have the runtime later decide dynamically that it can utilize $N \times X$ threads. The threads should be created on demand but not destroyed, only put to sleep to avoid the same overhead for recurring parallelism spikes; nevertheless it is possible that creation time could exceed the duration of increased parallelism, making not only their creation pointless but also lose an opportunistic scale out. Alternatively the process could bootstrap with as many threads as there are cores but initially use only a
reasonable amount (and wait for the initialization of only that small subset); the rest would be readily available to wake up on demand making the delay from request to service quite small.

In essence minimizing wasteful utilization and handling idleness are important tasks of a system scheduler; however, handling the various parallel entities requires special consideration and the collaboration of the process’ runtime scheduler.

### 4.3.4 Requirements estimation trust

*Two level scheduling* [64] addresses the inabilities of the OS to keep track of per process hardware requirements, and the inability of the processes to infer system-wide resource availability. However, processes have little or no gain from being honest with their estimations. Truthful and thus frequent allotment adaptation potentially reduces contention but involves extra overhead; being greedy potentially increases contention but allows instant access to resources. From a game theoretical perspective, in the simple scenario of two processes A and B, there are the following cases:

**Case 1:** both A and B processes are dishonest.

**Case 2:** only A (or B) process is honest

**Case 3:** both A and B processes are honest

Either process looses the most in the second case, since by being honest it might suffer contention from a dishonest co-tenant plus having the overhead of adapting resources. Individual payoff is maximized in the third case, although there is uncertainty regarding the actions of other processes; hence each must concede case two being the probable outcome of honesty and reject case three completely. Consequently, case one becomes the best strategy for all. This reasoning can be extended to any number of processes running at the same time, without loosing validity.

There has been related work in providing higher incentives for truthfulness in cloud computing [150, 79]. However, those solution impose computational overhead which should not be introduced inside an operating system scheduler, where processes could have sub-second running time making any cycles spent by the scheduler significant.

Other related work has embraced self-adaptation of resources by the process. Self adaptation, besides failing to provide motivation for honesty, is also unaware of contention. On a heavily multiprogrammed system, hazardous contention avoidance might require allotting resources below the desired amount.

### 4.3.5 Scalability: computability and overhead

Theoretical computer science has long used scheduling, job assignment, and bin packing as definitive examples of NP-Complete problems. Optimal scheduling of multiprogrammed systems falls into the same group of problems. A job comes with a set of hard requirements in hardware locality and capacity, potentially a soft or hard completion deadline too. These
4.3. SYSTEM-WIDE SCHEDULING ISSUES

constraints need to be adhered by the scheduler for every job. In a multiprogrammed system it should be expected that \( n \in \mathbb{N} \) such jobs can be introduced simultaneously. Consequently the resulting problem is trivial to reduce to bin packing (using Karp reduction) making it computationally demanding.

Indeterminism stems from the desire for achieving an optimal solution in combination with the multiplicity of the problem. It is the fact that \( n \) jobs need to be scheduled simultaneously which makes an optimal solution deterministically unattainable. Otherwise called Global Optimization \([81, 80]\) this field has enjoyed much research with several computational methods developed \([6, 118]\). Nevertheless they suffer from any and all of the issues listed next, making them unsuitable components in any OS scheduler.

Global Optimization methods might require quantified job characteristics and other information which in reality are unknown or too costly to acquire; online profiling is impossible to perform in a timely manner in the context of fine grained task-parallelism with sub-second task execution time and unknown task-tree depth and width. Learning methods also move slightly out of scope of a general computing OS scheduler as they depend on repeated execution of identical workloads. Moreover, offline profiling of a process before every run is orthogonal to the desired goal for a "just-works fully automated generic scheduling solution". Other methods might simply be too computationally intensive to qualify for use within an OS scheduler. Nevertheless, the types of processes targeted by this project do not have hard deadlines, making suboptimal scheduling solutions acceptable.

An operating system scheduler’s primary goal is to facilitate each process in achieving its best performance, while considering global demand and avoiding contention. However, in the effort to do so, any such scheduler could be working against the process. Every time the scheduler is invoked it steals CPU time from the process in order to decide where and how much further CPU time to give it; if this decision process is not short, even if user time is optimal, the real time might be dismal. In other words any time spent in the scheduler is time lost from progressing computation. An Operating System scheduler should not stand in the way of the process but remain unnoticeable in the background.

4.3.6 Use case 1: a first attempt on the Barrelish OS

Barrelish is a novel approach to the old idea of a distributed operating system. It follows the multi-kernel model \([13]\), according to which there is one micro-kernel per each physical core in the system. These kernels work as one distributed operating system. No data is shared, only copied using message passing; all services are shared by being distributed and their status replicated. This design provides two great features right away: portability — since it can be deployed on any architecture as long as each available processor is supported — and scalability. The latter is an outcome of the OS’ most basic characteristic of assuming no shared memory and employing message passing for inter-core communication. Hence there is no dependence on complicated cache coherence mechanisms.

Thread scheduling in Barrelish is rather simplistic; each kernel uses the RBED algorithm \([29]\) to order the execution of the threads that are time-sharing its underlying core. There is no automated mechanism for migrating threads between cores, since that would
require kernels to stop being independent but partake in a distributed monitoring scheme. Nevertheless, each process is allowed to create threads arbitrarily on a specific malleable set of cores, called the domain, and migrate those threads within the domain on demand. Thus there is no inter-core scheduling done by the system, or in other words there is no centralized common control over the distribution of these threads onto the cores of a program’s domain. So, very frequently the combined load becomes highly unbalanced, without system-wide knowledge and control, occasionally under-utilizing the system.

This project investigated methods for providing system-wide resource management in the Barrelfish OS [146], with a prototype implementation called BIAS. Figure 4.3 depicts a simplistic overview of a running system that uses BIAS. In the figure’s example there are 4 cores and 2 task-parallel processes running; 3 of the cores are split between the processes; the 4th core is time-shared by both processes. The CPU drivers are minimal kernels, one per core and assumed homogeneous in this example.

BIAS implements adaptive scheduling in the application layer, with the application’s runtime forwarding parallelism feedback to a new system-wide scheduler. The system scheduler space-shares cores base on the received feedback. Upon contention (insufficiency of enough cores) threads of different processes are allowed to time share a core, until one of the processes can win exclusive access.

Figure 4.3: A snapshot of the BIAS system-wide resource manager over Barrelfish with two processes sharing all resources and time-sharing one core.
4.3. SYSTEM-WIDE SCHEDULING ISSUES

Although BIAS did provide evaluated positive results, it suffered from several irreparable shortcomings and was thus abandoned eventually. The issues where: i) inability to scale, 2) unfair resource distribution, leading to 3) uncontrollable performance degradation.

Scalability issues were the outcome of having a $N \times M$ problem to solve, $N$ being the aggregated number of threads, and $M$ the amount of available resources. Even though BIAS employed a segmentation mechanism to spit the management of resources into independent sections, the necessity for cross-segment status replication and centralized decision making incurred total delays proportional to the aforementioned complexity. In essence BIAS suffered from the disadvantages of using global optimization.

BIAS distributed resources unfairly resulting in non-uniform performance degradation of co-runners, due to blindly servicing their resource requests. Under heavy system load with no unassigned processors, if a process requested extra cores BIAS would service the request by removing exclusive access of the same amount of cores from another process. As an example, collocating multiple embarrassingly parallel workloads would eventually result in having all cores time-shared between all of them; although this scenario does not deviate from the current reality in OS scheduling, the overhead of thread management combined with the excess small overhead of BIAS would completely outweigh any benefits.

4.3.7 Use case 2: The evolution of the Linux scheduler

In recent years the Linux kernel has gone through three schedulers, each implementing different strategies with distinct trade-offs.

With version 2.4 Linux used the $O(N)$ scheduler which was a straightforward implementation of the time-slicing logic. CPU time was divided into epochs, within which each task could execute up to its individual time-slice. If the epoch’s length was not enough for a task’s time-slice, or for other reasons it was interrupted, then its next time-slice would be 150% in length. During every scheduling event, the $O(N)$ scheduler would iterate the complete list of tasks, applying goodness criteria to order their execution, eventually selecting the one to run next. This scheduling approach suffered from fairness and scalability issues, thus leading to its abandonment.

With version 2.6 came the $O(1)$ scheduler, designed to address many of the previous scheduler’s issues. Primarily it fixed execution complexity of scheduling events improving scalability and efficiency. It was based on the idea of the runqueue, meaning that a scheduling event comprised of a single dequeue operation. Heuristics were employed to identify process requirements, like if they were I/O or compute bound, so as to order them in the runqueue. Nevertheless, these heuristics became large in number, each custom tailored to a specific fine grained pattern, making the scheduler unmaintainable and difficultly extensible.

Linux version 2.6.23 brought a long anticipated scheduler change, in the form of the Completely Fair Scheduler (CFS), which was partly inspired by the Rotating Staircase Deadline Scheduler (RSDL) which was called for inclusion with version 2.6.21. CFS used the self-stabilizing red-black tree data structure to provide fairness, operating in $O(\log n)$;
fairness was defined through the uniformity of a process’ access to the processor, relative to all other processes sharing it. Quickly after, group scheduling was added for improved fairness toward highly parallel applications; fairness was extended to individual tasks instead of only between processes. The virtual runtimes of all tasks spawned by a process are shared across the group in a hierarchy; consequently each task receives relatively the same scheduling time as the rest of its group. Nevertheless group scheduling logic delegates configuration control to the administrator and is far from an automated load balancing method. Its default functionality is not far from a per process fairness model, where all get uniform access to the hardware.

Internally the CFS scheduler functions based on a model of the ideal processor with infinite capacity. A fair clock is run at a floating fraction of the real CPU’s clock speed. Per process time-slice is calculated by dividing the wall time by the number of waiting processes. A process’ waiting time is stored and used as a metric for ranking, where the longest waiting time gets the highest rank. This waiting time represents the time the process would have used on the ideal processor. When a process is scheduled — the one with the highest rank — its stored waiting time value starts decreasing proportionally to the CPU time being spent; concurrently, the waiting time of other processes increases. When any process’s waiting time value becomes the highest such metric in play — higher than the one currently running — the running process is pre-empted and switched out.

With group scheduling the logic above is applied first on a per group basis and then within each group. Thus a process A with multiple times more scheduling entities (like threads) (fig. 4.4a) won’t dominate a process B having much less (fig. 4.4b). Each process will be given a fair amount of CPU time, within which the scheduler will try to be fair in scheduling the individual threads. Practically CPU time will be divided equally between the processes, meaning that the many scheduling entities of process A will be given the same total time as will the ones of process B.

The CFS scheduler provides fairness in the form of equal, uniform access to the hardware; each process gets the same slice. However on modern CMPs, multiprogrammed with parallel applications, this notion of fairness is not always fair. To showcase that, assume that in isolation — unrestricted access to the same hardware with no co-runners — both processes A and B have the exact same execution time X. Lets further assume that the available processors are as many as the scheduling entities of A, while B has exactly half. When co-run, processors will be divided equally between A and B, with the former having two scheduling entities per processor and the latter only one. With some level of simplification it can be argued that B would execute in X time as if in isolation; however, the execution time of A would be degraded significantly by an excess Y time, as shown in figure 4.4c. If processes A and B belong to different users, then only the owner of A perceives any slowdown which is not really fair.

To capture the notion of fairness in scheduling multiprogrammed systems, this project introduces the metric of performance degradation variance (pdv); that is the difference in performance degradation between co-runners. The aim of a truly fair scheduler ought to be to minimize pdv simultaneously with minimizing the performance degradation of each co-runner individually. In other words CPU time should be partitioned so that each application runs as fast as possible with as little contention as possible. To achieve this goal, instead
4.4. **SYSTEM-WIDE SCHEDULING PRINCIPLES**

4.4.1. **Process A** executes in $X$ time with 4 scheduling entities on 4 cores.

4.4.2. **Process B** executes in $X$ time with 2 scheduling entities on 2 cores.

4.4.3. Co-running processes **A** and **B** over CFS.

Figure 4.4: Assume processes **A** and **B**, with 4 and 2 scheduling entities respectively. Co-running then over the Linux CFS scheduler means that Process **B** runs without performance degradation in $X$ time, while **A** in $X + Y$ time. With truly fair scheduling both should need the same $CX$ time to execute, $C > 0$.

of completely uniform access to hardware, processes should be given a slice proportional to their ability to utilize the machine as a whole.

### 4.4 System-wide scheduling principles

Having discussed the major issues with scheduling multiprogrammed systems, this section tries to formulate a set of principles to follow in designing a solution. These are to be seen as abstract results of a thought process and not as a recipe; it might very well be that complete satisfiability of these principles is infeasible. In summary the OS scheduler must provide a *fair* partitioning of resources custom tailored and proportional to the specific hardware requirements of each process; it needs to be diligent in handling idleness and parallelism fluctuations in the processes; to do those two things the scheduler needs to be aware of the requirements of the process but also needs to address the possibility of dishonest such feedback, and also accommodate for processes that do not provide such information at all; finally, said scheduler must be deterministic with negligible computa-
tional overhead while providing sufficiently good solutions. In all practicality this list does look infeasible as a whole; thus much of the following content discusses the threshold for accepting the partial satisfaction of said principles.

In the context of multi-criteria multi-objective optimization where this scheduling problem belongs, no solving method can meet the computability and overhead constraints stated, as they require $O(n^y), y \in N^*$ time [119]. Considering many-core architectures of one hundred cores or more, multiprogrammed with tens of processes, such complexity becomes prohibiting. It could be argued that fine grained processing of tasks is likely to be computationally lighter.

Some of the constraints need to be relaxed, either by accepting suboptimal solutions, increasing the scheduler’s computation window, or restructuring the problem so that indeterminism lies where it is not as significant. The first two alternatives are opposed to the initial objectives, meaning the only viable option is to restructure the problem. The multiple scheduling criteria are inherent to the problem and cannot be changed; however this is not true for both the objectives and the computation method. Currently the problem at hand has been considered as a single problem to be solved in one pass as a whole. Moreover the scheduling objectives as listed above seem intertwined, but in reality they don’t have to be. System-wide scheduling can be restructured to address a certain list of objectives asynchronously and independently of one another. Below we present an abstract reformulation but postpone discussing an actual solution till chapter 6.

- **Fairness**: given limited resources, all simultaneously running processes should sacrifice no more than the same percentage of their expected performance.
- **Efficiency**: the amount of underutilized resources of processes should be minimized.
- **Distrust**: all processes must be required to prove their requirements in the presence of contesting co-runners
- **Cost-Effectiveness**: scheduling actions should execute to completion in time significantly smaller than the desired scheduling interval
- **Near-Optimality**: schedules must satisfy criteria enough to evolve to optimal in one or two more iterations.

Expected performance is defined as the execution time of the same workload run in isolation with unrestricted access to all available hardware resources. Multiprogramming could potentially result in processes being allotted less resources than they could utilize, degrading their performance.

The first criterion is concerned with the ability of the resource manager to award each process with enough resources, so that the percentage of performance degradation is the same for all co-runners. The second criterion encourages conservation by allotting only the necessary amount of resources, allowing to power-off what remains idle. The third criterion demands that requests get satisfied over time but not immediately; processes should be given a promise of a certain allotment and an opportunity to win its realization; the inherent delay produced by such method could be a significant overhead under high system
load, thus the mechanism implementing it should be lightweight, computationally simple, and deterministic. The last two criteria concern the implementation and were discussed in the beginning of this section.
Part II

Solution Space
Chapter 5

Adaptive Work-Stealing Scheduling

5.1 Introduction

This section investigates adaptive work-stealing scheduling of fine grained task parallel applications. As summarized in the first chapter, adaptation requires a method for estimating available parallelism. Such estimations are mapped as number of worker threads, which directly translates into processing cores. The placement of these threads onto specific cores indirectly selects caches too.

For example an application whose tasks have large memory footprints, might benefit by not sharing caches between workers to avoid capacity conflicts. On the other hand an application doing multi-step processing of shared data might gain the most by sharing caches.

Conclusively appropriate selection of cores for placing the workers determines the placement of data onto the caches. However, in the context of two-level scheduling the application’s runtime should not handle thread placement or any other aspect of thread scheduling whatsoever; any application is rather inept to do so. Its primary responsibility — other than the placement of tasks — must be to estimate the number of threads optimally utilizable and forwarding said metric to the system level.

The estimation method should adhere by three principles:

1. the metric of parallelism must be history independent and portable thus hardware agnostic,
2. the decision policy must be optimistic by loosely defining the requirements’ upper bound, and opportunistic by capturing sudden short lived changes in parallelism,
3. the computation method must be scalable by needing a monotonically reducing size of profile data, eventually converging to a fixed size irrespective of further parallelism increments.
5.2 Parallelism quantifying metrics

The cornerstone of every requirements estimation mechanism is the metric used to quantify those requirements. It must reflect available parallelism, and in the context of adaptive scheduling, it must reflect future requirements. The former translates into a workload’s ability to fully utilize a certain number of processing cores simultaneously. The latter demands identifying an application’s parallel computation before it becomes ready for execution. Both of these criteria will be referred to as parallelism and potential respectively, or an application’s parallelism potential as a whole.

For an automatic online estimation mechanism that does not increase the development effort, identifying parallelism must not be delegated to the developer. Thus the mechanism must be completely decoupled from the application, encapsulated fully in the runtime. From that perspective there are two strategies for profiling an application; measure its impact on the hardware it is running on, or leverage the semantics of the programming model. ASTEAL by Agrawal et al. [3, 4] uses a hybrid of both, measuring the hardware effects of certain scheduling actions.

Many other projects have explored profiling the effects of execution on the hardware to infer application behaviour. Some predictively model cache usage patterns [28, 22, 32, 49], or monitor cache events mapping them to application behaviour [106, 11, 104, 136, 137], or leverage both for contention-aware scheduling [86, 63, 34]. SCAF [43] leverages performance monitoring and runtime adaptability feedback to partition resources. Iancu et.al. [83] explored resource oversubscription to address fluctuating requirements. Weng and Liu [151] used CPI\textsubscript{mem} to estimate resource requirements, as feedback for improving resource partitioning.

The biggest obstacles of depending on hardware counter measurements are portability and interpretation. Various processors implement different events, or the same events differently; this makes it hard to formulate a proper ever compatible foundation for an estimation mechanism; it would require constant refinement and improvement. Furthermore, even if measurements of the same specific hardware events can be acquired transparently, interpreting them as application behaviour cannot be concretely unambiguous. For example an increase in CPI is considered a good indication of increased cache misses; nevertheless, it cannot be easily inferred if it is by design (the application starts working on data previously not touched) or not.

In [155] Zhang et. al argue that their CPI based approach overcomes the variability of CPIs in workloads after gathering a extensive history of data. In [105] Mars et. al argue that their characterization method needs prior knowledge on the behaviour of the applications being profiled and training over the target architecture; both tasks cannot be performed in the context of mainstream computing.

This thesis has explored a completely hardware agnostic metric, leveraging work-stealing semantics by using a workers task-queue size as parallelism potential metric. Tasks placed in a worker’s task-queue are ready for execution but not assigned yet; since they are ready, all dependencies have been resolved; thus they directly correspond to future computation, executable in parallel. Tasks are logical, not bound to hardware. The same metric has been explored by Cao et. al with promising results [30].
5.3 Deciding allotment size satisfiability

A proper parallelism potential metric must be accompanied by a concrete policy for deciding the satisfiability of the given allotment size. By allotment we define the set of resources that have been already assigned to the application. In the context of a work stealing scheduler that is the number of worker threads. This decision policy should comprise of at least the following three states.

- **Underutilized**: Workers are not being utilized enough; allotment size could decrease without affecting performance,
- **Overutilized**: Workers are utilized fully, and there is more parallelism available. Allotment size could increase without affecting worker utilization levels,
- **Balanced**: Workers utilized and no excess parallelism detected.

This policy is generic and complete; it includes all possible states an allotment can be in, for prompting an adaptation action, or inaction if balanced. Moreover, these are the states that any selected metric’s values should map onto.

Labeling an allotment with any of the three aforementioned states, implies uniformity of utilization across all workers. An allotment’s status when individual workers are split between over- and under- utilization levels is undecidable. Such cases could be address by accepting a majority based decision. If more than half the workers are overutilized, more should be added. However what if the rest are not simply underutilized but even starving? This scenario could result in significant wastefulness of CPU time. It could be the case that the total parallelism is not enough for the current allotment’s size if the over-utilization is borderline, negating any usefulness from adding extra workers.

From the perspective of evaluating worker utilization status, a fundamental issue of the work-stealing is its inherent randomness. This randomness disallows any correlation between a specific worker and its ephemeral utilization level. The lack of task placement control means that any quantification can be performed for an allotment as a whole, with no guarantees on per worker consequences.

5.3.1 ASTEAL

ASTEAL is discussed in detail as it was selected as a good representative of a related but different approach in requirements estimation. Furthermore, ASTEAL was fully implemented and comparatively evaluated relatively to the contributions of this thesis.

ASTEAL monitors the cycles spent by workers conducting actions that do not help progress execution. In the context of work-stealing that is stealing time, including both failed and successful attempts, but excluding the execution of tasks. The argument is that if the time spent on these actions surpasses a certain threshold — marked as inefficient — it signifies absence of parallelism and the amount of workers should be decreased. ASTEAL measures satisfiability on a second dimensions, keeping track of previously requested allotment size and comparing it to the current size; if matched then the request is satisfied, otherwise deprived. The specific decision policy of ASTEAL is as follows.
• **Inefficient**: decrease allotment. The workload is unable to fully utilize its allotted workers.

• **Efficient and Satisfied**: increase allotment. The workload was allotted the desired resources and successfully utilized them.

• **Efficient and Deprived**: unchanged allotment. The workload was allotted less than the desired resources and successfully utilized them. The system is probably congested.

Parallelism quantification is done on the allotment as a whole. On every interval the aggregated wasteful cycles are compared against the adjusted length of the that interval. Thus any conclusion has lost information on the distribution of utilization levels across the allotment.

The above policy does not directly correlate to our initial policy. It misses a balanced state. A hidden detail is that the **Efficient and Deprived** state still corresponds to an increment request, repeating the previously unsatisfied request. The absence of a balanced state means that ASTEAL is over-optimistic; it will pro-actively request more resources than it has proof it can utilize; the added worker remain unsatisfied resulting in an immediate removal. This unnecessary allotment changes are wasteful. Moreover they risk missing an actual parallelism increment. Although ASTEAL can be argued to be opportunistic, its unprompted increments can lead to inaccuracy and eventually loss of performance.

Another downside with ASTEAL is that its metric quantifies past behaviour. Although the authors claim that their algorithm does not correlate job history with future requirements [4], it is obvious that any decision is provoked by effects during the previous interval. Consequently ASTEAL could fail to address a sudden short-lived changes in parallelism, or in other words it is not opportunistic.

### 5.4 Computing estimation requirements

After having cover basic principles for quantifying parallelism and deciding satisfiability, it remains to define a basis for parsing those data. On one hand the method must be arbitrarily scalable. Exhaustively profiling every worker thread translates into a monotonically increasing dataset relatively with the allotment size. This attribute means that a straightforward parsing algorithm would suffer monotonically increasing execution time.

A common way for bounding the execution time would be to profile only a sample of the workers, or process only a sample of the data set. In either case the work-stealing algorithm does not provide a way to score candidacy among workers. The primary problem is the randomness of the scheduling algorithm. Due to random victim selection, worker behaviour is not guaranteed to persist. Hence deciding the sample would have to be done on every interval. This requirements adds an extra monitoring level, other than the profiling data themselves.

This thesis chose to follow the path of changing the work-stealing algorithm to provide persistent worker behaviour, which in turn can be used to cap the sample size, without altering the algorithm’s semantics.
A second part needing special attention is the application of those requirements. Specifically the removal of workers. Adding new workers is an unproblematic action. New threads are created and placed on the added cores. The necessary bookkeeping is straightforward. Decrementing allotment size however, needs to elect specific workers for removal. One might argue that the most underutilized workers should be the ones selected. Once again work-stealing’s randomness mandates to iterate the status of each worker every time this selection needs to be performed. At an arbitrarily large scale this overhead could become prohibitively non-negligible. Considering also that this task cannot be easily distributed, data need be transferred across the processor topology, possibly off chip.

5.5 DVS: Deterministic Victim Selection

5.5.1 Description

Deterministic Victim Selection (DVS) was developed in order to overcome the issues discussed in the previous sections. It fundamentally changes the work-stealing scheduling logic guaranteeing predictable uniform distribution of tasks across workers, by removing all randomness; victim selection is a self-contained part of the work-stealing method, thus replacing said mechanism in any such runtime does not impose further implications. Moreover DVS does not affect performance. Experiments conducted with DVS in WOOL and Cilk++ have shown the same or better performance than the original versions of those libraries.

DVS does not include a mechanism for monitoring allotment status, nor estimating requirements; however its properties enable such actions as discussed in a later section. This section introduces and describes DVS in an informal way. Section 5.5.2 formally defines DVS and mathematically proves all of its properties.

![Figure 5.1: DVS virtual mesh grid topology for 64 physical cores (a). Every application has at least one thread called the source (b).](image)

The DVS algorithm is deterministic. It uses specific strict rules to redefine the set of possible victims for each worker, thus removing all randomness. Applying DVS requires worker threads to be pinned on their respective cores. Ideally there should be only one worker per core. In effect the pinned workers define a metric topology based on the underlying interconnect. DVS however does not use that topology; it creates a virtual 2
CHAPTER 5. ADAPTIVE WORK-STEALING SCHEDULING

dimensional topology resembling a mesh grid, as in the 64 core arrangement of figure 5.1a. Every application has at least one worker, which will be referred to as source worker. DVS marks that worker in a central place in the topology, as in figure 5.1b.

![Figure 5.1a: The allotment consists of 5 workers.](image1)
![Figure 5.1b: The allotment consists of 13 workers.](image2)
![Figure 5.1c: The allotment consists of 9 workers.](image3)

Figure 5.2: Placement of workers is done in spiral sequence horizontally and vertically from the source, starting on top

Every worker added thereafter is placed on the grid, adjacent to the source horizontally or vertically, in a spiral sequence. Figure 5.2a shows an allotment of 5 workers and 13 in figure 5.2b. The virtual diamond shape being constructed does not need to be complete for DVS to work. As in the allotment of figure 5.2c with only 9 workers.

The virtual topology is a non euclidean 2 dimensional metric space, where nodes are connected vertically and horizontally, but not diagonally nor by wrapping around the edges. The distance between nodes is computed as the least number of hops between them and equals the hop-count. As shown in figure 5.3, assume horizontally adjacent nodes x and y; their distance is 1. Now assume node z on top of y; its distance from x is 2. Moreover, distances are mentioned relative to the source. Assuming the source to be 0, saying worker at distance 1 means a worker 1 hop away from the source. Hence workers at distance 2 are further away, and closer to the source means constitute further inside in the allotment.

![Figure 5.3: Node hop-count horizontally and vertically defines their distance.](image4)

The position of a worker on the virtual topology is irrelevant to its physical position and also malleable. Workers are placed in order using a spiral sequence that maintains the same distance from the source, starting directly above the source (although this detail is
just a convention without significance) and jumps a further distance 1 when a diamond ring is completed, until no more workers are available in the allotment. Assuming an adaptive allotment that had worker \( x \) on core \( X \) removed, and later replaced by worker \( y \) on core \( Y \); worker \( y \) will take the place worker \( x \) on the grid. If worker \( x \) is later re-added, it will take a new place. Workers acquire properties according to their position relative to the source. Since their position can change, the rest of this section discusses properties by referring to workers as nodes, meaning any worker located at a specific position relative to the source.

As mentioned earlier, the source worker is the thread which initiates the application. The rest of the workers are unambiguously separated into certain classes. The classification is defined by the location of a worker in respect to the source. There are three classes: \( X \), \( Z \), and \( F \). The following list describe these classes, where precedence is held by former rules. Figure 5.4 illustrates these classes over a 64 core topology, assuming a 25 worker allotment.

Figure 5.4: 25 worker allotment, classified as per the DVS classification, over a virtual mesh grid topology.

- Class \( Z \) includes those workers located at the single maximum communication distance from the source.
- Class \( X \) includes those workers on the axes that span horizontally and vertically from the source, excluding those at maximum distance.
- Class \( F \) includes the remaining workers.

The DVS classification controls victim selection, by defining the potential victims for each worker. Each worker has a predefined victims set, which changes only if the allotment changes. Members of each class construct their victims set differently. In general stealing is allowed only between adjacent nodes, with few exceptions up to the hard limit of a 2 distance. Also each thief iterates its victims set in a strict order. The order is defined by the relative position of the victims to the thief. Hence DVS enforces a directed exchange of tasks between predefined pairs. In the end the DVS rules create a closed flow of tasks.
across the virtual topology. Each class performs a different function toward realizing that flow. A summary of the actual rules is as follows.

- Nodes on the axes (class \(X\)) are responsible for transferring load away outward, by stealing primarily from victims nearer to the source.
- Members of class \(F\) relocate load back inwards, by stealing primarily from nodes further away.
- Members of class \(Z\) balance the load across quadrants. Workers in \(Z\) first steal from within their own class (diagonally left and right); only upon failing that, they’ll search for new tasks from the inner parts of the allotment.

The three classes do not exist at all times. For example an allotment of up to 5 members consists of the source and an \(X\) class, as shown in figure 5.5a. Adding nodes up to 13 will introduce class \(Z\); as shown in figure, 5.5b, all nodes on the outer ring are classified as \(Z\) including those on the \(X\), as per the higher priority of \(Z\) classification rule. Class \(F\) is introduced after extending the allotment with nodes at distance 3 and further (figure 5.5c-d).

![Figure 5.5: Stealing relocates tasks; reimagine it as a flow of the load through the topology. Each arrow abstractly represents the directed relocation of tasks from the victim to the thief. White for outward flow and gray for inward reflow. Subfigures b-d show only one quadrant to avoid clutter.](image)

### 5.5.2 Formal definition

This section formally defines the DVS policy. However, several properties are not included in this section; they are presented as part of the requirements estimation algorithm that makes use of them.

**Definitions**

The communication distance between any two workers \(w_i\) and \(w_j\) is referred to as the **hop-count** (\(hc(w_i, w_j)\)); this is the shortest communication path between the physical cores where the two worker threads are located. The allotment of workers for a specific workload is defined as \(I\), and its source worker as \(s\). The maximum hop-count between the source \(s\) and any other worker in \(I\) is defined as \(d\).
Worker Classification

DVS classifies each worker, member of $I$, based on the following rules:

**Class Z** is defined as the set of workers at distance $d$ from the source.

$$Z_n = \{ w_j \in I : hc(w_j, s) = n \}$$

**Class X** consists of those workers that are neighboring only one worker at one less hop from the source.

$$X = \left\{ w_j \in I : \exists! w_i \in I : hc(w_i, w_j) = 1 \land \frac{hc(w_j, s)}{hc(w_i, s)} = hc(w_i, s) + 1 \right\}$$

**Class F** consists of the remaining workers excluding the source $s$.

$$F = I \setminus (X \cup Z \cup \{s\})$$

Victim selection rules

Workers in each class construct their victims set according to the following rules.

**Definition 1 (Victims set).** Each worker $w_i \in I$ can steal tasks from a subset of workers called the victims set $V_i$. Members of each class populate this set differently. It is defined as the union of two independently defined sets $V_a$ and $V_b$. The former is equal for all workers and includes the immediate neighbors (at distance 1). The rules defining the second set depend on the owner’s class, its distance from the source and the value of $d$.

$$V_i = V_{ia} \cup V_{ib}$$

$$V_{ia} = \{ w_j \in I : hc(w_i, w_j) = 1 \}$$

$$V_{ib} = \begin{cases} \emptyset, & w_i \in F \\ \emptyset, & w_i \in Z : d = 2 \\ \{ w_j \in Z : hc(w_i, w_j) = 2 \}, & w_i \in Z : d > 2 \\ \{ w_j \in X : hc(w_i, w_j) = 2 \}, & w_i \in X : hc(w_i, s) = 1 \end{cases}$$

It is important to note that when $d$ is 1, classes $X$ and $Z$ conceptually overlap; thus class $Z$ is introduced only when $d$ is 2 or more. Similarly, class $F$ exists only when $d$ is 3 or higher. Finally, in the special case of a one-dimensional topology (all cores placed in a row), $F$ can never be introduced but is also not needed.

Victims prioritization

The victims set is an ordered set according to the following ordering rules.

**Definition 2 (Victim prioritization).** We define a partial order $P(v_j \in V_i)$ where $v_j \in V_i$, where $V_i$ the victims set of worker $w_i$ in allotment $I$ as follows.
For members of class $X$

1. Priority increases inversely proportional to the victim’s distance from $s$, thus giving higher priority to victims located closer to $s$ that $w_i$.

For members of classes $Z$ and $F$

1. Priority increases proportionally to the victim’s distance from $s$, thus giving higher priority to victims located further away from $s$ that $w_i$.

The ordering rules between members of $X$ and $Z \cup F$ are opposed. The former prioritize victims closer to the source, while the latter those further away. These rules allow certain ordering conflicts as there can be multiple victims at the same distance from the source. These are intentional as there is no benefit by resolving them a certain way.

As mentioned, members of class $Z$ steal primarily from within their class. Thus they will exhaustively process a subtree before proceeding to another. In the same spirit, members of $F$ steal primarily from $Z$, thus they have a higher probability to steal leaves or small subtrees; also $F$ will quickly take higher up tasks from $Z$, meaning that the latter nodes will mostly be processing leaves. However, $F$ nodes can steal from $X$ too, resulting into stealing large subtrees. DVS was designed this way to overcome the major adaptation issue of choosing nodes to remove without having to monitor their utilization. This property is discussed in section 5.6 focusing on adaptation.

To adequately support that claim, it is important to measure the ratio between victims and thieves per node. Assume a complete diamond shaped allotment, of maximum distance at least 3 to include all classes. Nodes in $Z$ are divided into the peak (top of each quadrant) and the sides nodes. A peak node has a total of 3 possible victims, 2 in $Z$ and one in $X$. A side node has a total of 4 victims, 2 in $Z$ and 2 in $F$. The same nodes are thieves to them too, with nodes in $Z$ and $F$ primarily stealing from $Z$. Thus a random $Z$ node will have at least 3 other nodes primarily stealing from him. Let that node steal a task root of a large task subtree; its descendant tasks will quickly be stolen back. If there are more than 3 children, that node will continue with a child, moving down on the task-tree. If not, it will primarily try to steal from $Z$ or $F$, where chances are that it will get a tasks even further down on the task tree. Conclusively, even if a member of $Z$ steals a task high up on the task tree, the descendants will quickly be disseminated inward again, leaving that node to work mostly on small subtrees and leaf tasks.

5.5.3 Implementation

This section presents pseudocode for implementing DVS. It consists of algorithms for constructing each node’s victims set per DVS class. The constrained DVS victims sets have a maximum size of 8 elements and an average of 5. Hence the maximum memory complexity of implementing DVS is $\Theta(N)$, and $O(N^2)$ when $N \leq 8$. 
Algorithm 3 DVS Classification: Main function to construct the victims sets of all nodes.

1: function DVSInitAllotment(Allotment)
2:   idx ← 0
3:   DVSInitS(Allotment[0])
4:   for 1 < i < d do
5:     lx ← 1
6:     for 0 < j < d do
7:       idx ← idx + 1
8:       if idx == ||Allotment|| then return
9:       DVSInitV(Allotment[idx], lx, i, False)
10:      DVSInitX(Allotment[idx], i)
11:     DVSInitVO(Allotment[idx], lx, i, True)
12:     lx ← lx + 1
13:     for 0 < k < i − 1 do
14:       idx ← idx + 1
15:       if idx == ||Allotment|| then return
16:       DVSInitV(Allotment[idx], 0, i, False)
17:      DVSInitV(Allotment[idx], 0, i, True)
18:   end for
19: end for
20: end for
21: lx ← 1, lf ← 0
22: for 0 < j < d do
23:   idx ← idx + 1
24:   if idx == ||Allotment|| then return
25:   if d == 0 then
26:     DVSInitV(Allotment[idx], lx, i, False)
27:    DVSInitX(Allotment[idx], i)
28:   lx ← lx + 1
29: else
30:   DVSInitZ(Allotment[idx], d,(lx == 1))
31:  DVSInitV(Allotment[idx], lx, d, False)
32:  lx ← lx + 1
33: end if
34: for 0 < k < d − 1 do
35:   idx ← idx + 1
36:   if idx == ||Allotment|| then return
37:   DVSInitZ(Allotment[idx], d, False)
38:  DVSInitV(Allotment[idx], lf, d, True)
39:  lf ← lf + 1
40: end for
41: end for
42: end function
Constructing Victims sets

Implementing the DVS classification is conducted via the construction of each class’ victims set, since it has no other implications. Algorithm 3 is presents the primary function which iterates all nodes, calling a special function for each different class. The total iterations of this algorithm are \( N \), thus time complexity is \( O(N) \) with a \( \Theta(1) \) space complexity. The input array \( \text{Allotment} \) is assumed to hold all nodes consecutively, with 0 being the source. The input array is iterated sequentially and each element is placed on the virtual DVS topology in a spiral sequence, starting north of the source.

**Algorithm 4 DVS Outer Victims**: Find a node’s outer victims.

1: function \( \text{DVSInitVO}(\text{Node}, i, \text{hc}, \text{inX}) \)
2: \( v \leftarrow \text{Node}.\text{idx} + (4d + i) \)
3: if \( v \in \text{Allotment} \) and \( v \neq \text{Node}.\text{idx} \) then
4: \( \text{Node}.\text{Victims} \leftarrow \text{Allotment}[v] \)
5: end if
6: \( v \leftarrow v + 1 \)
7: if \( v \in \text{Allotment} \) and \( v \neq \text{Node}.\text{idx} \) then
8: \( \text{Node}.\text{Victims} \leftarrow \text{Allotment}[v] \)
9: end if
10: if inX then
11: \( v \leftarrow v + (i == 0?1 : 0) \ast (4d + i) - 1 \)
12: if \( v \in \text{Allotment} \) and \( v \neq \text{Node}.\text{idx} \) then
13: \( \text{Node}.\text{Victims} \leftarrow \text{Allotment}[v] \)
14: end if
15: end if
16: end function

**Algorithm 5 DVS Inner Victims**: Find a node’s inner victims.

1: function \( \text{DVSInitVI}(\text{Node}, i, \text{hc}, \text{inForZ}) \)
2: \( v \leftarrow \text{Node}.\text{idx} - 4 \ast (d - 1) - (d == 1?i : 0) \)
3: if \( v \in \text{Allotment} \) and \( v \neq \text{Node}.\text{idx} \) then
4: \( \text{Node}.\text{Victims} \leftarrow \text{Allotment}[v] \)
5: end if
6: if inForZ then
7: \( v \leftarrow v - 1 \)
8: if \( v \in \text{Allotment} \) and \( v \neq \text{Node}.\text{idx} \) then
9: \( \text{Node}.\text{Victims} \leftarrow \text{Allotment}[v] \)
10: end if
11: end if
12: end function
5.5. DVS: DETERMINISTIC VICTIM SELECTION

Algorithm 6 DVS source Victims set: Construct the source node’s victims set.

1: function DVSInitS(Node)
2:     for $1 < v < \min(\|\text{Allotment}\|, 5)$ do
3:         Node.Victims ← Allotment[v]
4:     end for
5: end function

Algorithm 7 DVS Core X victims set: Construct the core $X$ victims set extension.

1: function DVSInitX(Node, hc)
2:     if $hc > 1$ then return
3:     end if
4:     $v ← Node.idx$
5:     loop
6:         $v ← v + 1$
7:         if $v > 4$ then
8:             $v ← 1$
9:         end if
10:        if $v == Node.idx$ then
11:            break
12:        end if
13:        if $v ∈ \text{Allotment}$ and $v \neq Node.idx$ then
14:            Node.Victims ← Allotment[v]
15:        end if
16:     end loop
17: end function

Algorithm 8 DVS Z Class victims set: Construct the $Z$ class victims set extension.

1: function DVSInitZ(Node, hc, isPeak)
2:     $v ← Node.idx + 1$
3:     Node.Victims ← ∅
4:     if $v ∈ \text{Allotment}$ and $v \neq Node.idx$ then
5:         Node.Victims ← Allotment[v]
6:     end if
7:     $v ← Node.idx + (\text{isPeak} ? 1 : 0) * 4 * hc - 1$
8:     if $v ∈ \text{Allotment}$ and $v \neq Node.idx$ then
9:         Node.Victims ← Allotment[v]
10:    end if
11: end function

For every node, functions DVSInitVI (alg. 5) and DVSInitVO (alg. 4) will populate their victims sets with the node’s inner (closer to the source) and outer (further away from
the source) victims at distance 1. For classes $X$ and $Z$ there are special extensions of more victims at distance 2; these are found and placed in the node’s set by functions $DVSInitZ$ (alg. 7) and $DVSInitZ$ (alg. 8) respectively. Finally, function $DVSInitS$ (alg. 6) populates the source’s victims set, which includes the 4 adjacent nodes in class $X$. These nodes will be at array positions 1 till 5. All aforementioned helper functions have $\Theta(1)$ time and space complexity.

5.5.4 Discussion and Evaluation

This section discusses the beneficial properties and limitations of DVS as commentary to a rigorous evaluation process.

For evaluating DVS we have used several applications from various sources, faithfully ported to the WOOL work-stealing scheduler. Among them are FFT, nQueens, Sort and Strassen from the BOTS benchmark suite [52], selected as distinctive and popular examples of specific workloads. Their parallelism profiles range from the fine grained with a wide and balanced tree nQueens, to the quite irregular and coarser grained Strassen. FFT and Sort are thrashing the caches with the latter also being irregular. We have also included some micro-benchmarks; recursive Fibonacci (Fib), is embarrassingly parallel and rather finely grained which makes it scale linearly; Stress strains the runtime by varying the grain size; Skew is an adaptation of Stress producing a unbalanced task tree. nQueens, although highly parallel comprises multiple tasks of varying granularity, scaling sub-linearly with a small cut-off.

We run our experiments on two different platforms. One simulated and on real hardware. In the Simics v4.6 simulator we modeled an ideal parallel platform, running the Barrelfish OS. Simics is a full system simulator [103]. We have modeled a 32 core, 8x4 mesh topology where each instruction takes one cycle, including memory operations. The simulated model purposefully does not include a memory-hierarchy to isolate the behaviour of the estimation algorithms. The simulated results should not be correlated with those on real hardware. They are added to support that performance gains are not due to caches or other hardware related indeterminism.

Barrelfish is based on an interesting design which might be well suited for manycore architectures; it provides scalability and portability [133], major benefits when combined with widely distributed architectures [14], like most tiled manycore prototypes [101, 2, 71]. Also, it has no migration of execution between cores making worker threads pinned by default, while also allowing to execute programs in true isolation of other processes even the OS. All system services and auxiliary functions were executed on cores 0 and 1 which were never allotted to our test programs. The test environment of the two schedulers is controlled and the victim selection algorithm is the only difference in implementation; thus it can be argued that the adaptive scheduling is the only factor responsible for changes in the workload’s behaviour.

The second platform is based on Linux (v2.6.32) running on real hardware. The architecture is Opteron 6172 (AMD Magny-Cours) ccNUMA system with a total of 48 cores. There are 4 sockets, holding 2 NUMA nodes each. A node has 6 processing cores and 8GB
5.5. **DVS: DETERMINISTIC VICTIM SELECTION**

![Figure 5.6](image)

Figure 5.6: Two example allotments on two different architectures, classified as per the DVS rules. (a) shows a 27 worker allotment on a 8x4 mesh grid. All classes are incomplete. (b) shows a 35 workers on a 8x6 mesh grid, with core 28 being the source. Classes X and Z are incomplete.

<table>
<thead>
<tr>
<th>WORKLOAD INPUT DATA SETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td><strong>BFish</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Linux</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7: Workload input data sets for Barrelfish and Linux. Input presents parameters, while the cut-off controls the maximum recursion depth. Comma-separated values correspond to multiple parameters. When multiple parameters are required, they are given separated with commas.

of locally controlled RAM. Threads were pinned using pthread affinity while the system was running a minimum of necessary services.

Our implementation is based on an established work-stealing scheduler, WOOL. We implemented and evaluated two versions of our scheduler. One being the original WOOL implementation [59] (WOOL-LF) that employs semi-random victim selection. For the second version (WOOL-DVS) we replaced only the victim-selection algorithm with DVS; the structure of the workers, the task-queues, spawn, sync and steal operations where left as is.

On both platforms we tested allotments of varying sizes; namely 5, 12, 20 or 27 cores for Barrelfish and 5, 13, 24, 35 cores for Linux. It is important to note that due to the topology’s geometry, these allotments are not complete in respect to the classes as defined by DVS. Figure 5.6 visually presents the classification for the largest allotment on each platform.

The input dataset used for each program can be viewed in table 5.7. The input field corresponds to basic parameters, while the cut-off controls the maximum recursion depth and has a significant impact on the produced parallelism. We have selected small cut-off values to controllably induce worker starvation and diversify the parallelism profiles available: constrain the amount of tasks generated to impose worker idleness and adaptation possible. On the Linux platform we used larger inputs to minimize the effect of interfer-
Benefits

The benefits of DVS — assuming a task tree of sufficient depth run by a help-first work-stealing scheduler — are: i) reduced worst first steal latency, ii) uniform worker utilization, iii) inward load concentration, and iv) same or better performance. In depth explanation of these properties requires the formal definition of DVS, which has not been covered yet and is available in section 5.5.2. However, the rest of this section will try to explain the underlying ideas, hinting to the rule specific implications.

The first positive property of DVS is the reduction of the worst first successful steal latency across all workers. The initial phase of executing a task-parallel application is always the same; the master worker has spawned some tasks and all other idling workers need to discover it for successfully stealing. In the meantime they will be picking other workers, whose task-queues are empty, resulting in failed steal attempts. Some workers might be lucky (or optimized) to find the master worker fast, quickly spawning more tasks and increasing the number of effective victims; at some point in time the allotment converges and all workers have managed to find work. The time a worker spends before its first successful steal is called first steal latency. The average and worst such latency is an insightful metric for evaluating a victim selection algorithm.

Reducing the worst first steal latency means that all workers get utilized sooner, contributing more to progressing application execution; consequently there is a higher chance for the first task stolen to be the root of a much larger subtree; in turn this provides a worker with more locally spawned tasks to execute. If the available parallelism is much more than the allotment size, these workers will be running sequentially sooner. No stealing translates into immediate performance gains.

Figure 5.8: First successful steal latency, normalized to WOOL-LF (as 100%). FIB, Skew and Stress provide high parallelism very early, favoring random victim selection. DVS reduces the worst latency.

Figure 5.8 shows experimental results of average and worst first steal latency. DVS exhibits considerably worse average results for the very parallel benchmarks (FIB and Stress) while it is on par with RVS for all other. However, the worst first steal latency is measurably smaller with DVS, especially for non highly parallel and irregular applications.
5.5. DVS: DETERMINISTIC VICTIM SELECTION

Figure 5.9: Successful steals, normalized to WOOL-LF (as 100%) per allotment size. DVS results in less steals with fewer workers; thus workers are able to steal tasks higher on the task tree, running sequential thereafter (stealing from themselves).

Figure 5.10: Per worker useful execution time for Sort on real hardware. Useful is the aggregated time spent processing and successfully stealing tasks. The first worker is the source which includes some initialization.

Completely uniform worker utilization for a work stealer means that no time is spent on failed steal attempts. Thus all workers continuously execute application code and terminate at the same time. in that respect completely uniform utilization is infeasible by definition with either RVS or DVS. The former suffers from work discovery latencies when parallelism is not much more than the allotment size. The latter, DVS, has an inherent latency in spreading tasks across distance zones.

DVS does manage to improve uniformity of worker utilization. To evaluate this DVS was implemented in an optimized version of the WOOL scheduler (WOOL-DVS), and compared to the original version supporting leapfrogging (WOOL-LF); all code other than victim selection including applications were the same. Figure 5.10 shows the useful utilization of workers for the Sort benchmark, with various allotment sizes; all WOOL-DVS workers spend the same relative amount of cycles executing application code. This result is supported by figure 5.9 which shows that DVS workers performed much less steals (median values) for all benchmarks; the results shown are relative to the RVS result, where the amount of total RVS steal attempts with 5 workers per benchmark is set to 100% and all other measurements are relative to it. Figure 5.11 shows the standard deviation of worker useful time for all benchmarks; DVS worker useful utilization is significantly more uniform than RVS. The uniformity allows to infer the status of all workers by monitoring only
a sample. As discussed earlier this is an important requirement for a scalable requirements estimation mechanism.

The third property of \textit{inward load concentration} is a direct consequence of victim prioritization rules. As discussed in the relevant subsection, nodes in class $Z$ will be primarily
5.5. DVS: DETERMINISTIC VICTIM SELECTION

Figure 5.12: Performance measurements on the two platforms. Each plot shows the execution time in cycles (Y-axis) over several allotment sizes (X-axis).

processing leaves, with the bulk of the task-tree being processed by classes $X$ and $F$. This property has no consequence toward performance with a fixed pool size; contrary it might even increase the amount of steal attempts increasing the scheduling overhead. However, this overhead has been measured to be immaterial to the overall performance, while it eases adaptation significantly.

The final property of DVS is its performance. Figure 5.12 shows total execution time in cycles for the original and DVS implementations of WOOL, for various fixed allotment sizes. For the regular highly parallel applications ($FIB$, $Skew$, $Stress$) both exhibit identical performance. For more irregular applications however DVS improves with larger allotment sizes. One exception being the $Sort$ benchmark on Barrelfish (simulator), which only with 20 workers performed 9% slower. Since this is not mirrored on real hardware, its attributed to the algorithmic unsuitability of said benchmark and DVS scheduling, where its negatives effects are leveraged by the introduction of a cache hierarchy.

A final discussion point on DVS is the significance of task granularity. Nodes depend
on neighboring nodes to spawn new tasks in order to provide them work. If a node is involved on a long-running serial task, no such tasks will be spawned, and the further away nodes will starve. If this happens then it can be argued that the available parallelism does not support the amount of workers. Lets deconstruct this problem slowly. Assume the trapped node is in $Z$, then the starving node will be in either $Z$ or $F$. By construction $Z$ nodes are immaterial to feeding $F$ nodes, whom can start stealing from $X$. Further $Z$ starving nodes only add to the claim of reducing the allotment size. Lets assume the starving node is in $F$. That means that both $Z$ and $X$ classes have been incapable of providing enough tasks, supporting once again the claim to reduce the allotment size. The same logic applies to starving $X$ nodes.

However, what if other nodes have full task-queues located beyond the starving nodes’ reach. In the case of fine grained tasks or tasks producing plenty of child tasks, neighbors will quickly go through them and disseminate to other. If these stagnant tasks are long-running serial tasks, then with a help-first scheduler DVS could fail in disseminating the contents of said task-queue; a problem a work-first scheduler wouldn’t have as these child tasks could potentially be spawned by different nodes, ending up on different task-queues. In the case of a help-first RVS scheduler, any starving worker would potentially be able to steal those tasks, although at high scale they would have to be really lucky in selecting that one task-queue out of every other.

**Limitations**

DVS effectively controls the placement of tasks on an allotment to enable easy and accurate estimation of future parallelism, without imposing changes on the programming model. Nevertheless, such control brings forth certain limitations.

DVS depends on the recursiveness of the fork/join parallelism model to feed further away workers with tasks. The loner worker’s distance from the source, higher the recursion depth before it can steal its first task. Assume a worker $a$ at distance 3. The source starts the main function and spawns some tasks. These are stolen by workers at distance 1 (and 2). $a$ has to wait for a neighboring worker at distance 2 (or 1) to spawn a task before it can make its first successful steal. Thus for a worker at distance 3 to be able to steal at least a leaf task, the task tree must have a depth of at least 3.

This limitation means that DVS would fail with a wide but flat tree. A tree depth of only 1 could utilize workers up till distance 2 from the source — or 13 workers in total — due to the fixed virtual topology. However, it can be argued that a completely flat tree has little to gain from an elaborate work-stealing scheduling algorithm altogether. Since all tasks would be generated by the source and placed in its own task queue, a scheduler employing a centralized task-queue (like the OpenMP for-loop construct) would be more effective in both performance and memory.

A second limitation of DVS appertains only to the Cilk scheduler, specifically with the *continuation passing* mechanism. With Cilk, a worker switches execution to the spawned task, leaving the continuation of the parent as stealable for other workers. This means that a worker’s task spawning rate is potentially slowed considerably. DVS depends on the fact that a certain task will potentially have more children than its parent; thus being able
to feed a larger amount of further away neighbors. With Cilk’s continuation passing, a 
worker spawns a single task, before it switches to a new task, possibly having a preamble 
of computation before spawning more children. Although the performance penalty is con-
siderable, our experiments (presented in the next section) have shown that performance is 
similar with the original RVS-based Cilk.

5.6 Palirria: parallelism feedback through DVS

5.6.1 Description

Palirria (Greek παλίρροια, meaning tide) builds upon Deterministic Victim Selection (DVS) 
to define a method for dynamic adaptation of a work-stealing scheduler’s allotment size. 
It consists of: i) a method for estimating available parallelism, ii) a policy for deciding the 
suitability of the current number of workers relative to said estimation, and iii) a mech-
anism for dynamically adjusting the number of worker threads. The rest of this section 
intuitively describes each of these components, while they are formally defined in the next 
section, followed by implementation algorithms, and then discussion and evaluation in 
section 5.6.4.

The cornerstone of Palirria, like any such mechanism, is its chosen metric for quanti-
fying parallelism potential. Palirria exploits the properties of DVS which enable the use 
of a worker’s task-queue size as such metric. The dependence of Palirria on DVS is not 
a taxing requirement, since victim selection is an independent component of any work-
stealing scheduler. Palirria makes use of the controlled distribution and concentration of 
tasks that DVS creates, to infer the utilizability potential of the allotted workers. Based on 
a predefined threshold and the DVS classification of workers (see figure 5.4), the decision 
policy is as follows:

- **Under-utilized**: decrease allotment, if the size of the task queue of *each* worker 
  belonging to class Z is 0.

- **Over-utilized**: increase allotment, if the size of the task queue of *each* worker be-
  longing to class X is above a threshold $L$.

- **Balanced**: unchanged allotment, if previous conditions are both false or true.

Palirria’s decision policy claims that the state of class X is sufficient for inferring 
potential increase in parallelism. Going back to the DVS ruleset, X is responsible for 
transferring most of the load away from the source. Thus they steal tasks high up on the 
task-tree. If those tasks end up being leaves, spawning no other tasks, no other class could 
have pending tasks that are not leaves. It is important to note the sampling totality of both 
conditions, that all members of Z or X respectively need to evaluate each positively. For 
X this includes those nodes directly adjacent to the source. If the source cannot provide 
large subtrees, there is going to be no more exploitable parallelism.

However, class X members having empty task-queues is not a sufficient condition for 
evaluating lack of parallelism. Contrary if their task-queue’s size is beyond a threshold
CHAPTER 5. ADAPTIVE WORK-STEALING SCHEDULING

$O_i$: Outer victims of $w_i$

Figure 5.13: The set of outer victims $O_i$ for each worker $w_i$, is quantitatively equal to the thieves primarily stealing from $w_i$.

$L$, overall parallelism must have increased. $L$ is defined as an integer value larger than the amount of thieves of a $X$ member node located further away than itself; this set is later defined as $O_i$ for worker $w_i$, visualized in figure 5.13. $L$ is the single configuration variable for Palirria, directly controlling the sensitivity of the algorithm. By definition $L$ is different for each node in $X$; however, by equating amount of thieves with amount of victims, each worker can set its $L$ value independently of others, relative to its own $O_i$. Intuitively it means that if a node in $X$ has spawned enough tasks for all of its thieves, but these are not getting stolen, these thieves are otherwise occupied. Thieves of members of $X$ are primarily members of $F$, but they primarily steal from $Z$; that means that $Z$ is able to feed $F$ enough so that they do not need to steal from $X$. Consequently, these tasks are unnecessarily stacking up in $X$ and more workers could be employed.

The first condition speaks toward reducing the size of the allotment due to lack of parallelism. Again only a subset of all nodes is sufficient to infer such claim. If all nodes of $Z$ have no potential parallelism — their task-queues are empty — for one $F$ will not be fed, and second $X$ has not been able to provide them with large subtrees. Workers in $F$ will turn to $X$ for acquiring more tasks and $Z$ will starve. Hence class $Z$ is redundant and its members can be removed.

DVS has been constructed in such a way that members of $Z$ are the first to starve in the absence of sufficient parallelism. This property enables selecting those workers for removal irrespective of the current utilization level of any worker. Upon reducing the size of the allotment, workers will be reclassified, victims sets reconstructed, and quickly the load will be redistributed. If it doesn’t then parallelism is reduced urging for further shrinkage of the allotment size.
Another point is that the condition does not account for the current status of those workers, meaning what kind of task they are currently processing if any. A non-empty task-queue means that the owning worker is utilized with some task. Other workers might be utilized too. Palirria quantifies future parallelism and thus does not need to know if the current allotment size was sufficient for the parallelism that was available.

If both conditions evaluate to true simultaneously, it means that a sudden lack of parallelism could render $Z$ idle but $X$ is providing parallelism. Thus if parallelism levels are to be plotted such a situation occurs at an inflection point, ending a concave upward portion of the graph. Since parallelism is being increased there is not need to change the allotment size at that time.

Finally, both conditions evaluating to false simultaneously means the opposite situation of having $X$ workers produce no more parallelism than $Z$ members. For $Z$ to have stagnant tasks it means that both $Z$ and $F$ class members are probably utilized enough. This case could signify the end of a parallel section, although there is not evidence that a new parallel section is not following quickly after. On this scenario’s parallelism plot this case would be an inflection point, entering a concave downward part.

A first remark on the decision policy is that the source worker and class $F$ members are not considered. Remembering the virtual diamond shaped topology of DVS, the size of class $F$ expands faster than the $X$ and $Z$ classes as the total allotment size increases. The added nodes to be sampled are the size of the new $Z$ class plus 4 more nodes in $X$ minus the size of the previous $Z$ class, with every allotment size increment. This amount is fixed at 8. Hence as the allotment size increases, the sample size difference becomes less and less significant, until it eventually converges. This results in monotonically reduced overhead and improved scalability.

Another remark is that the decision policy includes a well defined neutral state. This means that the allotment size is not unnecessarily changed without evidence of respective change in parallelism.

The size of the task-queue is a value which most work-stealing schedulers have readily available. Since any dequeue implementation needs to keep pointers to both ends, calculating the size could be just a single subtraction, making the computation requirements of Palirria’s decision policy rather small.

Moreover, the notion of time needs to be defined as perceived in the model Palirria uses. Theoretically Palirria conditions treat the allotment as frozen in time, thus conceptually there is no duration inherent in the model. As if a photo was taken which then Palirria analyzes. This concession is necessary since no historical data are being used. For example it is not a false conclusion if a worker is marked as busy while it is actually just starting a stealing session. Stealing is an unavoidable step of the work-stealing mechanics, and any thief is arguably busy until stealing has completed unsuccessfully. Thus the thieving worker has not become idle yet and any stealable task in its victims’ queues is future work or in other words available parallelism.

This brings forth the notion of stealing as a process and when Palirria treats it as completed. DVS bounds any worker’s number of steal attempts to the constrained size of its victims set. Until this set has been exhausted by one steal attempt on each member, DVS anticipates the possibility of success and lack of parallelism cannot be argued. Palirria
however, is not monitoring the actions of the workers. Instead it check the amount of available tasks. Since there is no notion of duration accounted by Palirria conditions, a non-stolen task corresponds to work that has and will be performed in the future, by either the owner or a thief. To measure the excess work, Palirria simply counts the number of potential thieves and assumes that all are currently in the process of stealing, regardless of their actual status. If there are more than one task available for each potential thief, then the allotment should be increased. Finally this judgment is not done using total numbers, rather locally for certain workers, because of the inability to disseminate tasks freely at any distance from the source.

Finally, Palirria conditions are theoretically applied continuously, as opposed to a discrete sample. Of course that is infeasible and an interval must be set. Nevertheless the interval’s length and the conditions are not conceptually connected or co-dependent in any way. That means that Palirria is not judging the status for the upcoming quantum, rather only a specific instant. This does not affect the validity of Palirria; nevertheless, tuning the interval would affect the overall accuracy the same way sampling frequency affects the quality of sound.

5.6.2 Definition and discussion

The validity of the claim that Palirria infers global status via just a subset of workers must be proved; especially since this subsets growth is reversely proportional to the total increase of workers. This section formally defines parts of Palirria and provides a formal description of its functionality, to help the reader proceed to the proofs on the validity of the decision conditions (Claim 3).

Definitions

Before proceeding with defining and proving specific conditions, it is necessary to include a few more definitions extending those in section 5.5.2.

We define the communication distance between two workers \( w_i \) and \( w_j \) as the hop-count \( \text{hc}(w_i, w_j) \); this is the shortest communication path between the physical cores where the two worker threads are located. We define as diaspora \( d \) the maximum distance between the source worker \( s \) and any other worker in its allotment \( I \). Finally, the estimation conditions will be referred to as the Diaspora Malleability Conditions, or DMC for short.

A zone is a subset of workers located at the same distance (hc) from the source. A zone’s members can be of different classes. Each allotment consists of multiple zones. For any worker \( w \), its inner zone is at 1 hop less, while its outer is at 1 hop more. The outermost zone of an allotment includes all workers at distance \( d \) from the source.

Zones are important because they indirectly define the virtual topology; adding or removing a zone would impose reclassification of workers; completing a zone (by adding missing workers) would only expand workers’ victims sets. Moreover, upon deciding the reduction of an allotment the workers removed are of the outermost zone; since members of the outermost zone are not always of class \( Z \), a zone is a better definition.
Definition 3. We define a **Distance Zone** \( N_c(w_i) \) from \( w_i \), as the set of nodes that have the same distance from a specific node \( w_i \). Thus:

\[
N_c(w_i) = \{ w_j \in W : hc(w_j, w_i) = c \}
\]  

(5.1)

We further define a **Zone** \( N_c \) as the set of nodes that have the same distance from the allotment’s source \( s \). Thus:

\[
N_c = N_c(s) = \{ w_j \in W : hc(w_j, s) = c \}
\]  

(5.2)

**Lemma 1 (Zone size).** The total size of each Zone \( N_c \) is given relative to the topology’s dimensions \( m \) as:

\[
\|N_c\| = 2mc^{m-1}, \text{ where } 0 \leq k \leq d, m \in \mathbb{N}
\]

For DVS’s 2-dimensional topology we can simplify this equation as:

\[
\|N_c\| = 4c, \text{ where } 0 \leq k \leq d \tag{5.3}
\]

As a direct result there is formula for calculating the complete size of the thieves set, meaning all nodes but the source.

**Corollary 1 (Thieves set size).** Given lemma 1, the total number of thieves in allotment \( I_s \) is:

\[
\|T_s\| = \sum_{k=1}^{d} \|N_k\| = \frac{(2d + 1)^m - 1}{m}
\]

For a 2-dimensional topology this equation is:

\[
\|T_s\| = \frac{(2d + 1)^2 - 1}{2} = 2d(d + 1) \tag{5.4}
\]

**Corollary 2 (Allotment size).** The maximum number of nodes in allotment \( I_s \), with an \( m \)-dimensional topology, is: \(^1\)

\[
\|I_s\| = \|T_s \cup \{s\}\|
\]

(5.5)

All workers in an allotment are divided into **classes** according to their geometrical position in the allotment and in relation to the source. The reader is encouraged to revise the definitions of those classes in section 5.5.2.

Furthermore, certain members of a worker’s \( w_i \) victim set \( V_i \) are also grouped as \( O_i \subset V_{i_a} \subset V_1 \). \(^2\)

**Definition 4.** We define the **Outer Victims set** \( O_i \) as the set of the victims of worker \( w_i \) at distance 1 located in its outer zone.

\[
O_i = \{ w_j \in V_{i_a} : hc(w_j, s) = hc(w_i, s) + 1 \}
\]

\(^1\)This formula is a direct result of the topology’s geometrical properties and the proof is considered trivial.

\(^2\)For a formal definition of \( V_i \) and \( V_{i_a} \) please see section 5.6.2.
Since members of \( O_i \) are at distance 1 from \( w_i \), \( w_i \) is their victim apart from them being its victims. Thus \( O_i \) includes those workers whom \( w_i \) is stealing from but also providing work to.

For the remainder of this section we use \( \mu(\bullet) \) as the typical measure of a set, evaluating to the set’s amount of members, with \( \mu(\emptyset) = 0 \).

**Definition 5.** Task-queue size \( Q_i \) is defined as the set of tasks placed in a worker’s queue. These are tasks spawned but not yet stolen or processed.

**Theorem 1.** Each worker \( w_i \in I \) is a victim to as many workers as there are in its own victims set \( V_i \).

Thus we write \( \mu(V_i) \) and mean both the number of victims for worker \( w_i \) but also the number of other workers that steal from \( w_i \).

**Proof.** Assume any worker \( w_i \). Its victims set is \( V_i = V_{ia} \cup V_{ib} \). Each set will be approached individually:

- Set \( V_{ia} \) includes all workers at distance 1. This rule applies to all workers. It is self-evident that all workers in \( V_{ia} \) will have \( w_i \) in their respective \( V_{ia} \) set due to the same rule.

- Set \( V_{ib} \) applies only to workers in \( Z \) and a special subset of those in \( X \). Again this set is defined based on distance and defines victims of the same class as the owner of \( V_{ib} \); thus the same rule applies to all members of \( V_{ib} \), thus all workers in this set will also have \( w_i \) in their respective \( V_{ib} \) set. \( \square \)

**Definition 6.** Stagnant task Assume allotment \( I \), with source \( s \) and current diaspora value \( d \) and any worker \( w_i \in I \) with queue of tasks \( Q_i \) and victims set \( V_i \). All tasks in \( Q_i \) in excess of \( \mu(V_i) \) are called stagnant. Stagnant tasks correspond to work beyond what is required to satisfy all the thieves that depend on \( w_i \).

**Definition 7.** Task generation rate \( \Delta G_i \). Assume allotment \( I \), with source \( s \) and current diaspora value \( d \) and any worker \( w_i \in I \). Let \( G_i \) symbolize the tasks generated by worker \( w_i \). Then \( \Delta G_i \) is the amount of tasks spawned between the current timepoint and when \( w_i \) started processing its current task.

Definition 7 above symbolizes the amount of new tasks a worker created while still being busy with a task. Thus it quantifies the amount of parallelism it made available while still being busy processing a task.

**Definition 8.** Worker’s parallelism slack \( \text{Slack}_i \). Assume allotment \( I \), with source \( s \), current diaspora value \( d \) and any worker \( w_i \in I \). \( \text{Slack}_i \) is the amount of stagnant tasks in the task-queue of \( w_i \) at a specific time-point. It is defined as the worker’s task queue size minus the number of its outer victims. Thus:

\[
\text{Slack}_i = \mu(Q_i) - \mu(O_i) \leq \Delta G_i
\]
5.6. PALIRRIA: PARALLELISM FEEDBACK THROUGH DVS

Since \( w_i \)'s victims are also thieves primarily stealing from \( w_i \), \( \text{Slack}_i \) quantifies the minimum amount of parallelism slack available by \( w_i \) at that instant. It is minimum because of the assumption that all those thieves would be simultaneously idle and stealing from \( w_i \).

**Definition 9. Worker's work potential** \( N(i) \). Assume allotment \( I \), with source \( s \), current diaspora value \( d \) and any worker \( w_i \in I \). \( N(i) \) corresponds to the potential of a worker to have work and is defined as its own parallelism slack plus the aggregated slack from members of its victims set.

\[
N(i) = \text{Slack}_i + \sum_{j \in V_i} \text{Slack}_j
\]

Notice that a positive \( \text{Slack} \) denotes an excess in parallelism, while a negative a lack of it. Furthermore, a positive work potential for the workers of class \( Z \) speaks to the workload's utilization stability. In other words, it guarantees that workers in \( Z \) will have work. Workers in \( Z \) steal first from within class \( Z \). Workers in \( F \) steal first from within \( Z \) also. Thus the existence of stagnant tasks in \( Z \) means that members of \( Z \cup F \) have work. Moreover, if members of \( X \) did not have work they would steal from \( Z \) or \( F \). Thus the existence of stagnant tasks in \( Z \) means their entire innermost zone has enough work. This reasoning can be iterated down to the source, thus allowing to generalize the claim for the whole allotment.

**Claim 1 (Utilization stability).** If \( N(i) > 0 \forall w_i \in Z \) then the allotment is utilized.

**Malleability of Diaspora**

This section presents and explains the conditions which can be used for estimating the utilization level of a workload. An increase of workers should be performed when the amount of already produced work is enough to utilize the added resources; when the outermost workers (in \( Z \)) are found underutilized they can be removed without risking performance loss.

**Claim 2 (WSC). Workload Status Conditions:**

- **There is over-utilization** when the size of the task queue \( Q_i \) of each worker \( w_i \) in \( X \) increases beyond \( L \).

\[
\forall w_i \in X : \mu(Q_i) > L > \mu(O_i) \Rightarrow \forall w_j \in I : \text{Slack}_j > 0 \tag{5.6}
\]

- **There is under-utilization** when the task queue of each worker in \( Z \) is empty.

\[
\forall w_i \in Z : \mu(Q_i) = 0 \Rightarrow \forall w_i \in Z : \text{Slack}_i < 0 \tag{5.7}
\]

\( \mu(O_i) \) is the number of workers in the outer zone that can steal tasks from \( w_i \) and is different for every worker. \( L \) is theoretically bound at \( \mu(O_i) \); using different values for \( L \) (like \( \mu(O_i) + 1 \), but not constant) can tune the tolerance of the model. Conceptually
$L$ guarantees that momentarily there is immediately enough work for the new workers to steal. If these tasks are leaves, the allotment will most probably shrink in the next quantum. If not, the slack will be distributed very fast outwards, generating new stealable tasks further from the source.

**Claim 3 (DMC). Diaspora Malleability Conditions:** The following conditions evaluate WSC to decide if the allotment size should be changed

- **Increase** the diaspora value, and in effect the allotment size, when:

  $$d^+ \iff (\forall w_i \in X : \mu(Q_i) > L > \mu(O_i)) \land (\exists w_j \in Z : \mu(Q_j) > 0) \quad (5.8)$$

- **Decrease** the diaspora value, and in effect the allotment size, when:

  $$d^- \iff (\forall w_i \in Z : \mu(Q_i) = 0) \land (\exists w_j \in X : \mu(Q_j) \leq L > \mu(O_j)) \quad (5.9)$$

- **Balanced** allotment size exists when:

  $$d^w \iff \neg ((\forall w_i \in Z : \mu(Q_i) = 0) \oplus (\forall w_j \in X : \mu(Q_j) > L > \mu(O_j))) \quad (5.10)$$

**DMC validity on a ideal model**

Before proceeding with formally proving the claims above, it is better to discuss certain implications regarding how DVS distributes tasks. For clarity, first a ideal model is used where the task tree is complete, steal attempts are instantaneous, non-leaf task granularity homogeneous, and parallelism potential is at least 3. Although this model is close to ideal, it does cover a multitude of real application patterns. Nevertheless, the model is extended in the next section with several properties for possible irregularity.

**Figure 5.14:** Workers of class $Z$ can be characterized as *peak* and *side*, based on their location relative to the topology’s diamond shape.

DMC check the status of $X$ and $Z$, disregarding class $F$. The latter primarily steals from $Z$, so one might argue that they have the greatest probability to steal leaves. However,
workers in class $F$ will have reverted to other victims — potentially in $X$ — much before $Z$ workers have spawned stealable tasks. For $Z$ to spawn stealable tasks it is required that $d$ nesting levels of parallelism have been exposed. To elaborate, the root spawns tasks that feed class $X$; it takes $d - 1$ members before a $X$ worker neighboring $Z$ will spawn tasks, resulting at $Z$ stealing tasks that are at $d$ depth in the task tree. Lets assume this process takes $d - 1$ time units to complete, since task granularity is homogeneous. $Z$ is executing tasks after $d$ time units. At that time the $Z$ workers which have tasks are the three comprising a peak of the diamond shape (fig 5.14); focusing on the two side workers between them, they each have one $F$ worker neighbor which is also the furthest away from the source as $F$ can be. These $F$ workers then will be able to steal a task at a depth of $d + 1$, or in other words at least $d + 1$ time units have past before they can execute a task stolen from $Z$. It takes much more for $F$ workers close to the source to steal tasks through that path.

**Corollary 3.** It will be at least $d + 1$ time units before an $F$ class worker executes a task stolen from a $Z$ class worker.

Workers in $F$ also steal from $X$ and workers within $F$. Considering that $Z$ wont be able to feed tasks to $F$ until after $d$ time units, workers at the side boundaries of $F$ will have stolen from $X$ before that, with a bit smaller probability than $X$ workers would, since they would attempt $X$ victims only after having failed at $Z$ and $F$ at further away zones, as defined by 2. The $F$ worker that is the furthest away from the source is at distance $d - 1$. That means that it takes at least $d - 1$ units for a worker in $F$ to execute a task stolen from $X$ without passing through $Z$. Thus considering the task-tree, even the furthermost worker in $F$ will have a larger sub-tree to work with.

**Corollary 4.** After at least $d - 1$ time units all $F$ workers are executing a task.

**Corollary 5.** Workers in $Z$ have the highest possibility to get the smallest subtrees than any other class.
CHAPTER 5. ADAPTIVE WORK-STEALING SCHEDULING

The worker to receive a task last could therefore be the Z worker at the sides of the diamond. It was shown that the peak-side Z workers need to wait for a task at depth \( d \) of the tree. Thus their Z non-peak neighbors would steal a task at depth \( d + 1 \) from them, exactly as would F; if a Z quadrant comprises of more than 1 non-peak worker, the time units required for them to steal a task through Z grows linearly. These workers however won’t remain idle. Once the furthermost F workers spawn tasks, they will be stealing from them, since Z does steal from F just not primarily. It was shown that the furthest most F worker will be executing its first task at depth \( d − 1 \). Thus all non-peak Z workers have the opportunity to steal a task at depth \( d \), which is the same time units as the peak Z workers.

**Corollary 6.** All workers in Z have to wait the same \( d \) time units before their first successful steal.

At this point there are different outcomes depending on the total depth of the tree:

- If the depth of the tree is \( d \), then all Z workers are executing leaves. All F workers will execute non-leaf tasks.
- If the depth of the tree is larger than \( d \), then all workers get a non-leaf task.
- If the depth of the tree is \( d − 1 \), then Z workers do not get a task and they are the only idling workers.
- If the depth of the tree is \( d − 2 \), then Z workers and the furthermost F and X workers (part of zone \( N_{d−1} \)) do not get a task. Zone \( N_{d−1} \) would comprise the Z zone, if the actual Z workers are removed.

**Corollary 7.** If the depth of the tree is \( d − n \), \( n \in [1, d) \), then all workers in zones \( N_j \), \( j \in [d−n, d] \), cannot steal any task.

Corollary 7 exemplifies the meaning of zones and why Palirria changes the allotment size in steps of full zones. It shows that Z is the zone to be first under-utilized due to lack of parallelism. The analysis also showed that X is the class of workers to acquire the largest subtrees. Thus it is intuitive to conclude that if Z workers have empty task-queues, the execution of the task tree has already reached its final depth and parallelism will start to shrink. If however all X workers still have enough tasks for all the thieves they feed, there is probably some fluctuation which could change shortly; the decision is to do nothing. If X workers do not have enough tasks in their queues, the previous conclusion is probably valid and the allotment size should shrink. If Z workers have tasks in their task queues, the task tree depth is larger than \( d \) and since F workers are already occupied with non leaf tasks, the allotment size should increase.

**DMC validity on a realistic model**

Task-trees are seldom complete, with no fixed structure on task parallelism potential (amount of children) nor granularity; realistically it should be assumed that tasks have differing
granularity and arbitrarily many children. Finally steal attempts are not instantaneous and not even uniform in duration due to NUMA. A realistic model for validating Palirria can not make any assumptions on these characteristics.

Nevertheless, one part of the ideal model analysis still holds, which is that each class and zone can steal tasks only after a certain task tree depth. A difference of the non-ideal model however, is the absence of uniform sequence of events and time unit homogeneity. Varying granularity can cause delays and force alternate stealing paths for workers. A unbalanced tree could also cause local starvation. This section emphasizes on these issues and analyzes the validity of Palirria in respect to each and all of them.

The purpose of this analysis is that even though certain irregularities could potentially create false positives for Palirria, eventual changes to allotment sizes are not detrimental to the completion of the program nor negatively affect performance. The individual cases to be investigated are long running non-leaf tasks, long-running leaf tasks, and unbalanced task trees where leaves can occur at any depth.

In the first case, a worker \( P \) might be consumed in a sequential computation part of a long running non-leaf task. The spawning of child tasks could happen arbitrarily during its execution; spread across uniform or non-uniform intervals, or clustered close to the beginning or end of the execution.

If the children are spawned early there is no issue for the allotment since the parallelism potential is immediately available to other workers. This scenario is unaffected by allotment changes; shrinking by Palirria does not preempt workers, they are forbidden to steal but can finish their current task and their own task queue.

If tasks are spawned after the necessary time for all thieves dependent on \( P \) to become idle, they would be required to find alternative victims. This diversion might require these worker to get tasks much deeper in the task tree than they would normally get, according to the distance dependent scheme of DVS. Thus at this point a distinction has to be made on the task-tree depth. Let \( P \) be at distance \( k < d \) from the source. If the task-tree depth is \( d \) or less, then there is high probability the thieves dependent on \( P \) would starve; We’ll come back to this scenario later on. If it is \( d + x \), then every such worker at distance \( n \in [2,d] \) from \( P \), on \( P \)'s axis from the source, would get a subtree of depth \( sd = (d+x)-(k+n+2) \) where \( sd \leq d + x \) and \( k + n < d \). This formula would predict starvation when it evaluates to zero or less. Assuming \( d \) as a known constant and solving the system gives:

\[
\begin{align*}
  k + n + 2 &= d + x \\
  k + n &< d \\
\end{align*}
\Rightarrow
\begin{align*}
  d + x - n - 2 &= k \\
  d + x - n - 2 + n &< d \\
\Rightarrow
x &< 2
\end{align*}
\]

Thus there would be localized starvation only if the task-tree depth is \( d + 1 \). This starvation would affect only the portion of \( Z \) class that is on the axis of the source and \( P \). The rest of \( Z \) would find only leaves due to corollary 6 which applies. Thus the under-utilization WSC will evaluate true. Since the depth is \( d + 1 \) a \( X \) worker at distance 1, will spawn a subtree of depth \( d - 1 \). The furthest most \( X \) worker \( w_i \) will spawn a subtree of depth 1, which are the leaves to feed the \( Z \) class workers that neighbor it. If the amount of children is enough to evaluate positively the over-utilization WSC, the DMC evaluating true will be the balanced state and no new workers will be added. If the amount of children
is a multiple of $\mu(O_1)$, the allotment will be balanced and utilized. If not, then the over-utilization WSC will immediately evaluate to false, combined with the under-utilization WSC which as discussed is always true, the decrement DMC will fire, and $Z$ will be removed. Every DMC evaluation round thereafter will result in reducing the allotment size. Conceptually this is exactly what should happen, since a $d+1$ task tree is enough to utilize the outermost zone only with one recursion level.

The discussion above has assumed a uniform tree — not necessarily complete — where every branch has the same depth. If the tree is non uniform the logic is the same as before, except that there is no parallelism potential to be had at any point of the task’s execution. Let a certain worker $P$ anywhere but the $Z$ class could happen to execute a leaf. Workers on the axis of $P$ and the source will behave as in the previous case. Nevertheless, all other workers can find stealable tasks from other victims, with the same parallelism potential.

Conclusively the aforementioned situation can produce starvation only if the task-tree is at most $d+1$. Considering realistic numbers, a typical parallel application would have a depth of at least 5 for seeing benefits through work-stealing, as opposed to flat parallelism and a centralized queue. With DVS, a value of $d$ translates into 61 nodes. The real life applications that have been evaluated to scale the achieved speedup on that many cores are embarrassingly parallel applications that generate millions of tasks and quite deep task trees. Even with traditional random based work-stealing, many applications do not scale well [52, 65]. Thus having a requirement in the programming model that connects scale with task-tree depth is not unreasonable. It could also be viewed as a useful tool.

**Corollary 8.** A task-parallel application using a DVS-supporting runtime, should have a maximum diaspora value of “task-tree depth - 2”.

However, one important aspect has been overlooked; that of the tree width. So far it has been presumed that tasks spawn enough children to feed thieves at all zones. It is however not a necessity and depends on the width of the task-tree at the respective depth. This width can be uniform or not at all. A non uniform width could very easily cause localized starvation. If the overall tree depth is also borderline for utilizing the allotment, this starvation could be irreparable. Thus corollary 8 could be strengthened further:

**Corollary 9.** A DVS zone $N_c$ cannot be fully utilized if the task tree at the respective depth $c$ has width less than the zone’s size $4c$ (from lemma 1).

Conclusively, effective use of DVS and in effect Palirria depends on constraints over the amount and recursive depth of the application’s parallelism. Nevertheless, problematic are those cases involving quite small amounts of both, as discussed above; it should also be emphasized that small parallelism recursion depth and task parallelism potential affects randomness-based work-stealing methods too; on one hand DVS makes the distribution of that potential deterministic, on the other the issues with lacking depth are irreparable.

Randomness-based victim selection (RVS) can theoretically leverage increased parallelism potential with lacking depth by unconstrained stealing reach. However, experimental evaluation showed DVS outperforming those techniques due to the much faster and
deterministic distribution. With DVS less workers can be utilized when the tree is shallow; however, all utilizable workers will receive the same or larger subtree than with RVS, reducing the amount stealing.

### Time Related Considerations

Palirria and its conditions are meant to be applied periodically. Ideally they should be applied after every successful steal and spawn operation. This is utterly impractical so the period length has to be relaxed to something much larger and manageable. The model as described above makes two assumptions: i) that every task’s execution time is strictly longer than any single steal operation, and ii) that every task’s execution time is at least equal to the duration of as many steal attempts as there are members in the largest victims set in the allotment. For DVS the largest possible victims set size is 6; a successful steal attempt takes more cycles than a failed one. On average, 6 steal attempts where the last one is successful would take from a few hundred to a few thousand cycles at most. Thus the aforementioned constraints are reasonable in defining a lower bound for Palirria’s periodicity. An upper bound is arbitrary and can be set as tightly to he lower bound as the platform supports and the overall overhead constraints permit.

The reasoning behind the assumptions above is that for DMC to hold there must be enough time between invocation to allow a change in the status. Because the conditions include no notion of time, invocation in time less than the duration of a single task or a steal session could potentially produce a series of conflicting results, each negating the previous one.

### Proof of Worker Increase Condition’s Validity

To prove the validity of condition (5.8) it is required to prove its necessity and sufficiency. The first part is straightforward: show that when the condition is true, there is indeed an increase in workload enough to utilize one more zone of workers. For the following proofs tree depth is considered uniform but individual task parallelism potential and thus tree width are not.

The existence of a positive parallelism slack available to each worker, can be expressed as follows:

\[ \mu(Q_i) > L > \mu(O_i), \forall w_i \in X \land \exists w_j \in Z : \mu(Q_j) > 0 \Rightarrow \forall w_k \in I : \text{Slack}_k > 0 \]

(part 1: sufficiency).

Assume \( \mu(Q_i) > \mu(O_i) \Rightarrow \text{Slack}_i > 0 \ \forall w_i \in X \)

\[ \Rightarrow \Delta G_i > \mu(O_i) \ \forall w_i \in X \]

Now assume that: \( \forall w_j \in I \setminus X \Rightarrow \text{Slack}_j \leq 0 \)

\[ \Rightarrow \Delta G_j \leq \mu(O_j) \ \forall w_j \in I \setminus X \]
Moreover, since all $X$ workers have excess tasks, it can be assumed (depth uniformity) that $F$ workers at the same distance have an equivalent amount. This covers all $F$ workers. Thus all non $Z$ workers are busy or have the potential to get work from neighbors.

The initial condition needs at least one worker in $Z$ to have a non-empty task-queue. That means that at least one $Z$ worker has received a subtree of depth larger than 1. Due to the tree uniformity property then all $Z$ workers have potentially the same subtree available.

In effect all stagnant tasks in the $Z$ class are not required to feed non $Z$ workers. Moreover $Z$ workers are already busy. Consequently more workers directly neighboring the current $Z$ class/zone can be utilized at that exact instant.

However, the condition requires $\mu(Q_i) > \mu(O_i)$. Thus at least one more stagnant task to justify an allotment increase. Practically it is left up to the implementer of this model to decide on a value of the $L$ threshold, with a minimum of $\mu(O_i) + 1$.

The second and last part of the proof deals with the necessity of the condition. In other words:

$$Slack_j > 0, \forall w_j \in I \Rightarrow \forall w_i \in X : \mu(Q_i) > L > \mu(O_i)$$

(part 2: necessity).

$$\forall w_j \in I : Slack_j > 0 \Rightarrow \forall w_j \in I : \Delta G_j > \mu(O_i)$$
$$\Rightarrow \forall w_j \in I : \mu(Q_j) = \Delta G_j > \mu(O_j)$$
$$\Rightarrow \forall w_i \in X : \mu(Q_i) > \mu(O_i)$$

Proof of Worker Decrease Condition’s Validity

To prove the condition for decreasing an allotment, it must be shown that it verifies absence of excess workload.

(part 1: sufficiency).

Assume $\forall w_i \in Z : \mu(Q_i) = 0 \Rightarrow \Delta G_i \leq \mu(O_i)$
$$\Rightarrow \forall w_i \in Z : \forall w_j \in V_i : \Delta G_j \leq 0$$
$$\Rightarrow \forall w_i \in Z : Slack_i < 0$$

Finally, it must be shown than an absence of workload is concentrated in the members of $Z$.

(part 2: necessity).

$$\forall w_j \in Z \Rightarrow \forall w_j \in V_i : \Delta G_j = 0$$
$$\Rightarrow \forall w_i \in Z : \mu(Q_i) = \Delta G_j - \mu(V_i)$$
However, lack of excess tasks means that \( \text{Slack}_i < 0 \) thus:

\[
\forall w_i \in Z : \text{Slack}_i < 0 \Rightarrow \Delta G_j < \mu(V_i)
\]

\[
\Rightarrow \forall w_i \in Z : \mu(Q_i) = 0
\]

\[\square\]

5.6.3 Implementation

This section presents pseudocode for implementing Palirria. It assumes the existence of DVS, where each worker thread has a victims set as per the DVS classification. The Palirria implementation consists of an algorithm for evaluating the DMC, and discusses implications for altering the allotment size. The latter actions are highly dependent on the work-stealing scheduler, thus the points touched are at a high level.

**Computing DMC**

**Algorithm 9 Palirria Decision Policy**: Check DMC and change allotment respectively.

1: \( \text{ParallelismIncrease} \leftarrow \text{True} \)
2: \( \text{ParallelismDecrease} \leftarrow \text{True} \)
3: for all \( x_i \in X \) do
4: \hspace{1em} if \( |\text{TaskQueue}_i| < |O_i| \) then
5: \hspace{2em} \( \text{ParallelismIncrease} \leftarrow \text{False} \)
6: \hspace{1em} end if
7: end for
8: for all \( z_i \in Z \) do
9: \hspace{1em} if \( |\text{TaskQueue}_i| > 0 \) then
10: \hspace{2em} \( \text{ParallelismIncrease} \leftarrow \text{False} \)
11: \hspace{1em} end if
12: end for
13: if \( \text{ParallelismIncrease} \oplus \text{ParallelismDecrease} \) then
14: \hspace{1em} \( c \leftarrow \text{GetLastZoneSize}(d) \)
15: \hspace{1em} \( m \leftarrow \text{GetLastZoneMissingPercentage}(d, c) \)
16: if \( \text{ParallelismIncrease} \) then
17: \hspace{2em} \( \text{IncreaseAllotment}(c + (m > 0.5? \|N_{d+1}\| : 0)) \)
18: \hspace{2em} else
19: \hspace{3em} \( \text{DecreaseAllotment}(c + (m > 0.5?0 : \|N_{d-1}\|)) \)
20: \hspace{2em} end if
21: end if

Algorithmically checking the utilization status of an allotment means evaluating the DMC against it. Algorithm 9 sketches out the process, which can be split into three parts. The
first part checks the task-queue size of members of class $X$. This step comprises of maximum $4d$ iterations, if all zones are complete. The second step evaluates the task-queue-size of members of class $Z$. This step comprises of maximum $4d$ iterations if zone $N_d$ is complete, according to equation (5.3). Hence the total number of iterations is $4d$. Consequently the algorithm has $O(\sqrt{N})$ time and $\Theta(1)$ space complexity, since the following is true for the total size of the allotment:

$$2d(d+1) + 1 > 4d, \forall d \in \mathbb{N}$$

Palirria changes the allotment’s size based on Zones. Upon incrementing the allotment, if less than half of the outermost zone (class $Z$) is currently allotted, the amount of added nodes will be the additional nodes to complete said zone; if more than half is already allotted, the added nodes will be extended to the next zone also. Similarly, if less than half are allotted, both the incomplete zone and the immediately inner zone are removed.

As an allotment scales up, the amount of nodes sampled by DMC is increased by a constant maximum step of 8 iterations. However, the maximum increase in allotment size is $4d + 1$ (as per equation (5.3)). The factor of the sample’s size over the total allotment’s size is $\frac{8d}{2d(d+1) + 1}$ is monotonically reducing. Hence as the allotment scales up, the sample’s size is increased less and less. At infinite scale, the sample’s size would be relatively 0.

**Increasing an allotment**

The action of adding more worker threads is straight forward and in all respects the same as the initial bootstrapping of the allotment. The necessary steps are listed below:

1. Allocate per thread descriptor structures and populate with necessary values, except the victims set.

2. Reclassify the whole allotment, placing new workers as nodes in the virtual topology and reconstructing their victims sets as per the DVS rules and algorithms presented in section .


The first action is part of the work-stealing scheduler’s stock bootstrapping process and is beyond the scope of this report. Since new worker threads are added, the victims sets of all DVS nodes need to be updated by executing the algorithms from section 5.5.3. This process can be optimized according to the following cases:

- if the added nodes complete the existing outermost zone $Z_d$, only a few nodes are affected; in the worst case this includes the up to $4d$ nodes of the outermost zone and the $4(d - 1)$ nodes of previous to last zone. In total that is $8d - 4$ reclassifications.

It can be shown that this is a monotonically reducing fraction of the total number of nodes for diaspora values greater than 1.
5.6. PALIRRIA: PARALLELISM FEEDBACK THROUGH DVS

**Proof.** Using Lemma 2 we need to show that if \((8d - 4) = \lambda(2d(d + 1) + 1)\) and \(\lambda = f(d)\), then \(1 > f(d_i) > f(d_j) > 0, \forall d_j > d_i > 1, d_j, d_i \in \mathbb{N}\).

- Let \(d_2 = 2\) then \(\lambda = \frac{12}{13} < 1\), and the initial hypothesis is true.
- Assume that for \(d_k > d_2\) it is true that \(f(d_2) > f(d_k) \Rightarrow \frac{12}{13} > \frac{8d_k - 4}{2d_k(d_k + 1) + 1}\).
- For \(d_{k+1} = d_k + 1 > d_k\) we have that
  \[
  f(d_{k+1}) = \frac{8d_{k+1} - 4}{2d_{k+1}(d_{k+1} + 1) + 1} = \frac{8d_k + 1 - 4}{(2d_k + 1)(d_k + 1) + 1} \\
  \Rightarrow f(d_{k+1}) < \frac{8d_k - 4}{(2d_k)(d_k + 1) + 1} \\
  \Rightarrow f(d_{k+1}) < f(d_k)
  \]

- If the addition introduces a new zone \(Z_{d'}\), the added nodes can be up to \(6d' - 3\) and the preexisting affected nodes are \(6d - 10\) as before, for a total of \(12d - 13\). This is again a monotonically reducing fraction of the total nodes. The proof is similar to the previous case, thus omitted.

Algorithm 3 can be adapted to iterate only selected consecutive nodes as shown below in algorithm 10.

**Decreasing an allotment**

Decreasing an allotment is not a straight forward process. The primary issues are: i) the selection of the workers to be removed, and ii) how to handle their task-queues if non empty. DVS preemptively handles both issues. Nodes on the outermost zone are kept utilized *just enough*, thus being the first to starve in the absence of sufficient parallelism. Moreover, the allotment is decreased if and only if the task-queues of all nodes in the outermost zone are empty; thus there is a good chance they won’t have the opportunity to populate them with tasks between evaluating the DMC and enforcing removal.

The Palirria prototype employs what is called as lazy worker removal. Workers labeled for removal (removed-workers) are forbidden from stealing, but allowed to empty their task-queues, and be stolen from. The first time a removed-worker becomes idle, it immediately shuts down by blocking on a conditional. This state of idleness does not include a situation of having to sync a stolen task still executing. Removed-workers will then pick a task from their own queue, busy wait, or poll the state of the task to be synced till its

---

\(^3\)There are maximum \(2d' - 1\) from the incomplete \(Z_{d'} - 1\) zone and up to \(4(d')\) nodes in the new zone \(Z_{d'}\).
Algorithm 10 Incremental DVS Classification: Adapted function to construct the victims sets of nodes from some zones. Code segments identical to algorithm 3 are omitted.

1: function DVSI\text{}NIT\text{}A\text{}LLOTMENT(Allotment, StartZone, EndZone)
2: \hspace{1em} idx ← 2\text{}StartZone(\text{}StartZone − 1) + 1
3: \hspace{1em} idxFinal ← 2\text{}EndZone(3\text{}EndZone − 1)
4: \hspace{1em} DVSI\text{}N\text{}itS(\text{}Allotment[0])
5: \hspace{1em} for StartZone < \text{}i < EndZone do
6: \hspace{2em} \ldots
7: \hspace{2em} if \text{}idx > idxFinal \text{}or \text{}idx \notin \text{}Allotment \text{}then \text{}return
8: \hspace{2em} \ldots
9: \hspace{1em} end for
10: \hspace{1em} lx ← 1
11: \hspace{1em} lf ← 0
12: \hspace{1em} for 0 < \text{}j < d do
13: \hspace{2em} \ldots
14: \hspace{2em} if \text{}idx > idxFinal \text{}or \text{}idx \notin \text{}Allotment \text{}then \text{}return
15: \hspace{2em} \ldots
16: \hspace{1em} end for
17: \hspace{1em} end function

ready. There are many ways to implement this adaptive part and Palirria does not require any specific method.

Lazy removal’s duration might surpass the interval of evaluating DMC. Also, the allotment is not being reclassified until all removed-workers have blocked; hence subsequent DMC evaluations could be performed on blocked nodes. In case the under-utilized result persists, the selection of removed workers will be the same having no negative effect.

If the result of evaluating DMC changes to balanced it is discarded; the reasoning is that events could overlap, having a balanced state because workers blocked, which was the desired effect from decreasing the allotment size. Palirria does not spend cycles monitoring the removal process, thus not knowing if all removed-workers have blocked the new DMC balanced evaluation is discarded.

If the result is over-utilized then the state of the allotment has changed, or too many workers were removed. At that point lazy-removal is reversed. Removed-workers are unlabeled and blocked ones restarted. No new workers are added to the allotment though.

A method for removing workers is not strictly part of the requirements estimation mechanism. However it is a necessary part of any adaptive work-stealing scheduler. The task-queue is commonly on the stack thus the owning thread can’t do much with it without considerable cost. Cilk++ and Cilk Plus on the other hand implement detachable task-queues, which can be passed along to other workers. This feature would enable immediate worker removal. This project has not explored this implementation direction, although the Palirria method can fully support it.
5.6.4 Discussion and Evaluation

Evaluation of Palirria was conducted using the same benchmarking applications, on the same Barrelfish and Linux platforms, running on the same Simics simulator and real hardware as DVS; all details including workload input data sets are discussed in section 5.5.4.

Our evaluation strategy is based on a comparison between three implementations. First is the original non-adaptive version of the runtime scheduler (referred to as WOOL), using different fixed sets of workers. The same scheduler is augmented with each of the resource estimation algorithms, ASTEAL and Palirria as described in their respective sections. WOOL is used as a baseline for all comparisons, revealing the unavoidable overhead introduced by any resource estimation algorithm and the adaptation mechanism, and to provide a baseline for the expected performance.

Additionally we have implemented ASTEAL over DVS victim selection, referred to as DSTEAL in the evaluation. Via DSTEAL we aim to isolate further the accuracy of the adaptive algorithm between ASTEAL and Palirria. DSTEAL’s base scheduler is the same as Palirria (WOOL with DVS victim selection), although the adaptive algorithm is that of ASTEAL.

The operating system’s scheduler removes and adds workers in sets. These sets are the zones as defined in section 5.6.2; they are of fixed size so the total number of workers sequentially scales between them. With ASTEAL, DSTEAL and Palirria each application starts with the minimum set of 5 workers (zone 1 plus the source) and is left to adapt automatically. Using WOOL, we executed each application with multiple fixed allotment sizes. Namely, 5, 12, 20 and 27 workers for the 32-core simulated platform and 5, 13, 24, 35, 42 and 45 workers on the 48 core real machine.

The OS scheduler always satisfies allotment increment requests, up to the total number of cores available. The absence of external constraints in the expansion of the allotment evaluates the ability of the adaptive algorithm to recognize the existing parallelism while also conserving resources. We perceive these properties as fundamental and necessarily evaluated before investigating true multiprogramming scenarios.

The adaptive implementations (ASTEAL, DSTEAL, Palirria) were started with 5 workers and left to expand automatically. Highly parallel workloads like FIB, loose performance this way. However, this strategy can be beneficial if resource conservation is the primary goal. To maximize performance such workloads can start with a big allotment, and shrink automatically.

Note that due to the topology’s geometry, most of the larger allotments are not complete in respect to the zones and classes as defined by DVS. Figure 5.6 visually presents the classification of a 4-zone allotment on each platform.

Since work stealing schedulers don’t include a master thread, we have implemented a helper thread mechanism to execute the estimation and decision algorithms asynchronously to the workers. Each application has a single such thread being invoked at a regular interval. These helper threads are pinned to Core 1, where no worker is running. With this design, changes in the allotment happen in the background with minimal interruption of the actual execution. Note that the data used by these algorithms are generated by the workers themselves, constituting overhead to the execution.
CHAPTER 5. ADAPTIVE WORK-STEALING SCHEDULING

Performance measurements, on Barreelfish (simulator)

Figure 5.16: Results for fixed 5, 12, 20, 27 workers, and self-adapting schedulers. (a) Total execution time normalized to first column. (b) Average percentage of wasted cycles. (c) Adapted number of workers over time.
Figure 5.17: Useful (blue bar bottom) and non-useful (red bar top) normalized time per worker ordered by zone. Useful are the cycles spent successfully stealing and processing tasks. Plots per application (row) are normalized to the first bar (source worker) of the corresponding WOOL-27 run.

Also core 0 is reserved for the system scheduler which controls thread creation and destruction. Because thread creation is expensive and dependent on the threading library used, we have chosen to remove thread control from the applications to make comparison
with fixed-allotment runs fair. The system scheduler provides a pool of threads in advance, one per core, and is also responsible for pausing and resuming them when an application is adapting. Thus in all time measurements the initialization time and thread creation have been excluded.

Finally, the results reported were of the second best run among 10. A single run produces a large set of correlated codependent numbers; getting an average of those would not add up on real hardware and distort conclusions where differences are small.

On the simulator results are always the same down to the last cycle; that is due to the complete absence of a memory hierarchy, which eliminates any indeterminism and showcases the algorithm. On real hardware, we sometimes got unreasonably slow runs due to interference which would unfairly gear the median toward a worse result. For most configurations, the standard deviation between all runs was between 0% and 2%; thus we refrained from plotting error bars to avoid unnecessary clutter.

Time measurements are in cycles, acquired by reading the \texttt{rdtsc} register of the underlying core. Total execution time is measured by the source worker, placed on the same core for all experiments.

Benefits

Palirria aims to maintain, if not improve, performance while using less threads by reducing cycles spent wastefully. In other words it strives to maximize worker effective utilization. Workers are not added unless immediately utilizable, not removed unless already idling. Figures 5.16 and 5.18 quantify this claim on the simulator and real hardware respectively. Their column (c) shows a comparison of execution time in cycles (X axis) versus total amount of workers used (Y axis). Separating the benchmarks into highly parallel (FIB, Skew, Stress) and irregular (FFT, nQueens, Sort, Strassen), observe that for the former group a fixed allotment is always performing better. Nevertheless, between the adaptive runtimes, Palirria achieves a smaller area bellow its plot on the simulator, resulting in less execution time, less resources, or both. On real hardware all runtime perform the same. Column (a) focuses on execution time to ease analyzing of the results. Column (b) shows the median wasteful cycles per runtime; in all runs on both platforms Palirria reduces this metric the most. An exception is the Strassen benchmark, albeit considerably Palirria outperforms all.

Figures 5.17 and 5.19 present wasteful cycles in even more detail, by showing one bar per thread. Due to space limitations, only the largest fixed allotment’s results are shown. All plots are normalized relative to the fixed allotment’s result; hence 100 is the total execution time of said run, and if another runtime ends at 110 it means 10% slower, if at 90% 10% faster. These plots directly show how Palirria utilizes less workers for irregular workloads, without performing worse than the other adaptive runtimes, and in some cases it even improves performance.

Consequently it can be claimed that Palirria achieves both removing idling workers faster reducing failed steal attempts, and transforming available concurrency into parallelism more effectively.
Performance measurements on Linux (real hardware)

(a) (b) (c)

Figure 5.18: Results for fixed 5, 12, 20, 27 workers, and self-adapting schedulers. (a) Total execution time normalized to first column. (b) Average percentage of wasted cycles. (c) Adapted number of workers over time.
### USEFUL TIME (normalized to WOOL-42) on Linux (real hardware)

<table>
<thead>
<tr>
<th>Application</th>
<th>WOOL-42</th>
<th>ASTEAL</th>
<th>DSTEAL</th>
<th>Palirria</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td><img src="chart" alt="FFT" /></td>
<td><img src="chart" alt="FFT" /></td>
<td><img src="chart" alt="FFT" /></td>
<td><img src="chart" alt="FFT" /></td>
</tr>
<tr>
<td>FIB</td>
<td><img src="chart" alt="FIB" /></td>
<td><img src="chart" alt="FIB" /></td>
<td><img src="chart" alt="FIB" /></td>
<td><img src="chart" alt="FIB" /></td>
</tr>
<tr>
<td>nQueens</td>
<td><img src="chart" alt="nQueens" /></td>
<td><img src="chart" alt="nQueens" /></td>
<td><img src="chart" alt="nQueens" /></td>
<td><img src="chart" alt="nQueens" /></td>
</tr>
<tr>
<td>Skew</td>
<td><img src="chart" alt="Skew" /></td>
<td><img src="chart" alt="Skew" /></td>
<td><img src="chart" alt="Skew" /></td>
<td><img src="chart" alt="Skew" /></td>
</tr>
<tr>
<td>Sort</td>
<td><img src="chart" alt="Sort" /></td>
<td><img src="chart" alt="Sort" /></td>
<td><img src="chart" alt="Sort" /></td>
<td><img src="chart" alt="Sort" /></td>
</tr>
<tr>
<td>Strassen</td>
<td><img src="chart" alt="Strassen" /></td>
<td><img src="chart" alt="Strassen" /></td>
<td><img src="chart" alt="Strassen" /></td>
<td><img src="chart" alt="Strassen" /></td>
</tr>
<tr>
<td>Stress</td>
<td><img src="chart" alt="Stress" /></td>
<td><img src="chart" alt="Stress" /></td>
<td><img src="chart" alt="Stress" /></td>
<td><img src="chart" alt="Stress" /></td>
</tr>
</tbody>
</table>

Figure 5.19: Useful (blue bar bottom) and non-useful (red bar top) normalized time per worker ordered by zone. Useful are the cycles spent successfully stealing and processing tasks. Plots per application (row) are normalized to the first bar (source worker) of the corresponding WOOL-42 run.

### Limitations

Palirria does not account for task granularity. It can be argued that very fine grained tasks might not offset the cost of adding more workers threads. However, the tasks being measured are stagnant. Irrespective of granularity the application at that time could indeed...
utilize more workers. Thread creation cost is highly platform dependent and could be potentially made to be very small. Thus in all evaluations it has been excluded by creating one thread for every available processor in advance. Idle and removed threads are blocked on conditionals, and it takes a signal to add them.

Alternatively, Palirria could be extended to account for task granularity if such information could be profiled statically or dynamically. Then a product of that metric could be mapped to the cost of thread creation on the current platform. Such an addition is not a small task, constituting significant future work.
Chapter 6

Pond: Exploiting Parallelism Feedback in the OS

6.1 Introduction

Current and older operating systems have traditionally allowed processes to request access to as many hardware resources they see fit, and then load balance to reduce share contention. Legacy systems with a single or just a few processors resorted to resource time-sharing between processes or threads. Hence it was never required to restrict the amount of either.

Emerging many-core architectures have allowed for space-sharing resource management, where each workload receives an exclusive allotment of resources. However, most parallel workloads do not have static resource requirements; hence allotting fixed amounts of resources would lead either to loss of performance for the application or under-utilization of the resources. Being able to adapt the allotment is fast becoming a requirement.

The previous chapter explored the ability of individual workloads to estimate their parallelism, in terms of processing cores. It also discussed how the application’s runtime can adapt to a dynamically expanding or shrinking allotment, emphasizing on the selection of best suited resources for removal. This chapter explores how to select which cores to place an application on. A space-sharing strategy in a multiprogrammed system poses constraints on such selection.

Related work has embraced self-adaptation of resources by the application. Self adaptation, besides failing to motivate honesty, is also unaware of contention. On a heavily multiprogrammed system, hazardous contention avoidance might require allotting resources below the desired amount.

Regarding requirements estimation, we have found that insight in the application’s runtime scheduling is sufficient for accuracy. Given the right metric for quantifying future work — instead of making statistical assumptions based on historical data — the runtime scheduler is by definition the broker of the available work units; with every programming model, every action related to the generation of concurrent entities — be that new threads
CHAPTER 6. POND: EXPLOITING PARALLELISM FEEDBACK IN THE OS

(3) Parallel application anatomy: data, data processing tasks, and runtime; all components can be decoupled.

(b) Traditional process model. Application and system do their own scheduling in disconnect.

(c) Process model where runtime is decoupled from computation tasks, and offered as a service by the OS; trusted requirements estimations are used in contention-aware system-wide resource management.

Figure 6.1: From the perspective that the runtime can be completely decoupled from the application, we depart from the traditional process model and migrate all thread control into a OS provided system service.

through pthreads, tasks in Cilk or OpenMP, etc. — includes a call to the runtime. Thus the runtime is in the ideal position to monitor and measure available concurrency. Even in the most complicated situation of dependency bound data-flow execution, it is the runtime which is given the dependency tree and has to analyse it. Related work has presented improvements in performance and accuracy through such mechanisms [145, 30, 75, 74, 33].

Instead of incentivizing honesty in requirements estimation we propose transferring said mechanism out of the application’s control all together. To that end, we decouple the runtime scheduler from application code and provide it as a system provided service. Figure 6.1c illustrates our proposed scheme as a departure from the traditional process model in figure 6.1b. Section 6.4 elaborates on the specifics.

We view parallel applications as the union of three components: data, data processing code, and runtime scheduling (see figure 6.1a). We need to show that runtime scheduling code can be decoupled from data processing code. We make the leap and say that this is by default true when the runtime scheduler is bundled as an external library, which is true for every programming model. Think of the lowest level of abstraction being a threading library (like pthreads). Although primitive there is still a runtime which can be emulated through Pond; pthread_create() would create a new task, assigned to a worker thread of the OS’s choosing.

6.2 Restructuring the problem

Section 4.4 expanded on a list of principles for the ideal OS scheduler of parallel applications over a many-core architecture. These principles were: Fairness in distribu-
6.2. RESTRUCTURING THE PROBLEM

tion, Efficiency in management, Distrust in requirement servicing, computational Cost-Effectiveness, and Near-Optimality of schedules. The previous section 4.3 explained why its probably impossible to adhere by all these points in solving the problem at hand, which is scheduling $n \in \mathbb{N}$ jobs arriving simultaneously with multiple criteria and multiple objectives.

Nevertheless the aforementioned principles can be met in solving an equivalent problem which transfers the indeterminism on properties that do not matter. The impossible factor is the amount of dimensions the problem has. One concession that could be made is reducing dimension in only one by serializing the scheduling process. Instead of having the scheduler work periodically on a batch of queued up requests, it can be designed as an asynchronous event driven mechanism. Scheduling actions are driven by applications one at a time on demand. There are three primary such actions:

1. Start an application on $X$ amount of resources,
2. Expand an allotment by $Y$ amount of resources,
3. Reduce an allotment by $Z$ amount of resources.

These three actions do not need to be periodical as long as the OS orchestrates while having the power to adapt their requirements. That means that variables $X, Y, Z$ which an application requests are subject to approval by the OS scheduler, depending on real time availability and overall distribution strategy. So, requests are serialized and serviced opportunistically. Such strategy is not novel and it has been shown that it can lead to severe mismanagement and significantly unfair performance degradation [134]. Any application could simply ask for all resources, forcing either starvation of subsequent requests or constant reallocation; both negative scenarios.

However the OS scheduler could be especially distrusting and frugal. It could do that by enforcing a mixture of space and time sharing strategies where the latter is used temporarily by allowing processes to win exclusive access from co-runners. Such a method could provide the first three principles; having processes win access to resources in a controlled refereed manner (Distrust) means that each has the opportunity to get what it can actually utilize (Fairness), while loosing all that it can’t (Effectiveness). The rest of this chapter explores exactly how such method was designed.

Nevertheless implementations remains an important issue. Not only the scheduler needs to be lightweight to not induce overhead, by serializing requests potential indeterminism in servicing one can severely delay all subsequent. Thus the serial computational parts must be deterministic and with bounded execution time.

the method described is not one that derives from traditional scheduling theory; it is in fact opportunistic and stochastic. Thus optimality cannot be directly evaluated although it is trivial to show that the resulting schedules are not optimum.
6.3 Process model

At the system level, resource management decisions have to consider the status of all the physically available resources to avoid over- or under-committing them. Traditionally, the allocation of resources to a workload has been agnostic of the requirements of the workload [134]. Either the system blindly follows the requests of the application, or it allots a smaller amount thus delegating adaptation responsibility to the application. However, an allocation of resources to a workload is the union of two sets; the set of physical resources the workload is allowed to utilize, plus the set of threads to run on them. Altering an allocation can mean modifying the size of either set. Current techniques have exclusively dealt with distributing physical resources among running processes. By prioritizing a space-sharing strategy, it is crucial that each thread runs on its own core exclusively. When resources are removed the runtime must optimally remove equally many threads. When resources are added, more threads must be created. Parallelism fluctuations happen at the millisecond scale [65], making either function cost-ineffective when repeated often. Scheduling that disables and re-enables threads is also not optimal [7, 107, 141].

For a workload whose parallelism has been mapped to tasks, these tasks are the only currency to be brokered. Ideally, there should be as many workers as there are tasks. In reality the runtime must use the resources that the system provides. Thus from the application’s perspective, resources should be threads. In some extent this is conceptually already in effect, although the responsibility of matching the thread count to the allotted resources is left up to the application itself. Considering a configuration where allotments are malleable at a rate of a few milliseconds, this model is not productive. The allotment of resources by the system should include the capacity to execute tasks without requiring the intervention of the application. Moreover, there is duality of roles in current parallel applications. Tasks represent an algorithm solving an arbitrary problem or performing a certain processing of data that can be independent of the execution platform. The runtime’s threads schedule the tasks for execution, with the effectiveness of the scheduling heavily dependent on the intrinsic aspects of the execution platform.

Effective system-wide resource management requires a new system software model. This model would replace the threads and process concept with a more lightweight execution abstraction of tasks. A process (or application) will then be represented by a collection of tasks and an allotment of cores (homogeneous or heterogeneous). The runtime executing the tasks would be offered by the system as part of the allotment; no scheduling logic should be included in the application itself. The size of the allotment should depend on the amount of parallelism it can currently use, on the load of the system and possibly on the priority of the application. The main advantages of this approach would be:

- Better resource utilization. No application can over-subscribe resources as only potential parallelism is expressed. An application only requests resources it can efficiently use.
- Compositional parallelism. Since the task abstraction expresses potential parallelism and is the only way of expressing concurrency, different parallel software components naturally can be composed to utilize even more parallelism.
6.4 Pond-workers: Threads as a service

In this section we will discuss the general layout of Pond, focusing on the implications of our proposed variations to the process model. Figure 6.2 depicts the anatomy of Pond; it consists of 4 components:

1. **architecture dependent model**: inspects and models the underlying architecture, it is consulted for selecting cores given sharing and contention effects.

2. **architecture independent distance-based resource manager**: handles application initial placement and allotment malleability requests.

3. **per core worker**: one per core pinned execution container for application code. They are persistent and shared among all applications.

4. (Optional) **application monitoring module**: performs application level scheduling and through self-monitoring estimates resource requirements.

The first two components handle the distribution of resources among running applications. The first component records available resources into a customizable architectural model; hardware threads are modeled directly and mirrored in the execution environment through workers (the third component); caches and cache share effects are handled indirectly through specific modeling constructs. Section 6.5 presents architectural modeling, algorithms and logic in detail. Workers are execution containers — for now think of them as kernel threads — which are pinned one per physical hardware thread. Workers are persistent, in the sense that they are created upon booting Pond; they are also shared by all applications, meaning that they can host code segments irrespective of process ownership.

The last module monitors the application’s runtime and estimates requirements. It could be optional if the application provides such mechanism using the Pond API. In the
absence of a fast mechanism for system calls, it might be prudent to not migrate the runtime into the OS. The Pond resource manager does not require it.

It could be argued that the proposed process structure follows either the 1:1 or N:M threading model. As shown in figure 6.3, The actual thread anatomy is split into three layers: i) the system threading support (i.e. pThreads as in the figure), ii) the, one per core OS level part of the worker which could be shared by iii) M number of processes, thus M different runtime schedulers. The first and second layers are unique per thread and encapsulated by a single LWP (lightweight process). The third layer has an arbitrary multiplicity, each which its own address space. However from the perspective of each application, it implements only 1 thread per LWP; thus from the application's perspective the model is 1:1. Nevertheless, from the system perspective there are multiple user threads per LWP, which however are managed and scheduled by the OS and not userspace (as with pure N:M implementations, like in [7]). Consequently Pond departs from the traditional way of reasoning about this concept.

6.4.1 Pond over a monolithic kernel

Pond-workers are designed to provide a selection of runtime scheduling techniques arbitrarily. When an application is started over Pond, it has the opportunity to select the runtime scheduling. Pond will create the process and its address space and load both the runtime library, and application code and data into the same address space. However the runtime and the requirements estimation mechanism need to be protected from the application; we do that by isolating application code and data through a sandbox. The sandbox is implemented using privilege modes, as proposed by Belay et al. [15]. Assuming an x86 architecture, the runtime, monitoring and estimation code run in ring 0, setting the supervisor bit to all their memory entries; application code is run in ring 3. Figure 6.4 provides an overview of this process, assuming sharing of workers between many applications.

When a Pond-worker is time-shared, it needs to perform context switches at a very fine granularity. To optimize this function we employ address space identifiers (ASID): TLB entries are tagged with a unique per process identifier, negating the need for flushing.
upon a switch and reducing cost [147]. ASID is implemented by Intel, AMD, and ARM, while most server-class RISC architectures support tagged TLBs. Intel calls their ASID implementation as process context identifiers (PCID); on ARM it is part of the Context ID Register. Even so, time-sharing is meant to be avoided as detailed in section 6.5.3.

The usage of tagged TLBs for lightweight context switching has been investigated by several projects [96, 147, 148, 153, 95]. Tagging TLB entries and not flushing the table upon switching, makes that table a shared resource. As such the size of the table becomes a much more significant property. Intuitively as the size of the table grows the amount of TLB misses gets reduced; however the average page walk latency increases. The type of tags and the memory access patterns of the workload are also important factors which affect overheads greatly. Venkatasubramanian et al. [147] evaluated tagged TLB usage using the Simics simulator and found the performance benefits to vary significantly due to these factors; the specific overhead reduction was between 1% and 25%.

Interaction between Pond-workers and application runtime is performed through a generic API of upcalls which can implement the functions of most programming models. We separate this API into three categories: initialization, execution, and termination. The first category includes functions for initializing runtime entities, like constructing the thread-descriptor if pthreads is used. Execution refers to the scheduling mechanics of the runtime library for parallel activity placement, like stealing and syncing for a work-stealing programming model such as Cilk. Finally termination includes the necessary calls for terminating runtime entities, like managing state when forcibly removed due to adaptation.

6.4. Pond over a micro-kernel

Additionally to implementing Pond on traditional monolithic OSs, we’ve also explored the possibility of a microkernel. One interesting part is the need to have all of Pond’s components outside the kernel. It becomes complicated due to the requirement of a two-layer protection. Pond workers need access to the memory of all running applications, meaning one unified address space. They also need to be protected from application runtime code, which as before needs to be protected from application code. One way to implement such a scheme is by having Pond and all applications as a single process, but manually assign distinct ASID for each one, then sandboxing the application as before. Although limited, ASIDs can be requested and reserved by Pond. Intel’s FCID is 12-bits long providing enough ids. An ASID on ARM architectures though is only 8 bits long, giving a non-negligible — but also not detrimental — limit of 256 simultaneous applications.

As a use case we explored Barrelfish OS [14, 13], as seen in figure 6.5. Currently Barrelfish requires significant changes in order to extend its dispatchers (a notion similar to linux’s kernel threads) with a unified memory address space across multiple kernels, and equal capabilities. An application is started on a single dispatcher (kernel) and extended to more through child dispatchers. One example of a limitation is the inability to open files through the child dispatchers, disallowing them from hosting independent Pond-workers.

---

1Due to space limitations we only include a high level description of this API, although a full specification is readily available upon request.
Furthermore, a method to load binaries into existing address spaces is another necessary extension. Delikoura [46] has worked toward a new dispatcher with said features, although her implementation is not part of the Barrellfish main tree. By design, when an application interrupts, Barrellfish will switch to the kernel scheduler, which then resumes a dispatcher instead of the application segment; Pond has the opportunity to employ custom dispatchers which (in place of workers) can execute the time-sharing logic and context switch to any application sharing the underlying physical core. This process involves 3 context switches although through tagged TLBs the cost is reduced significantly.

6.5 Scheduling

Pond is a resource manager that automatically load balances multiprogrammed systems. Our method provides the tools for automatic placement of jobs, and adaptation of their resources throughout their execution. The latter assumes the workload can continuously provide estimations of its resource requirements, truthful or not.

Building upon the idea of two level scheduling, Pond consists of the system-wide scheduler managing all processing units, and the program’s runtime which advises on usage efficiency. In this paper we assume the latter and focus on the system side. A adaptive work-stealing runtime requests a change in its allotted resource via the following API:

- **INCREASE(amount)**: request amount more cores.
- **DECREASE(amount)**: request amount number of cores released.

The system does not need to verify the aforementioned requests. It assumes that the application can effectively estimate its own requirements. To meet them, Pond tries to...
space-share first, falling back to time-sharing under certain conditions. Cores are allotted to programs optimistically; meaning that the system expects the application to be greedy and ask for a few more than it can guarantee be kept utilized. This assumption leverages unavoidable accuracy errors in the online estimations. The last allotted cores function as the testing ground for further expansion of the allotment. They are allowed to be time shared.

For the selection of cores to add to or take from an allotment, a model of the underlying architecture is consulted. Depending on the chip’s design the actual criteria might differ; however the model transforms them into an abstract geometric space where an architecturally agnostic distance measure defines and quantifies candidacy.

The following sections present each part of the scheduling method in detail.

Figure 6.6: Two examples of modeling a typical quad-core, 8 hardware thread, Intel iCore processor.

### 6.5.1 Architecture modeling

We categorize hardware resources as direct and indirect; the former refers to hardware threads which a program is directly given access to; as indirect we refer to all those resources which a program is passively assigned access to due to its placement, like certain cache banks, interconnect channels, and memory buses. We use two modeling constructs to organize hierarchies of said resources:

- **a unit**, which represents a single physical hardware thread and in effect an assignable Pond-worker.
- **a group**, as a cluster of units or other groups.
A unit is the scheduling currency; per application resource allotments consist of these units. Groups are meant to place together units that share—or explicitly shouldn’t share—a certain indirect resource. Pond strictly prioritizes allotting units within the same group. There are cases where hyper-threading might harm performance, hence hardware threads of the same core could be grouped separately, to place each application on workers of pairwise different cores. On the other hand, a group can mirror the cache hierarchy exactly when sharing low-level caches is beneficial. For example figures 6.6a and 6.6b show modeling an Intel quad-core hyper-threading processor with either strategy.

This hierarchical organization creates a multidimensional geometrical space where the distance between units quantifies candidacy for co-assigning one with the other. Each type of interconnect network defines this distance differently; it could relate to the actual link latency, or it could be a completely virtual metric that satisfies certain scheduling necessities.

The following API is provided to the higher, HW agnostic, layers of the system scheduler:

- `get_distance_of_units(a, b)`: returns the numerical distance between two units a and b, given the active architectural model.
- `get_units_at_distance_from(a, dist)`: returns a set of units at distance dist from unit a, given the active architectural model.

During this project we studied some major types of chip interconnect networks currently available. Next we describe how we defined a geometrical space for each one.

**Mesh network: TILEPro64**

EzChip’s Mesh network maps directly to a metric space, where core location is defined by a Cartesian coordinate system and geodesic distance provides the metric. Moreover, resources are not shared directly. Each core sees the sum of all remote L2 banks as one big L3. A user can define a Coherency domain over a set of cores; this domain is kept aligned with per application allotment and does not require mirroring in the architectural model.

**Algorithm 11 get_distance_of_units**: implementation for a mesh network.

```
1: function GET_DISTANCE_OF_UNITS(a, b)
2:     dist := 0
3:     if a ≠ b then dist := |a.x - b.x| + |a.y - b.y|
4:     return dist
5: end function
```

Algorithms 11 and 12 present the implementation of the model API for a mesh architecture. Their complexity is $O(1)$ and $O(4 \times dist)$ respectively. The former uses a typical Cartesian district space, while the latter iterates directly the periphery using dist as radius. In the second algorithm the buffer array should be allocated at size $4 \times dist$ which
Algorithm 12 get_units_at_distance_from: implementation for a mesh network.

1: function GET_UNITS_AT_DISTANCE_FROM(a, dist)
2:     buffer ← ∅
3:     max := 4 * (dist − 1) + 4
4:     vx = a.x + dist
5:     vy = a.y
6:     step_x = −1
7:     step_y = 1
8:     for all i ∈ [0, max) do
9:         if vx, vy within bounds then
10:             v = (vy − 1) * MaxPerRow + vx − 1
11:             if v ≠ a and v within bounds then
12:                 buffer ← v
13:         end if
14:     end if
15:     vx = vx + step_x
16:     vy = vy + step_y
17:     if vx = a.x − dist then step_x = −step_x
18:     if vy = a.y + dist or vy = a.y − dist then step_y = −step_y
19:     if vx = a.x + dist and vy = a.y then break
20: end for
21: return buffer
22: end function

is the maximum possible number of units available. The coordinates for each unit are assumed stored in a global array of bitfield structures. The algorithm also assumes known topology dimensions to calculate arithmetic bounds.

Point-to-point link: HyperTransport and QPI

P2P link networks usually have complicated designs. Cores are grouped into nodes, a socket consists of multiple nodes, and a system might have many sockets. Even within a socket not all nodes are connected to all other. Based on the grouping logic (as in figures 6.6a and 6.6b) all units (cores) within the same group are set at distance 1. Thereafter group pairwise distances are calculated following the inter-NUMA node access time, normalized to the smallest value being 2.

Algorithms 13 and 14 present the implementation of the model API for a Point-to-point link architecture. Their complexity is \( O(1) \) and \( O(N + M) \) respectively, where \( N \) is the number of groups (hardware nodes) and \( M \) the number of units within each group (cores per node). The algorithm assumes known the total number of groups (NumberOfNodes) as well as the maximum number of units within each group (CoresPerNode). Finally, it is assumed there is a two dimensional array holding the distances between all groups; this array is referenced as distances in the algorithms.
Algorithm 13 \texttt{get\_distance\_of\_units}: implementation for a Point-to-point link network.

1: \textbf{function} \texttt{GET\_DISTANCE\_OF\_UNITS}(a,b)  
2: \hspace{1em} node\_a := a.id/CoresPerNode  
3: \hspace{1em} node\_b := b.id/CoresPerNode  
4: \hspace{1em} return distances[node\_a][node\_b]  
5: \textbf{end function}

Algorithm 14 \texttt{get\_units\_at\_distance\_from}: implementation for a Point-to-point link network.

1: \textbf{function} \texttt{GET\_UNITS\_AT\_DISTANCE\_FROM}(a, dist)  
2: \hspace{1em} buffer ← ∅  
3: \hspace{1em} node\_a := a.id/CoresPerNode  
4: \hspace{1em} for all \(i \in [0, \text{NumberOfNodes})\) do  
5: \hspace{2em} if \(\text{distances[node\_a][i]} = \text{dist}\) then  
6: \hspace{3em} for all \(j \in [0, \text{CoresPerNode})\) do  
7: \hspace{4em} \(v := i \times \text{CoresPerNode} + j\)  
8: \hspace{4em} if \(v\) within bounds then \(buffer ← v\)  
9: \hspace{3em} end for  
10: \hspace{2em} end if  
11: \hspace{1em} end for  
12: \hspace{1em} return \(buffer\)  
13: \textbf{end function}

\textbf{Bidirectional ring: Intel MIC architecture}

A ring network places all cores in a 1-dimension closed loop, with bidirectional data transfers. Thus cache transfer latency is proportional to the on-chip distance. A unit consists of 1 HW thread and a group contains all units within a physical core. Distance within a group is 0. Cross group distance is set to the shortest number of hops between the two nodes. Intuitively we want to keep each allotment packed to as smaller on-chip area as possible.

Algorithm 15 \texttt{get\_distance\_of\_units}: implementation for a ring network.

1: \textbf{function} \texttt{GET\_DISTANCE\_OF\_UNITS}(a,b)  
2: \hspace{1em} node\_a := a.id/CoresPerNode  
3: \hspace{1em} node\_b := b.id/CoresPerNode  
4: \hspace{1em} return \(\min(|node\_a - node\_b|, |\text{NumberOfNodes} - node\_a + node\_b|)\)  
5: \textbf{end function}

Algorithms 15 and 16 present the implementation of the model API for a bidirectional ring architecture. Their complexity is \(O(1)\) and \(O(2 \times M)\) respectively, where \(M\) is the number of units within each group (cores per node). The algorithm assumes known the total number of groups (\text{NumberOfNodes}) as well as the maximum number of units within each group (\text{CoresPerNode}).
Algorithm 16 get_units_at_distance_from: implementation for a ring network.

1: function GET_UNITS_AT_DISTANCE_FROM(a, dist)
2:     buffer ← ∅
3:     node2 := node − dist
4:     if node2 < 0 then node2 := node2 + NumberOfNodes
5:     v := node2 * CoresPerNode
6:     for all i ∈ [0, CoresPerNode) do
7:         if v within bounds and v ≠ a then buffer ← v
8:     end if
9:     v := v + 1
10:    end for
11:    if dist = 0 then return buffer
12:    node2 := node + dist
13:    if node2 ≥ NumberOfNodes then node2 := node2 − NumberOfNodes
14:    v := node2 * CoresPerNode
15:    for all i ∈ [0, CoresPerNode) do
16:        if v within bounds and v ≠ a then buffer ← v
17:    end if
18:    v := v + 1
19:    end for
20:    return buffer
21: end function

6.5.2 Space sharing

Pond primarily space shares the available workers, by pinning them on processors exclusively. The rigidity of such scheme requires optimized selection algorithms. The process can be broken down into three tasks: initial placement, allotment size increase, and decrease. When applications terminate the same principles apply as a allotment size decrease.

Initial placement: selecting a source core

Based on our resource management strategy it is ideal that new programs are started with a high potential for allotment expansion. The initial worker – which we call the source of the allotment – should be placed centrally to a large idle area of the chip’s network.

We use algorithm 17 to pick the next candidate source as the center point of the largest idlemost area, as defined by the architectural model’s metric space. First the set of idle cores is iterated; the core selected has the smallest sum of inverse distance from all active sources. This means that this core is the furthest away from all other active programs. If no idle core exists the same algorithm is applied to cores allotted, but not the source, to other programs.
Algorithm 17 GetNextSource: Pick worker in biggest idle area as next source.

```java
1: function GETNEXTSOURCE(Idle, Busy, Sources)
2: Source := MinWeightCore(Idle, Sources)
3: if Source ≠ NULL then
4: return Source
5: end if
6: Source := MinWeightCore(Busy, Sources)
7: return Source
8: end function
```

Algorithm 18 MinWeightCore: Select worker furthest away from all existing sources.

```java
1: function MINWEIGHTCORE(Workers, Source)
2: MWC := NULL
3: wmin := MAXIMUM_DISTANCE
4: for all c ∈ Workers do
5: wc := \sum_{j∈Sources} \frac{1}{MinDistance(c,j)}
6: if wc < wmin then
7: wmin := wc
8: MWC := c
9: end if
10: end for
11: return MWC
12: end function
```

Allotment increase: choosing cores to expand a running allotment with

Much of this project’s value stems from facilitating dynamically adaptive resource allotments. We believe that only the OS has the capability to ensure optimal resource selection on a multiprogrammed system. The application only requests a desired number of extra cores and the system scheduler decides if it can meet the request and with which specific cores.

Algorithm 19, selects those cores that are closest to the source of the requesting program. First it iterates the set of currently idle cores. If the requested amount is not met it tries to find allotted but time-shareable cores (see section 6.5.3). It might happen that the requested amount is not met.

Allotment Decrease: choosing cores to remove from a running allotment

Removing workers from an allotment is a slow process. Firstly, there is no constraint against meeting the requested amount, except removing the source. As per algorithm 20, the workers furthest from the source are selected. They are eventually phased out given the requirements of the programming model. Pond notifies the runtime which must enforce starvation upon the worker and eventually remove it. For example with work-stealing
Algorithm 19 Increase: Add workers, most idle and closest to the source

1: function INCREASE(Amount, NotAllotted)
2: \[ R := \emptyset \]
3: \[ L := \emptyset \]
4: while Amount > ||R|| and ||NotAllotted|| > 0 do
5: \[ c := \text{pop}(\text{NotAllotted}) \]
6: if 0 < assignments(c) < LIMIT then
7: \[ L := L \cup \{c\} \]
8: else
9: if assignments(c) = 0 then
10: \[ R := R \cup \{c\} \]
11: end if
12: end if
13: end while
14: while Amount > ||R|| and ||L|| > 0 do
15: \[ R := R \cup \{\text{pop}(L)\} \]
16: end while
17: return R
18: end function

Algorithm 20 Decrease: Remove workers furthest from the source

1: function DECREASE(Amount, Allotted)
2: Output: R: set of workers to release
3: \[ R := \emptyset \]
4: while Amount > ||R|| and ||Allotted \setminus R|| > 1 do
5: \[ R := R \cup \{\text{head}(\text{Allotted})\} \]
6: end while
7: return R
8: end function

workers can finish processing their task queue but cannot steal new tasks, while getting stolen from also.

6.5.3 Time Sharing

Exclusively space-sharing all cores requires some method for resolving simultaneous requests for the same resources. Traditionally, space sharing schedulers either allot arbitrarily or follow rigid patterns \cite{5, 157}; in both cases allotments are exclusive. In our scheme of spatial partitioning, time-sharing is employed on spatially boundary workers only as a testing ground for an application’s ability to increase its utilization. Through increase requests, applications can express their ability to fully utilize a time-shared core; the resource-manager then has to decide giving exclusive access of the time-shared worker
to that application. If many co-tenants have issued \textit{increase} requests during the same time quantum, the worker remains shared.

Figure 6.7 presents the simple access state protocol, assuming sharing between only two applications for simplicity.

Figure 6.7: State machine for time-sharing a worker between applications $a$ and $b$.

This sharing mechanism allows Pond to force applications to prove their resource demands even when being dishonest. By implementing time-sharing only on workers that are spatially the furthest away from the source worker, the spread — and in effect the dominance — of one application over others is constrained. Thus, even if one process’ parallelism feedback is honest among dishonest co-runners, it has an opportunity to maintain its desired amount of resources, because the aggressive co-tenants’ requests are not automatically met.

Nevertheless, Pond’s accuracy in identifying dishonesty could be improved with the help of hardware events monitoring; to some extent there can be some simple correlation of cache usage with useful utilization of a core; the reasoning is simple and quite naive: a spinning worker should not be accessing the cache much. Such mechanism has not been implemented in the current prototype.

A worker allotted to more than one application, keeps track of its \textit{distance from the source} for each. A worker selects for execution the application with the smaller allotment and for which it is closer to its source. However, every time an application is selected, the worker penalizes it by increasing these values. In the next iteration it will be harder for the same application to be selected. Selecting a different application resets the values. The actual pseudo-code is presented in algorithm 21.

6.6 Experimental Methodology

For evaluating Pond we’ve used a small set of task parallel applications, exhibiting varying parallelism irregularity and diverse memory requirements. Our aim was to cover significant
Algorithm 21 Run: Decides next application to run.

1: function Run(Apps)
2: sz := ∞; hc := ∞
3: P := NULL; A := NULL
4: while True do
5:     for all a ∈ Apps : a is active do
6:         if a ≠ A and a.sz ≤ sz and a.hc < hc then
7:             sz := a.sz
8:             hc := a.hc
9:             A := a
10:         end if
11:     end for
12:     sz := sz + 1
13:     hc := hc + 1
14:     A(A.args)
15: end while
16: end function

cases although our options were limited by the absence of open source adaptive runtime schedulers for most programming models. The applications we used are: fft, nqueens, sort, and strassen from the BOTS benchmark suite [52], fib and cholesky from the Cilk-5 package [66]. All applications were ported to the Palirria adaptive runtime scheduler [145].

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem size</th>
<th>Arguments</th>
<th>Recursive depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>4000 10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>2^{26}, 2048, 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strassen</td>
<td>4096 64, 64 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>2^{27}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nqueens</td>
<td>14, 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Application input data sets for the evaluation of Pond. Values given as "problem size arguments, recursion depth cut-off", while some don’t use a cut-off.

The data input for each application was selected so that it controls the requirements pattern but also provide a run time between 8 and 10 seconds, so as to allow adaptation events to have a measurable effect. Also, similar execution time between all applications was important for the types of experiments we conducted. These input values are shown in table 6.1. Finally, the requirements estimation was setup with a long 300ms interval time and a high sensitivity for increments. Such settings reduced adaptation overhead but provided a quick and certain convergence to the ideal allotment size.

For simplicity we chose to implement everything in a single binary. The implementation is such that the Pond scheduler, the runtime, and the application code are all completely decoupled. Via an integrated prompt the user has the choice of executing application with Pond or not, selecting the runtime scheduler, and disabling thread pinning. One thread per core is created and blocked on a condition before any commands can be issued.
Disabling the Pond scheduler and thread pinning as well as running only a single application, is equivalent to executing over the normal Linux scheduler; we will call this a Linux run. A multiprogrammed Linux run refers to multiple simultaneous instances of our binary, running one application each. Thus each application can get as many threads as there are physical cores. A Pond run refers to an application running with threads pinned and Pond managing cores. A multiprogrammed Pond run means one process running multiple applications within. Applications must all share threads totaling the number of physical cores. We will write group run when generally referring to a specific group on either Linux or Pond.

We executed each application in isolation and in all possible groups of 2, 3, and 4. All results are the median of 10 runs. For each execution we measured the execution time in cycles of each co-runner separately. Out of this we derived three metrics used in the evaluation and related figures. First the execution time for each application in isolation as a baseline, and in each group it partakes i) on Linux, and ii) on Pond. We provide a dedicated plot per application, to compare its performance degradation in absolute numbers.

Next we calculated the performance degradation variance (pdv) for each group run, as the standard deviation between the individual performance degradation of each co-runner. This variance metric is coupled in the same plot with the relative worst degradation (rwd) for that group run, as \( \frac{\text{performance on Linux}}{\text{performance on Pond}} \) of the application that had the worst degradation for each group; note that it might be different applications on Pond and Linux suffering worst performance degradation.

As an example of goodness criteria, assume a group of 3 applications: the pdv on Pond is 10% and on Linux 5%, with rwd at 1.1. This means that the Linux run had a more uniform performance degradation among co-runners, however the worst affected application executed faster on Pond by 10%. Indirectly this shows that all other applications must have been faster on Pond too. Since the variance on Linux is 5%, we can infer that all applications on Linux suffered higher performance degradation than on Pond.

All runs where performed on Linux (v2.6.32) running on real hardware. The architecture is Opteron 6172 (AMD Magny-Cours) ccNUMA system with a total of 48 cores. There are 4 sockets, holding 2 NUMA nodes each. Each node has 6 processing cores and 8GB of locally controlled RAM. Threads were pinned using pthread affinity while the system was running a minimum of necessary services. Pond modeled this architecture as in figure 6.6a.

### 6.7 Evaluation

<table>
<thead>
<tr>
<th>application</th>
<th>allotment size</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>38</td>
</tr>
<tr>
<td>fft</td>
<td>42</td>
</tr>
<tr>
<td>fib</td>
<td>48</td>
</tr>
<tr>
<td>nqueens</td>
<td>42</td>
</tr>
<tr>
<td>sort</td>
<td>28</td>
</tr>
<tr>
<td>strassen</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 6.2: Application ideal allotment size in number of worker threads.
6.7. EVALUATION

Our evaluation strategy consists of two experiments. The first assumes dishonest requirements estimations by having applications request a constant number of threads; we chose the amount of threads that gives best performance in isolation for each application as shown in table 6.2. With any group the total amount of threads exceeds the availability of cores. Consequently a Linux run oversubscribes the system, while Pond has to forcibly resize allotments and time-share workers. We consider this environment adversarial and will be referring to it as such.

The second scenario uses trusted estimation requirements either by migrating the runtime to the system, or by assuming honest application feedback. Applications are started with 1 thread, and let adapt their allotment progressively. A Linux run still oversubscribes the system although considerably less. Similarly a Pond run will produce less time-sharing.

![Figure 6.8](image1.png)

Figure 6.8: Multiprogramming using non-adaptive runtime. The Y axis is in log-10 scale. The X axis considers the worst performance degradation for each group.

![Figure 6.9](image2.png)

Figure 6.9: Multiprogramming using adaptive runtime. The Y axis is in log-10 scale. The X axis considers the worst performance degradation for each group.

Figures 6.8 and 6.9 reuse the results from chapter 4 figures 4.1 and 4.2 respectively, combined with results with using Pond. To recap, the first experiment emulated adversarial conditions where applications requested only their ideal maximum number of cores. The second assumed trusted requirements estimations, adapting application allotments appropriately. In the first case (fig 6.8) Pond reduced degradation variance from 122% to 50% on average while achieving less degradation by a factor of 3.2x. In the second experiment, Pond reduced degradation variance from 49% to 18% on average while again co-runners degraded performance by a factor of 2.9x less, as shown in figure 6.9.
6.7.1 Adversarial Scenario

In this section we discuss the results where applications start with their ideal number of threads, and only request said number.

Figure 6.10 shows the $pdv$ as the standard deviation of the performance degradation between collocated applications. Pond run results range from 3.3% ($fib, sort$) to 104% ($fib, nqueens, cholesky, strassen$), with an average and median value of 50%. Linux run results range from 10% ($fft, sort$) to 306% ($fib, nqueens, cholesky, fft$), with an average of 122% but a median of 43%. Thus for most groups Linux resulted in more uniform performance degradation per co-runner.

However, looking at per application performance in figure 6.11, all are consistently faster in Pond runs. This observation is supported by the $rwd$ plot where the median is 3.26, meaning that for most groups the application that degraded performance the most, was impacted 3 times less.

There are no cases were Linux achieved better results; however there are a few that matched Pond performance, the leftmost groups on figure 6.10. Interestingly the combination of $fib, nqueens, strassen$ gave lower $pdv$ on Linux, meaning that all applications suffered equal slowdown. Whereas on Pond, variance is bigger meaning that some applications executed faster than on Linux. The $rwd$ metric compares the worst degradation, hence a large variance means all other co-runners managed less degradation.

In overall, Pond resource management produced better per application performance and lower degradation variance under adversarial conditions. This experiment serves to show how Pond’s time-sharing mechanism dynamically enforces allotment adaptation and on-demand exclusive access to hardware cores even when the applications are not willing to adapt.
Figure 6.10: Using dishonest requirements estimations, performance degradation variance on the left Y axis on Linux (red) and Pond (blue), and relative worst degradation on the right Y axis (green). X axis shows the applications in each group.
Figure 6.11: Per-application performance with dishonest requirements estimations. X axis shows the co-runners.
6.7.2 Trusted Requirements Estimation

In this section we discuss the scenario where applications start with only 1 thread, and let to adapt autonomously providing trusted requirements estimations.

A first observation is that adaptation improved Linux results, but so it did for Pond runs too. Figure 6.12 shows the performance degradation variance, where Pond runs achieve lower values on average and median at 19%, versus 48% average and 39% median for Linux runs.

Of course a low variance is insignificant if the actual per application performance is low. The plots in figure 6.13 show the execution time in cycles for each application separately. Pond runs exhibit performance better than Linux overall. Median worst relative performance for Pond runs is 2.89 better than Linux. Individual application execution time is also lower in Pond runs, with very few exceptions.

There are two cases where Linux runs gave better rwd and lower variance; namely fib, sort and fib, nqueens. Regarding the former group, fib achieved significant better performance than Pond; sort execution favored the Pond run. Sort is an application that requires almost half the cores fib does on out test system; moreover, the embarrassingly parallel fib’s performance is impacted significantly with every single core adjustment. Consequently, Linux’s equal opportunity strategy gave more CPU time to fib than to sort on shared resources, than Pond.

The fib, sort group’s performance is a great example on how Pond could benefit from other techniques like lightweight online profiling using hardware counters. Such data could augment the distance and allotment size conditions a worker uses to select the next application to run. Example such data could be the utilization levels, which coupled with insight on the runtime scheduling can isolate useful from wasted cycles.

The second interesting group fib, nqueens is actually a false negative for Pond. As it is observable in the per application plots, both fib and nqueens executed faster on Pond. Fib suffered worst degradation with the Pond run, but nqueens with the Linux run. It so happened that the degradation of nqueens on Linux was less than that of fib on Pond giving a lower rwd to Linux, even though both applications individually suffered greater degradation on Linux than on Pond.

There were no other groups that gave better performance for all applications on Linux; although there are a few cases were pdv is lower, it was due to greater slowdown and not performance improvement.
Figure 6.12: Using trusted requirements estimations, performance degradation variance on the left Y axis on Linux (red) and Pond (blue), and relative worst degradation on the right Y axis (green). X axis shows the applications in each group.
Figure 6.13: Per-application performance with trusted requirements estimations. X axis shows the co-runners.
6.7.3 Limitations

Primarily Pond has been design with space-sharing partitioning as the favorable strategy. This means that by design Pond assumes an abundance in hardware resource availability, where there exists at least one physical core per workload. The proposed design also requires that source workers are not shared; this means that there is an indirect limitation imposed on the number of simultaneous applications, to the value of existing physical cores. Nevertheless, this limitation is easily mitigated in a proper implementation of Pond within the OS kernel space. Current issue for not having source workers shareable is the inability to escape application code. Employing interrupts however can easily change this.

The presented design of Pond works for compute-bound applications with any memory requirements. However it does not consider I/O or other types of resources except processing cores and the cache attached to it, arguably an important shortcoming for considering general usability.

Finally, there is no straightforward method to predict expected performance degradation, even though neither do current schedulers, which would be a necessary feature for HPC and datacenter deployment; when performance really matters, like in HPC environment, the preferred solution is to waste hardware by placing applications on dedicated set of resources [105, 69], even though there has been a lot of research on optimizing collocations.

6.8 Related Work

Regarding runtime migration to system space, Apple’s Grand Central Dispatch [8] follows similar principles as our project in offloading thread control to the OS; applications use system calls to register tasks while the OS decides on thread count and scheduling. The underlying XKU thread scheduling kernel utilizes varied priority queues to determine task execution urgency. Contrary to Pond, this mechanism depends on the application to tag its tasks appropriately.

Recent OS projects have dealt with multiprogramming manycores [45, 17, 87, 45, 73]; Tessellation OS [39] partitions available resources into containers, Cells, and employs a thread scheduler within each cell.

Techniques targeting cache contention could be used in parallel to Pond to further improve core selection. Such projects either predictively model cache usage patterns [28, 22, 32, 49], or monitor cache events mapping them to application behavior [106, 11, 104, 136, 137], or leverage both for contention-aware scheduling [86, 63, 34].

SCAF [43] leverages performance monitoring and runtime adaptability feedback to partition resources. Iancu et.al. [83] explored resource oversubscription to address fluctuating requirements. Bini’s et.al. ACTORS model [18] abstracts threads as virtual resources mapped to physical cores on deadline priorities for embedded systems. Weng and Liu [151] used $CPI_{mem}$ to estimate resource requirements, as feedback for improving resource partitioning.

Targeting fair QoS, several projects have proposed fairness models [84, 113, 85, 116], hardware and platform features [91, 77, 117].
Finally, there has been work proposing OS contention aware resource management based on application performance estimation, via a mixture of the aforementioned techniques on a variety of platforms [94, 62, 5, 93].
Part III

Epilogue
Chapter 7

Summary and Future Work

7.1 Thesis summary

Emerging multi- and many-core architectures transitioned software development to parallel paradigms, thus bringing forth the issue of efficient job placement for maximizing performance and minimizing contention. Scheduling for multiprogrammed parallel architectures has been explored extensively with SMP configurations. However modern CMP enabled new very fine grained compute-bound types of workloads, which have not previously been considered to the same extent. Fine grained parallelism poses a scheduling problem since it cannot be profiled online in a timely manner, nor offline since its profile varies due to many factors like platform and architecture, as well as frequently exhibiting fluctuating parallelism during their execution.

This thesis embraced two ideas, namely two-level scheduling and task-parallelism, as its foundation. Chapters 3 and 4 investigate the trade-offs between various scheduling strategies, contributing a theoretical base on which to build a solution to the problem at hand. Then chapters 5 and 6 propose and evaluate a solution which touches upon all parts of the stack, with almost no difference on the established interfaces between components and layers.

Chapter 2 explores architectural characteristics of modern CMPs and their effect on software. This thesis axiomatically accepts the implications these novel hardware designs bring onto software. The abundance of processing cores coupled with elaborate cache hierarchies and complicated coherence protocols increase the complexity on partitioning said resources among running applications. The various interconnect networks employed in connecting the plethora of components add one more dimension of complexity in scheduling, making locality an important criterion. In modern multi- and many-core architecture various components can be shared, in or outside the processing unit; this sharing can result in significant contention effects, extensively degrading performance when multiprogramming. Hence the important properties an effective scheduling strategy should address are to increase locality between single process allocations, reduce contention both within threads of the same process and those of others.
Chapter 3 discusses the state of the art in programming paradigms for efficiently utilizing modern CMPs. After a short introduction of the vast landscape of choices, a reasoning is given on why task parallelism is a solid choice to further investigate; task-parallelism can accurately model parallelism in most forms; it is also able to handle very fine grained parallelism. The rest of the chapters discusses strategies for runtime scheduling of task-parallel applications, emphasizing on work-stealing the strategy used in this thesis. Current work-stealing schedulers employ randomness in selecting task placement, which hinders uniform placement and in effect quantifiable approximation of the available parallelism. The chapter serves to identify a set of required properties for a runtime scheduler in order to enable accurate, lightweight, dynamic adaptation without reducing performance.

Chapter 4 delves into the system layer, establishing the role the operating system scheduler plays in effective resource distribution. It redefines fair scheduling as equal relative performance degradation, instead of equal access; the latter — a traditionally used approach from the era of single processor machines — proves inefficient on modern CMPs due to the fact that lack of application parallelism will produce excess synchronization which is wasteful use of CPU time. Alternatively applications should be given resources proportional to their true individual requirements for optimal performance, given overall demand and contention. the chapter end by listing necessary properties for a OS scheduler in order to provide the desired benefits.

Moving on to the next part of the thesis, chapter 5 defines specific principles for a dynamic online parallelism estimation method, being a hardware agnostic, history independent parallelism metric, a complete and relaxed decision policy for the sufficiency of an allotment’s size, and a lightweight and scalable implementation to make the whole method practical. To that end two individual components are presented: DVS and palirria. DVS is a simple drop in replacement for any work-stealing scheduler’s victim selection algorithm; it provides uniform task placement across workers and performance improvements for special parallelism profiles. Palirria builds upon DVS to provide a mechanism for approximating currently available parallelism; the approximation is based on actual future work, using a sample which converges to a fixed size as the allotment size expands.

Finally, chapter 6 introduces and evaluates Pond, a novel take on the principles of scheduler activations for system level resource partitioning and thread management. Pond does away with the N:M threading model for 1:1 threads that are shared by all processes; further more it proposes a design for migrating thread control from the application to the OS, enabling trusted parallelism estimation and fast, lightweight context switches without TLB flushes. Resource distribution is performed on a hybrid space and time-sharing basis.

7.2 Contributions

The contributions attributed to the work performed within the thesis include:

- An original analysis on the trade-offs of different resource partitioning strategies at the system on multiprogrammed multi- and many-core architectures,
7.3. **FUTURE WORK**

- An original analysis on the implications of different scheduling strategies at the user level for dynamically adaptable task-parallel runtime schedulers,
- A novel deterministic approach to work-stealing scheduling of task-parallel applications, with mathematically proved properties and evaluated performance,
- A novel method for the estimation of the resource requirements of task-based work-stealing parallel applications, with mathematically proved properties and evaluated performance,
- Comparative evaluation of closely related work on semi-random work-stealing scheduling with and without a cycle based requirements estimating method,
- A novel implementation design of system provided runtime scheduling for trusted parallelism estimation, with lightweight context switching,
- A space- and time- sharing hybrid resource allocation method for uniform performance degradation of co-runners on multiprogrammed multi- and many- core architectures.

Some work included in this thesis was conducted in collaboration with others.

- The systems group at ETH and Time Harris of MSR provided sufficient thread support in Barrellfish and established guidelines on building a thread manager on top of the multikernel.
- Karl-Filip Faxén, developer of the WOOL work-stealing runtime, assisted in porting WOOL to Barrellfish, and also advised on integrating DVS and Palirria implementations into WOOL.
- Artur Podobas, PhD student at KTH ICT, assisted in porting the BOTS benchmarking suite to WOOL.
- Ananya Muddukrishna, PhD student at KTH ICT, assisted in characterizing task parallel applications and specifically those included in the BOTS benchmark suite.
- Chau Ho Bao Le, KTH MSc student, ported the Cilk++ work-stealing runtime library to Barrellfish [12].
- Eirini Delikoura, KTH MSc student, developed a prototype Barrellfish implementation for threads shared between multiple processes.

### 7.3 Future work

The field of parallel runtime scheduling has been the focus of extensive research for some decades now. Thus there have been a large number of proposed methods with as many implementations. This has led to fragmented solution space making it very difficult to devise
generic solutions for the cooperation between application and operating system. Moreover
CMPs multiprogrammed with fine-grained parallel applications have only recently started
to be thoroughly investigated. Consequently there is plenty of ground to be covered before
getting to a comprehensive solution. The rest of this section iterates several improvements
to the contributions of this thesis in order to generalize their functionality.

- **Data dependency resolution support in Palirria**

  The current design and implementation of Palirria does not directly support some
advanced features of task-parallel runtime schedulers. Most prominent among them
is data-dependencies for emulating dataflow programming. Palirria is design to
count every spawned task as instantly available parallelism. By introducing data-
dependencies tasks are not ready before these dependencies are resolved. A mecha-
anism which allows to differentiate between ready and not ready tasks in a cost-
effective manner would allow Palirria to only count the ready tasks. A naive ap-
proach would be the implementation of two task-queues: one for placing tasks upon
spawning them, transferred to the second one when ready. Palirria would then only
iterate the second task-queue.

- **Dynamic Palirria interval**

  The interval at which Palirria is invoked to decide parallelism availability is a criti-
cal factor to its accuracy. Workloads can exhibit rapid fluctuations which if captured
via a very tight interval can lead to thrashing by continuously alternating between
utilization states. If the interval is long then Palirria might be slow in addressing
changes, missing opportunities for improving performance further. Our evaluation
used a manually set interval length, the same for all workloads and implementa-
tions, which could potentially be suboptimal. To that end, a mechanism which could
dynamically adjust the interval length could prove beneficial. Current Palirria imple-
mentation already support a feature the detects rapid fluctuations and discards them,
until the same non-balanced outcome persists; this mechanism depends on the inter-
val to be quite small however. A more sophisticated mechanism would be to work
with subintervals and a continuously adjusted primary interval length. The subin-
tervals can gather samples, while the primary interval — every $N^{th}$ subinterval —
could use the samples to make decision. By adjusting $N$ the primary intervals length
could become more or less sensitive to small fluctuations.

- **Implement and evaluate Threads-as-a-Service inside an OS kernel**

  Chapter 6 proposed a design on how user threads could be provided as a service by
the OS. This design consists of two major components: 1) providing a thread like
execution environment, one pinned on each physical core, and 2) implement memory
protection through privilege modes. The first component is trivial to accomplish.
However the second requires a radical redesign of memory management. Use of
tagged TLB is mandatory, a feature which is not standard and has been implemented
differently by all major processor providers. Barrelfish is an interesting platform
to implement such features; E. Delikoura’s work [46] resolved the first issue of
having OS provided threads dynamically load user binaries, still missing the memory protection sandbox implementation.

- **Extend Pond to handle generic parallel and non parallel applications**

  A system scheduler cannot be customized to favour only a certain type of applications. For Pond to be truly efficient, it requires to consider contention from applications which it is not handling. These can be parallel application that do not provide parallelism feedback and non-parallel applications. For Pond it does not really matter if the applications providing feedback are task-parallel as long as the feedback follows the principles discussed in chapter 6. Support for non parallel applications is straightforward, having an application started on its source worker but never requesting to expand its allotment. Applications that do not provide feedback will either start with a specific amount of threads fixed throughout their execution, or use thread control commands to privately alter their allotment size. These calls can be captured by Pond and treated as requests and either service them or reject them by having said call return an error code, which is normal behaviour for most threading libraries.

- **Improve Pond accuracy by considering hardware monitoring information**

  Parallelism feedback acquired through the semantics of runtime scheduling cannot provide information on the application’s behaviour towards memory. When deciding to time-share cores or share cache banks knowing the expected cache requirements can be the key for improved placement. Of course with task-parallel applications this has little gain unless all of the application’s tasks have uniform requirements, since the worker could steal any task with varying memory footprints. Nevertheless, online hardware monitoring can only profile tasks that have already been executed and it is highly possible that they will differ from tasks further down on the task tree.
Bibliography


[58] EZChip. EZchip Introduces TILE-Mx100 Worldâ€™s Highest Core-Count ARM Processor Optimized for High-Performance Networking Applications, 2015.


