

Co-Scheduling oder Kein Co-Scheduling? Effiziente Ausnutzung von Großen Mehrkernrechnern

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Barbara Anna Sarközi, BSc

Matrikelnummer 01633043

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Wien, 17. Oktober 2021

Barbara Anna Sarközi

Sascha Hunold



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

To Co-Schedule or Not To Co-Schedule? Efficiently Utilizing Large Multicore Machines

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Barbara Anna Sarközi, BSc

Registration Number 01633043

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Vienna, 17th October, 2021

Barbara Anna Sarközi

Sascha Hunold



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Barbara Anna Sarközi, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. Oktober 2021

Barbara Anna Sarközi



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

This thesis was an adventure, which is going towards its end. It would not have been the same great experience without my supervisor, Prof. Sascha Hunold. I am extremely grateful to have had the opportunity of crossing paths with him and being able to work together on a research topic. At the start of this work, I was worried about finding enough motivation to work on the same topic for several months and then having enough energy to finish the work. As it turned out, there is only one factor relevant for being motivated and having energy for the work, which is fun. Looking back at the past months, I can honestly say that it was exciting and fun working on this thesis topic. Who would have thought that the master thesis would eventually be one of the most thrilling assignments of my university education? And I am confident that this pleasant environment was built by my supervisor, Prof. Hunold: he always made time in his tight schedule to fit in weekly or two-weekly meetings, he found encouraging words, even if something was going wrong, he was always excited to see results and to work together on challenges. I feel very lucky having experienced this type of work, because I know that it is rare. In this sense: Thank You, Sascha, for this opportunity and Your constant motivation and support, thank You for making this thesis work such a lovely experience, I will miss working together!

I would also like to appreciate the encouragement and moral support of my family and friends, who always listened to my problems, brainstormed together with me, and helped in any other possible way they could.

Last but not least, I want to thank my mum, who always supports me and is doing her best helping me achieve all of my dreams and goals.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Hardwarekomponenten in Rechenclustern und Computersystemen entwickeln sich fortlaufend weiter, was zu einer steigenden Anzahl an Prozessorkernen in Mehrkernrechnern führt. Der leistungsfähigste Supercomputer in Österreich ist der VSC-4 mit 37 920 Prozessorkernen. Um diese enorme Anzahl an Kernen effizienter nutzen zu können, müssen parallele Anwendungen systematisch auf den Rechenknoten ausgeführt werden, beispielsweise durch gleichzeitige Ausführung, durch Co-Scheduling, auf einem Knoten.

In dieser Arbeit wollen wir vorhersagen, ob zwei parallele Anwendungen auf einem Mehrkernrechner mit oder ohne Co-Scheduling ausgeführt werden sollen. Die Anwendungen sollen nur gleichzeitig ausgeführt werden, wenn die Laufzeit des Co-Schedulings die Laufzeit eines dedizierten Durchlaufs nur minimal verändert. Das Ziel ist es, das Potenzial eines Co-Schedulings lang laufender Anwendungen durch Messungen kurzer Durchläufe vorherzusagen.

Zuerst untersuchen wir Laufzeit- und Skalierungsverhalten von diversen OpenMP Anwendungen. Wir bestimmen Co-Scheduling-Strategien, die zu Ressourcenkonflikten von Kernel Teilen führen. Weiters untersuchen wir das Laufzeitverhalten von Anwendungen durch das Messen von Hardware-Performance-Events auf den Prozessoren. Zuletzt erstellen wir ein Vorhersagemodell mittels logistischer Regression und sagen so das Co-Scheduling-Potenzial zweier Anwendungen mit Hilfe von Performance-Zählern vorher.

Wir stellen fest, dass das Ausführen von Anwendungen auf unterschiedlichen Sockets zu keinen Ressourcenkonflikten führt, jedoch dass das Teilen von Sockets mit einer verstreuten Zuordnung zu höheren Kernelzeiten führen kann. Weiters zeigen wir auf, dass es notwendig ist, Kernel-Programmenteile zu synchronisieren. Dafür stellen wir unsere neuartige Synchronisierungsbibliothek vor. Mit Hilfe unserer Vorhersagemodelle ist es möglich, das Co-Scheduling Potenzial zweier Anwendungen vorherzusagen. Dies stellt eine gute Möglichkeit dar, Mehrkernrechner effizienter zu nutzen.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Hardware components of computing systems are improving constantly, which leads to an increasing number of cores on multicore machines and computing clusters. The most powerful supercomputer in Austria, the VSC-4, has 37 920 cores, and it is important to utilize the cores efficiently. Therefore, it is necessary to execute parallel applications methodically, e.g., by simultaneously executing (co-scheduling) them on compute nodes.

In this work, we predict whether two parallel applications running on a multicore machine should be co-scheduled or not. Two applications should only be co-scheduled, if there is at most a small slowdown of the co-scheduled runtime compared to the dedicated runtime of an application. We are interested in predicting this co-scheduling potential of long executions by sampling a short execution.

We start by assessing runtime and scalability behaviors of diverse OpenMP applications and determine co-scheduling strategies leading to resource sharing conflicts of compute kernel sections. Then, we analyze hardware performance counters and select a subset of relevant counters indicating slowdowns. Finally, we create a prediction model using logistic regression and predict the co-scheduling potential of two applications using performance counters.

This thesis shows that executions on different sockets do not lead to resource sharing conflicts, whereas sharing sockets with a scatter affinity mapping lead to increased kernel times. We show the necessity of synchronizing kernel sections of co-scheduled applications, and therefore introduce a synchronization library. Our prediction models demonstrate the possibility of predicting co-scheduling potentials of two applications, which represents an efficient way of utilizing multicore machines.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

List of Figures	xv
List of Tables	xvii
1 Introduction	1
2 Background and Related Work	7
2.1 Background Information	7
2.1.1 Multicore Architecture on Shared Memory System	7
2.1.2 What is OpenMP?	10
2.1.3 Measuring Parallel Computing Performance	10
2.1.4 Scheduling and How to Solve Co-Scheduling Problems	12
2.1.5 Affinity Mappings	13
2.1.6 Regression Analysis	14
2.2 Overview of Used Benchmarks	16
2.2.1 Rodinia OpenMP Applications	16
2.2.2 SPEC OMP2012 Applications	20
2.3 Related Work	22
3 Co-Scheduling Preparation: Parallel Scalability and Scheduling Potential	27
3.1 Novel Execution Framework for Co-Scheduling Applications	28
3.2 Single Core Execution Behaviors	30
3.2.1 Execution Experiments with Rodinia	32
3.2.2 Execution Experiments with SPEC OMP2012	39
3.3 Parallel Scalability	44
3.3.1 Scalability of Rodinia Benchmarks	44
3.3.2 Scalability of SPEC OMP2012 Benchmarks	47
3.4 Performance Potential of Co-Scheduling	49
3.4.1 Definition of the Integer Programs	49
3.4.2 Evaluating the Optimization Potential of Co-Scheduling	53
4 Characterizing Co-Scheduled Applications With HW Performance Counters	59

4.1	Configurations for Measuring the Influence of Co-Scheduling	60
4.2	The Problem of Measuring Co-scheduled Applications	62
4.2.1	Synchronization of Programs as the Solution	63
4.3	The Influence of Co-Scheduling with Different Scheduling Configurations	67
4.3.1	Co-Scheduling Experiments	69
5	Prerequisites for Predicting Co-Scheduling Behaviors	77
5.1	Correlation of Performance Metrics with Time	77
5.1.1	Difficulties with likwid Performance Counters	78
5.1.2	Correlation of PAPI Performance Events	80
5.2	Limited Group of Performance Metrics Relevant for Prediction	83
5.2.1	Finding Good Representatives	84
5.2.2	Selecting Relevant Performance Events	90
6	Predicting Co-Scheduling Potentials	91
6.1	Prediction Model Idea	91
6.2	Building a Prediction Model	92
6.2.1	Data Normalization	93
6.2.2	How to Choose Program A?	94
6.2.3	Functionality of the Prediction Model	94
6.3	Evaluation of the Prediction Model	94
6.3.1	Test and Train Data Sets	95
6.3.2	Distribution of PAPI Events between Training and Validation Sets	96
6.3.3	Evaluating the Logistic Regression Models	99
7	Conclusion and Future Work	103
	Bibliography	105
	Appendices	111
	Code Template for Measuring PAPI Performance Events	111
	Pseudo Code of the Prediction Model	112

List of Figures

1.1	Supercomputer Trend Data	1
1.2	Relation between Running Time and Parallel Efficiency	2
1.3	Thesis Overview	5
2.1	CPU Cores and Threads	8
2.2	hydra Cluster	8
2.3	A hydra compute node	9
2.4	Optimal Speed-Up and Parallel Efficiency	11
2.5	Compact Example	13
2.6	Scatter Example	13
2.7	True/False Positive/Negative	15
2.8	Metrics for Binary Classification Problems	15
3.1	Gantt Chart - Case Seq 1	31
3.2	Gantt Chart - Case Seq 2	31
3.3	Gantt Chart - Case Concur	31
3.4	Rodinia - kernel time - gcc	33
3.5	Rodinia - user time - gcc	34
3.6	Rodinia - small - Arithmetic Intensity	35
3.7	Rodinia - kernel time - compiler comparison - seq 1	38
3.8	Rodinia - kernel time - compiler comparison - concur	38
3.9	SPEC OMP2012 - benchmark execution time - gcc	40
3.10	SPEC - Arithmetic Intensity	40
3.11	SPEC OMP2012 - execution time - compiler comparison - seq 1	43
3.12	SPEC OMP2012 - execution time - compiler comparison - concur	43
3.13	Rodinia - small - hydra - Parallel Scalability	44
3.14	Rodinia - small - hydra - Parallel Efficiency	45
3.15	Rodinia - medium - hydra - Parallel Scalability	46
3.16	Rodinia - medium - hydra - Parallel Efficiency	46
3.17	SPEC OMP2012 - test - hydra - Parallel Scalability	47
3.18	SPEC OMP2012 - test - hydra - Parallel Efficiency	48
3.19	SPEC OMP2012 - train - hydra - Parallel Scalability	48
3.20	SPEC OMP2012 - train - hydra - Parallel Efficiency	49
3.21	Scheduling Comparison (AIP) - Rodinia - small and medium - hydra	55

3.22	Scheduling Comparison - SPEC OMP2012 and Rodinia - hydra	57
4.1	Visualization Socket Tests - Different Sockets	60
4.2	Visualization Socket Tests - Compact Same Sockets	61
4.3	Visualization Socket Tests - Scatter Both Sockets	61
4.4	Synchronization of Two Programs	62
4.5	Overhead of sync for Rodinia Benchmarks	64
4.6	Relevance of Sync - backprop and myocyte - small - 1 socket	65
4.7	Relevance of Sync - backprop and myocyte - small - scatter	65
5.1	Correlation Matrix - likwid - hotspot/backprop	79
5.2	Correlation Matrix - PAPI Performance Events - hotspot/backprop	82
5.3	Correlation Matrix - PAPI Performance Events - hotspot/hotspot3d	83
5.4	Significance Levels of AOV	85
5.5	Relevance of AOV - Difference between Significance Levels	86
5.6	Running times of hotspot dedicated and co-scheduled - PAPI	86
6.1	Logistic Regression Evaluation of Random Test Set	95
6.2	Performance Event Distribution of backprop	97
6.3	Performance Event Distribution of hotspot	97
6.4	Performance Event Distribution of hotspot3d	98
6.5	Performance Event Distribution of myocyte	99
6.6	Logistic Regression Evaluation of backprop	99
6.7	Logistic Regression Evaluation of hotspot	100
6.8	Logistic Regression Evaluation of hotspot3d	101
6.9	Logistic Regression Evaluation of myocyte	101

List of Tables

2.1	System description of the hydra computing cluster.	10
2.2	Tools used for measuring hardware performance counters.	12
2.3	Rodinia - Input Size <i>small</i>	21
2.4	Rodinia - Input Size <i>medium</i>	22
3.1	Compiler overview.	32
3.2	Performance counters obtained with <code>likwid-perfctr</code>	34
3.3	Arithmetic Intensity Overview - Rodinia - small	36
3.4	Arithmetic Intensity Overview - Rodinia - medium	37
3.5	Arithmetic Intensity Overview - SPEC OMP2012 - test	41
3.6	Arithmetic Intensity Overview - SPEC OMP2012 - Train	42
3.7	Basic Integer Program (BIP) variables.	50
3.8	Advanced Integer Program (AIP) variables.	51
3.9	Schedule Comparison Makespan Overview	54
4.1	Performance Counter Difference - no sync vs. sync - hotspot/kmeans/small	66
4.2	Performance Counter Difference - no sync vs. sync - hotspot/lud/small .	67
4.3	Overview - Socket Notation	68
4.4	Influence of Co-Scheduling - Rodinia backprop - Sharing a Compute Node	69
4.5	Influence of Co-Scheduling - Rodinia backprop	71
4.6	Influence of Co-Scheduling - Rodinia hotspot	72
4.7	Influence of Co-Scheduling - Rodinia myocyte	74
5.1	Groups of likwid performance counters	78
5.2	PAPI Performance Groups	80
5.3	Experimental setup for finding relevant performance counters.	86
5.4	PAPI Performance Events Significance / hotspot-streamcluster-hotspot3d	87
5.5	PAPI Performance Events Significance / hotspot-leukocyte-nw	88
5.6	PAPI Performance Events Significance / hotspot-heartwall-backprop . . .	89
5.7	PAPI Performance Group for Prediction	90
6.1	Test and Train Data Set Evaluation	96



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Introduction

Computing systems are constantly evolving, they get better and faster, and this requires better and faster hardware. One main component of computing systems are CPUs that comprise several cores. Computing systems can contain several CPUs and thus the number of cores in a system is increasing steadily. This trend of more and more cores in a supercomputer is shown in Figure 1.1. We see a rapid evolution of computing hardware over the last few years.

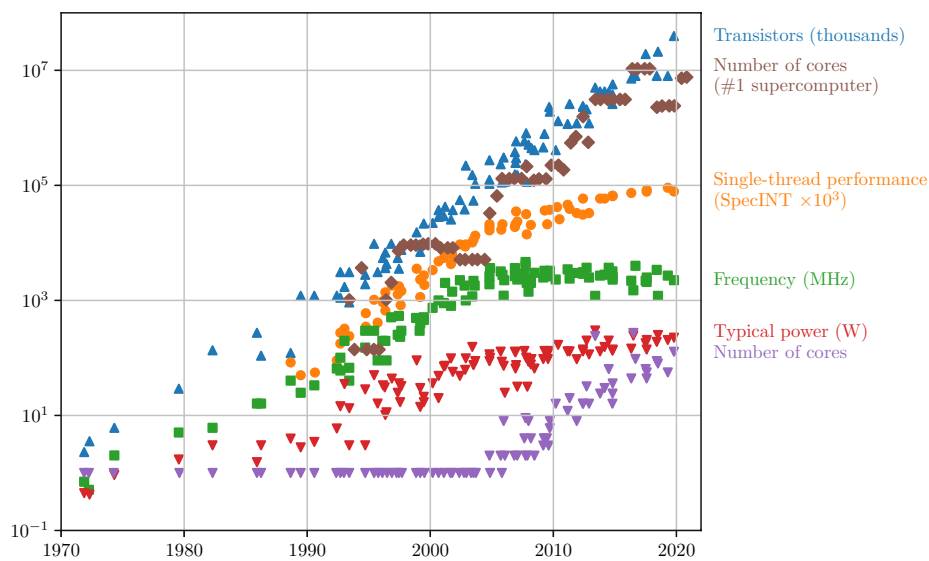


Figure 1.1: Supercomputer trend data [30, 42].

When we take a closer look at the number of cores shown in Figure 1.1, we see the increasing number of cores over the past few years: a processor was composed of only one

core until the early 2000's years. In 2010, the number of cores was 16, in 2019 already 128. There is a similar increasing trend for the number of cores comprised in the number one supercomputer of that time: the top supercomputer in 1993-1996 was *Numerical Wind Tunnel* with 140 cores [30], the fastest supercomputer in 2016-2018 was *Sunway TaihuLight* with 10.6 million cores across the computing system.

The most powerful supercomputer in Austria is located in Vienna, in the Vienna Scientific Cluster [7]: the VSC-4 [8]. The VSC-4 cluster has 790 compute nodes, where each node consists of two processors with 24 cores each. This means that one compute node provides 48 cores and the whole VSC-4 system 37 920 cores in total.

We see, supercomputers provide a tremendous number of cores. This requires scheduling of multithreaded applications to use the provided cores efficiently. Applications running on compute nodes often demand the maximum number of available cores for their execution with the goal of minimizing the execution time. Sometimes, using more resources does not even lead to smaller execution times, sometimes it only improves the running time minimally. To depict this phenomenon, we present an example in Figure 1.2.

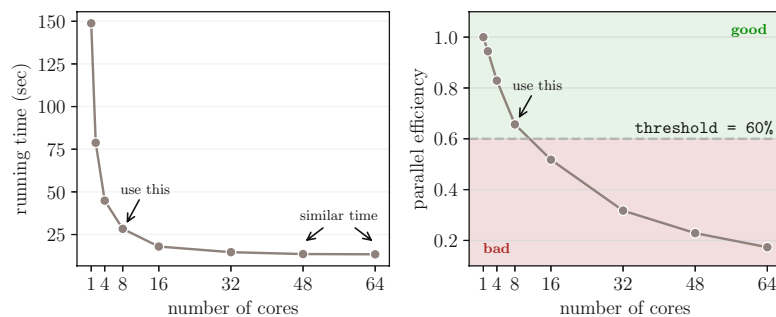


Figure 1.2: Relationship between the running time and parallel efficiency of `srad_v1`.

Figure 1.2 shows the runtime behavior of the `srad_v1` benchmark from the Rodinia benchmark suite on the 64 core machine `nebula`: `srad_v1` has a serial execution time of approximately 148 seconds, with 48 cores, we achieve a running time of about 13.9 seconds, using 64 cores, the running time is around 13.2 seconds. As we notice, there is only a minimal time difference between using 48 and 64 cores. Thinking about the energy needed to power 48 and 64 cores, we could use less resources, like the 48 cores, and still achieve a good performance result. Taking this thought to greater extent, having thousands of applications using all available resources causes the needed power for maintenance to skyrocket [38]. Therefore, energy efficiency is often considered as an additional goal supplementing runtime minimization for scheduling problems [10, 25, 38]. This signifies the importance of finding a good balance between the minimal runtime and more down-to-earth energy and power requirements.

Further on, we see the relation between the running time and parallel efficiency of `srad_v1` in Figure 1.2. The parallel efficiency ($speedup/cores$) is an important measure,

and we often define an *acceptable parallel efficiency*. This acceptable parallel efficiency can be described as the threshold value used to differentiate between an acceptable and an unacceptable parallel efficiency: parallel efficiencies above the threshold are acceptable, values below this threshold are unacceptable. In Figure 1.2, this acceptable parallel efficiency is 60%, and only parallel efficiencies over this threshold value are considered acceptable. We see that in this example the parallel efficiency is only acceptable for 2, 4, and 8 cores, but for more cores the parallel efficiency is too low to be acceptable. Using this information, we can decide on using 8 cores for `srad_v1` on `nebula`, and achieve a small running time with an acceptable parallel efficiency.

Consequently, applications often shall not be executed with all available cores, but run in parallel with less cores. If we execute an application with less cores, there are free cores and the question arises, what to do with these free cores. To make sure that the computing system is utilized efficiently, we can co-schedule different applications on one compute node to use available resources reasonably. But then, questions arise of how many cores should be used and what applications can be co-scheduled with each other, without significant performance losses.

If we look at supercomputers, like the VSC-4, there are certain parallel applications that are typically running on such supercomputers. LAMMPS [5] and GROMACS [3] are exemplary software modules running on supercomputers [4, 6]. LAMMPS, a Large-scale Atomic/Molecular Massively Parallel Simulator, “is a classical molecular dynamics code with a focus on materials modeling” [5], and GROMACS “is a versatile package to perform molecular dynamics” [3]. For us, it is not important to know what these software modules calculate, the important fact is that they can be executed in parallel mode and are commonly pre-installed and subsequently commonly used on supercomputers.

The goal of this thesis is to assess the co-scheduling potential of two applications on multicore machines. To assess this potential, we need to evaluate and study several other aspects too, like the scalability and scheduling potential of applications, and the characterization of performance counters, to finally predict a co-scheduling potential. For assessing the co-scheduling potential of two applications, we assume that one application is fixed beforehand, e.g., to such a commonly used software module as LAMMPS or GROMACS. A program then requests to be executed on the same computing system. Can we co-schedule this application with our fixed program or not? We show that it is possible to predict whether to co-schedule two applications or not. This knowledge can be used by supercomputers to train models for commonly used programs, like LAMMPS or GROMACS, and predict the co-scheduling potential with any other application.

In this thesis, we do not use large-scale distributed applications, like LAMMPS or GROMACS. We experiment on multicore machines with OpenMP benchmarks to cover a variety of application characteristics. We research different ways of resource sharing, i.e., sharing the cores of a compute node by several applications, and later predict the co-scheduled performance of four Rodinia benchmarks using performance counters and logistic regression. The contributions of this thesis are the following:

- We examine the technical side of Rodinia [19, 20] and SPEC OMP2012 [35] benchmarks by doing different runtime experiments.
- We schedule applications using elaborate methods to significantly improve the makespan, i.e., the overall execution time. Therefore, we provide an integer program prototype for calculating how many and which cores should be used for an optimal schedule.
- We explore different ways of sharing resources provided by a compute node, i.e., different strategies to share cores on a compute node.
- We examine the behavior of co-scheduled kernel sections, and show the necessity of a synchronization mechanism around kernel sections. Thus, we introduce a small C library to synchronize compute kernels for precisely measuring hardware performance counters.
- We analyze hardware and performance counters using `likwid` [27] and `PAPI` [45] to find appropriate counters for the prediction of co-scheduling performances and behaviors. We notice measurement inconsistencies and trace it back to potential problems of `likwid`'s Marker API.
- We create prediction models for four Rodinia benchmarks to show the possibility of predicting co-scheduling potentials of applications using logistic regression with performance counter values. The goal is to use a small execution for the prediction itself, but predict the co-scheduling potential of the applications' full length execution.

The structure of this thesis is depicted in Figure 1.3. Chapter 2 contains background information on prerequisites needed in this work, e.g., information about multicore machines, parallel performance measurements, classification problems, etc. We explore the used Rodinia and SPEC OMP2012 benchmarks and discuss related work. In Chapter 3, we analyze execution behaviors of the used benchmarks, such as the single core and scalability performance. Additionally, we show the potential of scheduling by comparing basic scheduling techniques, like a fully parallel schedule, with schedules created by an integer program. Then, we define possible resource sharing granularities, i.e., how applications can share a compute node, in Chapter 4. We notice fluctuating kernel times when measuring kernel times of co-scheduled programs, and subsequently introduce a library for synchronizing various program parts, like kernel sections. Using this `sync` library for synchronizing kernel sections of co-scheduled programs, we explore what resource sharing strategies might lead to conflicts shown by increased execution times, i.e., slowdowns of applications. With this knowledge, we devise a specific strategy, our *scatter* resource sharing, to measure performance counters in Chapter 5. We demonstrate problems occurring with measuring performance counters with `likwid`'s Marker API and show that kernel times do not correlate with any of the available hardware counter values, even for same measurements. Therefore, we switch to `PAPI` hardware counters, also called performance events, and determine relevant events needed for our prediction. In Chapter 6, we then use some of the `PAPI` performance events to create a prediction model using logistic regression. This model predicts whether two applications should

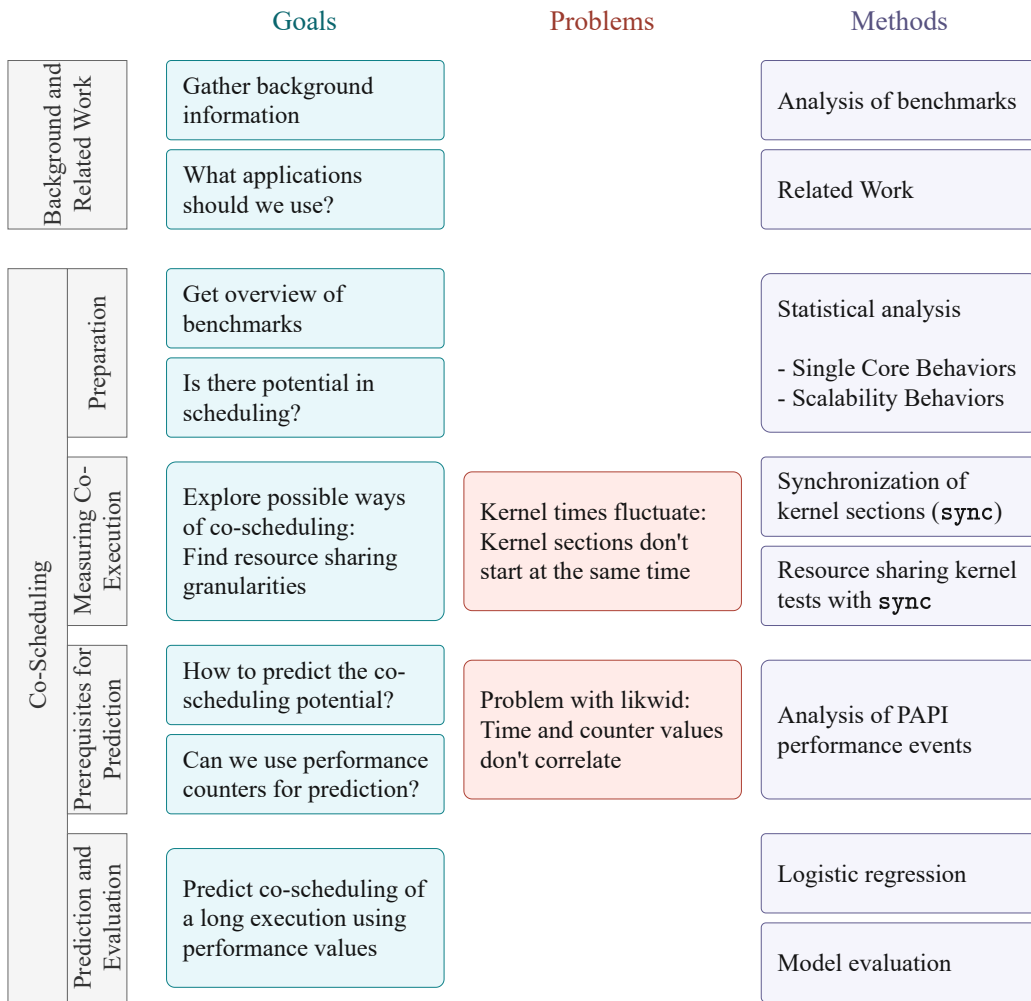


Figure 1.3: Overview of the thesis structure.

be co-scheduled or not. For the prediction, we use measured counter values of a short execution and predict the co-scheduling potential of this program's long execution. We observe that the performance values for the short and long execution have to correlate to create a sufficiently good prediction model.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Background and Related Work

In this chapter, we take a look at multicore machines and shared memory systems. We execute OpenMP applications on such machines and measure the performance of these parallel programs, e.g., the speedup or the parallel efficiency. Further, we introduce scheduling strategies and the concept of regression analysis. Finally, we explore work done in this area and summarize related topics.

2.1 Background Information

2.1.1 Multicore Architecture on Shared Memory System

For executing programs in parallel, we need corresponding computing resources that provide parallelism. As for all computers, the main computing resource is the CPU (Central Processing Unit) or the processor, which resides in a socket. Nowadays, a processor contains several cores. Figure 2.1 shows a CPU with four cores, i.e., a quad-core CPU. Each core on the CPU is an individual physical component [44] and handles instructions on its own. Therefore, several cores lead to parallelism since each core can compute its own batch of instructions independently. Regarding parallelism, it is important to look at the number of threads per socket. As shown in Figure 2.1, it is possible that one core has two threads. This means that there is not only one path, but two paths for each core. Even though two independent computations can be executed concurrently on one core with two threads, the applications still share the one computing resource, the CPU, which alternates between the applications. Nonetheless, the number of available threads is the duplicate amount of cores if there are two threads per core. OpenMP applications use available threads, where the maximum number of threads corresponds to the number of cores \times threads per socket.

Even though there are several cores on one CPU, there is always an interest to increase parallelism by using more cores. A CPU has a certain number of cores, but the collabo-

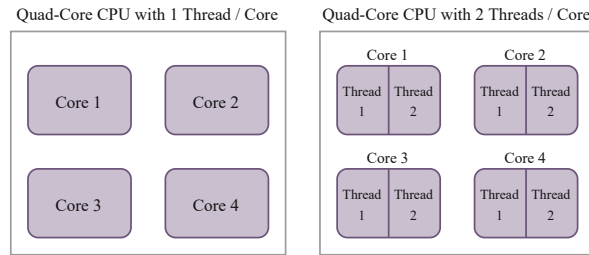


Figure 2.1: CPU cores and threads.¹

ration of several sockets increases the total number of cores on a system. If a server has more than one socket, and therefore has more than one processor, we are talking about multi-processor servers that support multi-sockets. Intel Xeon Scalable Processors [2] support two-socket, four-socket, and eight-socket configurations. The CPUs used for such systems are specifically designed to work together in these multi-socket configurations. Therefore, these systems lead to an increased parallelism since the number of cores on this one computing system is the sum of cores on all participating processors.

We see that cores reside on separate processors and therefore on separate sockets, but still create a unified computing system. Therefore, this single machine requires memory shared by multiple processors. Accordingly, these systems are also called *Shared Memory Parallel Computers* (SMPs) [18]. If each processor accesses any memory location with the same speed, “they have *uniform memory access* (UMA) time” [18]. In comparison, *non-uniform memory access* (NUMA) [?] is an architecture, where each processor has its local memory that can be accessed fast by the processor itself, but still shares its memory space with the other processors that access this local memory slower. Therefore, the memory access time is not uniform for each processor and thus called NUMA.

As an example of a multicore machine, we take a look at the hydra computing cluster, which we use in this thesis for experiments. The hydra cluster system, shown in Figure 2.2, contains a frontend node called hydra and 36 compute nodes named hydra01 to hydra36. These 36 compute nodes can be used as individual or connected computing resources, depending on the use case. We use one compute node individually for the execution of OpenMP applications.

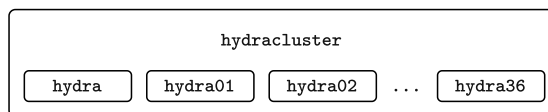
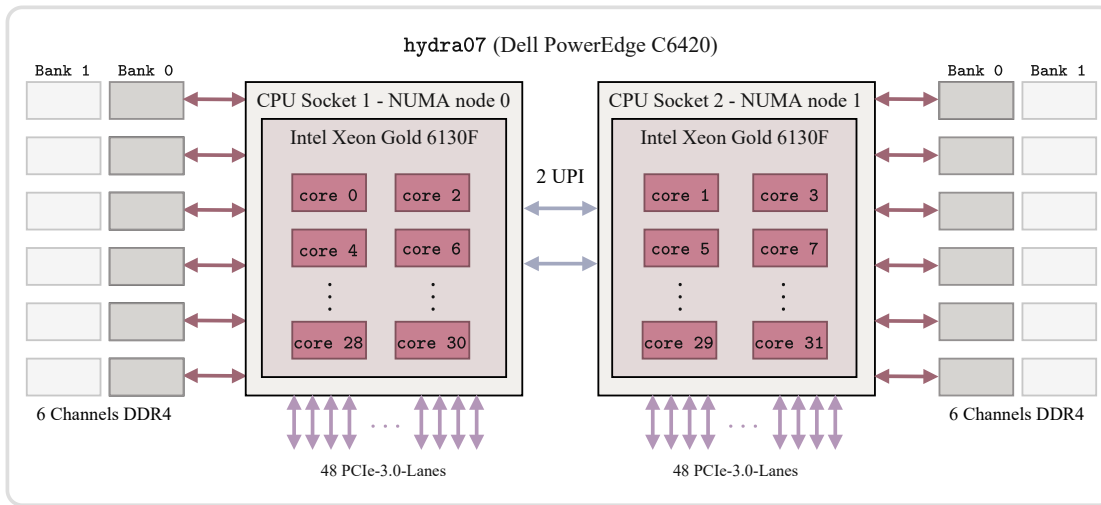


Figure 2.2: Scheme of the hydra cluster.

All 36 compute nodes on hydra are constructed equally. Figure 2.3 shows one of these compute nodes, e.g., hydra07. Each compute node is a Dell PowerEdge C6420 machine that supports a two-socket configuration.

¹<https://www.temok.com/blog/cores-vs-threads> (Accessed on 2021-08-15)

Figure 2.3: Composition of a hydra compute node.^{2,3}

As we see in Figure 2.3, there are two CPU sockets and in each of them an Intel Xeon Gold 6130F processor resides. This processor contains 16 cores with two threads per core, but threading is disabled on the hydra nodes. Therefore, each processor on hydra07 contains 16 cores with one thread per core, and thus the whole hydra07 compute node provides 32 cores. Each socket also corresponds to one NUMA node. We notice this *non-uniform memory access* through the six connections of each socket to six channels of DDR4 Dual Inline Memory Modules (DIMMs), which is the local memory of each processor. For communication between the sockets, there are two UPI (Intel Ultra Path Interconnect) channels that are responsible for the point-to-point interconnect between processors.

Listing 2.1: Physical core assignments on a hydra compute node.

NUMA node0 CPU(s) :	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
NUMA node1 CPU(s) :	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31

The physical core mapping on any Linux machine can be shown with `lscpu`. On any hydra compute node, the physical core assignment corresponds to the mapping shown in Listing 2.1.

Experimental Setup

We conduct our experiments on single compute nodes of the hydra cluster. A summary of relevant system information for our parallel computing experiments is given in Table 2.1.

²<https://www.aspsys.com/solutions/hpc-processors/intel-xeon-skylake> (Accessed on 2021-08-15)

³<https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html> (Accessed on 2021-08-15)

Table 2.1: System description of the hydra computing cluster.

hydra cluster	
Compute Cluster	36 compute nodes
hydra compute node	
Processor	Intel(R) Xeon(R) Gold 6130F CPU @ 2.10GHz 32 cores on 2 sockets with 2 NUMA domains
Memory	97.45 GB

2.1.2 What is OpenMP?

“OpenMP is a shared-memory application programming interface (API) [...] to facilitate shared-memory parallel programming [18].” OpenMP is not a programming language, but an extension to the used programming language. We can add specific pragmas to parallelize parts of the program. OpenMP can be used for C, C++, and Fortran code.

We use applications annotated and therefore parallelized with OpenMP for our experiments on the `hydra` compute nodes. This means that one application can be executed in parallel using up to 32 cores.

2.1.3 Measuring Parallel Computing Performance

For examining the parallel computing performance, the main metric is the elapsed time for the execution of an application. From the time measurements, we can directly derive parallel performance metrics, like the speed-up or the parallel efficiency. But there are several possibilities to measure the time: we can measure the elapsed time observed as a user, the *user time*, i.e., the wall clock time, or the *kernel time*, the computational kernel time measured as the elapsed time of kernel sections only. To clarify these two time measurements, we take a look at an example C code using OpenMP from Listing 2.2.

Listing 2.2: Example of a dot product implementation using C [18].

```

1 int main(int argc, char *argv[]) {
2     double sum = 0;
3     double a[256], b[256];
4     int status, i;
5     int n = 256;
6
7     for (i = 0; i < n; i++) {
8         a[i] = i * 0.5; b[i] = i * 2.0;
9     }
10
11     #pragma omp for reduction(+:sum)
12     for (i = 1; i <= n; i++) { sum = sum + a[i] * b[i]; }
13
14     printf("sum = %f \n", sum);
15 }

```


This program from Listing 2.2 calculates the dot product of two arrays a and b . If we measure the *user time*, then we measure the time needed for the whole program code, from line 1 to line 15. In comparison, the *kernel time* only occurs in the pragma and therefore only includes the time needed for the code lines 11 and 12. For the computational *kernel time*, we exclude *preprocessing steps*, like variable declarations and initializations, and *postprocessing steps*, like printing results, from the *kernel time* measurement. Even though the *user time* and *kernel time* of this program might not differ significantly, preprocessing steps may include reading input data from input files, while postprocessing steps may include writing a big amount of data to result files. We will see that it is application dependent, how much time the pre- and postprocessing steps take proportionally to the wall clock time. For parallel computing metrics it is obvious to use the *kernel time*, since we are only interested in kernel level processes, not user level processes.

Two relevant metrics for measuring the performance of a parallel algorithm are the *speed-up* and the *parallel efficiency* [31]. The formula for the *speed-up* is $\frac{T_1}{T_p}$, where T_1 is the elapsed time for the sequential execution of a program using one core, and T_p is the elapsed time for the parallel execution with p processors. The *parallel efficiency* is derived from the *speed-up* by calculating $\frac{\text{speed-up}}{p}$, where p stands for the number of used processors. If we think of a perfectly parallelizable program, the elapsed time using two processors takes half the elapsed time as opposed to using one processor. This means that the speed-up will be as high as the number of processors, in this case two. Consequently, the parallel efficiency is the speed-up divided by the number of processors, which then is one.

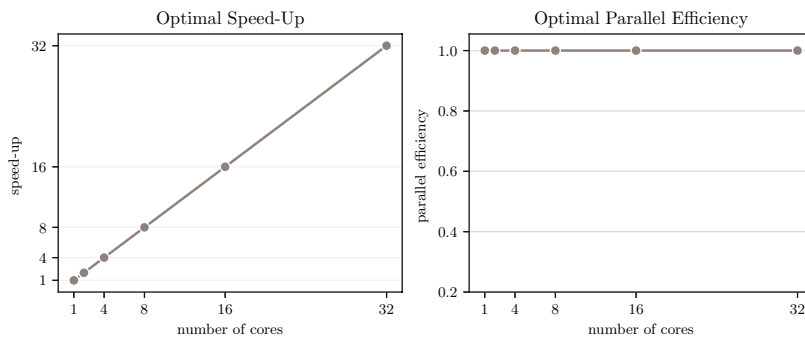


Figure 2.4: The optimal (linear) speed-up and the optimal parallel efficiency.

Figure 2.4 shows the optimal speed-up and the optimal parallel efficiency of a perfectly parallelizable program. The linear speed-up equals the number of processors used, while the parallel efficiency is always one. Later, we will see that this optimal behavior, and even approaching optimal behaviors, is rare.

Additionally to these basic measurements calculated from the elapsed time, CPUs provide hardware counters that can count various performance metrics, which can be retrieved

with special tools like `likwid` [27] or `PAPI` [45]. Such performance counters are, e.g., the number of floating point operations, the number of cache misses, the memory bandwidth, and many more. From these counters, we can derive another important parallel performance metric, the *arithmetic intensity*. The arithmetic intensity is the ratio of the floating point operations to the memory bandwidth [37], i.e., the work-to-memory-traffic ratio. We used the listed `likwid` and `PAPI` versions in Table 2.2 for our experiments.

Table 2.2: Tools used for measuring hardware performance counters.

Tool	Version
<code>likwid</code>	5.1.1
<code>PAPI</code>	6.0.0.1

2.1.4 Scheduling and How to Solve Co-Scheduling Problems

“Scheduling is a decision-making process” [40] and is used to allocate resources to tasks over time periods. Scheduling always has a goal, namely, an optimization goal. In our context, we use scheduling to make a decision of which jobs are executed on which machines [17]. There are n jobs that should be allocated to m machines. Since we only use one machine and want to schedule on cores instead of machines, we define scheduling as finding job allocations to cores for some specific time intervals. The optimization goal of scheduling is to minimize the total cost function [17], but this total cost function can be defined in numerous ways. A very common optimization goal for scheduling jobs to machines is minimizing the makespan C_{max} , the “completion time of the last job to leave the system [40].”

Since scheduling is an optimization problem, the question arises of how to solve it. As for optimization problems in general, we could use some heuristic or reduce the scheduling problem to other optimization problems. One example is using linear or integer programs [17, 22, 40] to decide job-machine allocations. A linear program maximizes or minimizes an objective function representing the goal of our scheduling problem. Additionally, constraints define the scheduling behavior and therefore limit the possible result space. For a linear program, “the objective and the constraints are linear in the decision variables [40].” An integer program is a specific type of linear program, where the decision variables are integer numbers. The problem of allocating jobs to machines or cores can restrict the decision variable type even more, if we answer the allocation with yes or no, i.e., if the decision variable is binary. Even though linear programs can be solved in polynomial time [22], integer programs are NP-hard. Therefore, often scheduling algorithms are created that are heuristics trying to find a good resource allocation. We will use integer programs as a proof of concept to show that scheduling leads to runtime advantages compared to basic resource allocations.

2.1.5 Affinity Mappings

With scheduling, we commonly do not use all cores of one machine for one application, but only a few. For such use cases, there is the possibility to map the threads of applications to physical processing units. This is called *thread affinity*, where we map threads to *places* [36], such that the operating system cannot reassign them to other places. There are two commonly used thread affinity mappings: *compact* and *scatter*.

For illustration purposes we assume a four-socket system with four cores on each socket. Using this system configuration, we depict the *compact* and *scatter* thread mapping for the case of mapping seven threads in Figures 2.5 and 2.6.

A *compact* affinity mapping assigns the $(n + 1)^{th}$ thread to a free thread context as close as possible to the thread context of the n^{th} thread [1]. We see this in Figure 2.5, where we start assigning threads to socket 1, until it is full, and then fill the next socket, socket 2. Then, our seven threads are mapped to as close as possible thread contexts.

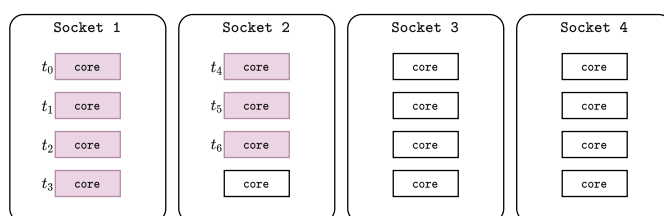


Figure 2.5: Example of a *compact* affinity mapping of seven threads on a four-socket system with four cores on each socket.

In comparison, the *scatter* mapping “distributes the threads as evenly as possible across the entire system” [1], which represents the contrary to a *compact* mapping. Figure 2.6 shows that the thread mapping assigns one thread to each socket, then does another round of assigning one thread to each socket. With this method, we distribute the threads across our four-socket system.

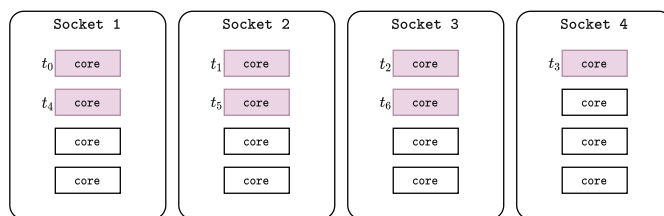


Figure 2.6: Example of a *scatter* affinity mapping of seven threads on a four-socket system with four cores on each socket.

2.1.6 Regression Analysis

We are interested in predicting values based on given input variables. Machine learning models are helpful, and mathematical models can be used for a basic data analysis. Regression analysis is such a technique, which is used to predict the *dependent variable* based on the given *independent variables* [21, 29]. A regression analysis tries to find a connection in the provided data by building a mathematical model.

There are several types of regression, one well known regression is a *linear regression*, which assumes a linear relationship between the independent variables to predict the dependent variable, i.e., it builds a linear function for prediction. Another regression analysis form is *logistic regression*, which predicts a binary outcome from the independent variables. A logistic regression also builds a function, but decides using the *log odd* [29] the binary outcome. This means that logistic regression predicts a likelihood of the binary outcome and uses this to decide, which value the dependent variable should have.

Metrics for Binary Classification Problems

Since we are interested in predicting binary outcomes, we need to validate the behavior of our binary classification problem, e.g., the logistic regression. Since the binary outcome only takes two values, there are four possibilities how the actual and expected outcome are related [24]:

- *True Positive*: The output is correctly labeled as positive, i.e., both the actual and expected outcome are positive.
- *False Positive*: The output is incorrectly labeled as positive, i.e., the actual outcome is positive, but the expected outcome is negative.
- *True Negative*: The output is correctly labeled as negative, i.e., both the actual and expected outcome are negative.
- *False Negative*: The output is incorrectly labeled as negative, i.e., the actual outcome is negative, but the expected outcome is positive.

These four categories are illustrated in Figure 2.7, where we see how these names are derived from the selected and actually relevant items.

For evaluation purposes, we are often interested in the number of true positives, false positives, true negatives, and false negatives. A *confusion matrix* displays these four categories [24] in a tabular layout.

From these values we can derive actual evaluation metrics, such as the precision or recall of the binary classification problem. Figure 2.8 shows how these metrics are calculated:

- The *recall* is the ratio of true positives to all relevant items.
- The *precision* is the ratio of true positives to all selected items.

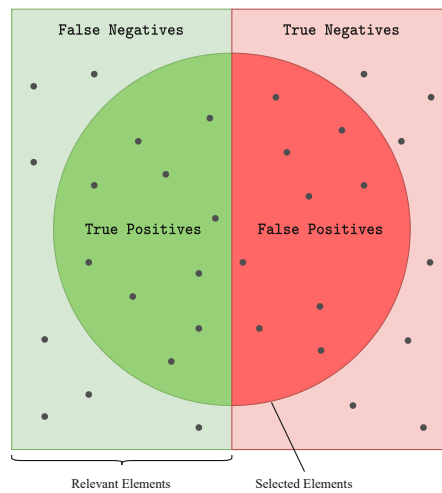


Figure 2.7: Illustration of true positives, false positives, true negatives, and false negatives.⁴

- The *true positive rate* describes the same ratio as the *recall*, i.e., the ratio of true positives to all relevant items.
- The *false positive rate* is the opposite of the *true positive rate*, i.e., the ratio of false positives to all non-relevant items.

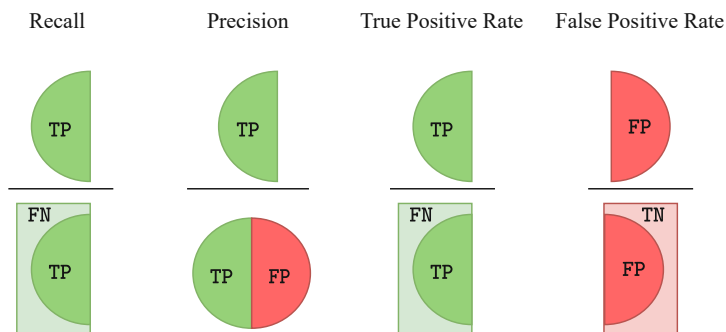


Figure 2.8: Metrics used to evaluate results of binary classification problems.

These evaluation metrics are often plotted in a *Receiver Operator Characteristics* (ROC) curve or *Precision-Recall* (PR) curve. A ROC curve plots the false positive rate on the x-axis, and the true positive rate on the y-axis. In comparison, the PR curve plots the recall on the x-axis, and the precision on the y-axis. The ROC curve is not optimal for presenting highly skewed data because it presents the results too optimistically. Therefore,

⁴<https://medium.com/analytics-vidhya/evaluating-ml-models-precision-recall-f1-and-accuracy-f734e9fcc0d3> (Accessed on 2021-10-13)

a PR curve is additionally used since these curves present a good alternative to the ROC curves [24] for skewed data.

2.2 Overview of Used Benchmarks

This section presents an overview of the Rodinia and SPEC OMP2012 benchmarks. The Rodinia benchmark suite does not provide inputs, therefore we analyze the benchmarks in more detail to find appropriate input sizes. For the SPEC OMP2012 suite, we take a closer look at the defined input instances.

2.2.1 Rodinia OpenMP Applications

The Rodinia benchmark suite [19, 20] contains 23 applications for CUDA, OpenMP, and OpenCL parallel models. Since we are targeting multicore systems, we consider the OpenMP applications, and 19 out of the 23 applications are available for OpenMP. There are no predefined input sizes for the Rodinia benchmarks, thus we need to specify them. To do so, it is necessary to get a knowledge of how to execute the applications.

Applications

We want to compare applications with one another. Thus, we will later choose such input sizes that the applications have a similar serial running time, a similar elapsed user time without parallelization. Some benchmarks will be excluded from our runtime experiments since we cannot find a suitable execution that fits the input sizes.

b+tree

A **b+tree** is a search data structure and the benchmark traverses the **b+tree** to get higher speedups. The usage of the application is `./b+tree.out cores <cores> file <file> command <command>`. The input file `<file>` is enumerating a range up to a given number. The command file contains commands, like the number of bundled queries to run on the CPU and GPU.

The command file used in these scalability experiments uses the same commands as the exemplary command file given by the benchmark suite: `j x y` and `k z`. These commands can be described as running a range search of `x` bundled queries on the CPU and GPU with the range of each search of size `y`, and running `z` bundled queries on the CPU and GPU.

backprop

The Back Propagation application **backprop** “is a machine learning algorithm that trains the weights of connecting nodes on a layered neural network [19].” The **backprop** application is executed as `./backprop <number of input elements>`. The parameter for the number of input elements determines the layer size of the neural network.

The number of cores needs to be changed in the header file `backprop.h` since there is no parameter given to set the number of cores directly in the application call.

bfs

The `bfs` application is a breadth first search using a graph, which is given as a parameter in an input file. The exact usage is `./bfs <cores> <input file>`. The graph input file can be generated with a provided generation script. Even though the parameter `<cores>` should set the number of cores used, this does not happen, but we always use an explicit affinity mapping. The user elapsed time of the serial code part in general is quite high, because there are only small improvements in time comparing using one or all available cores.

For the sake of similar input sizes, we need to define a very large graph with several million nodes for `bfs`. Since this graph is given in the input file, the preprocessing of reading in this graph file is high and very significant for `bfs`.

cfd

The `cfd` application is a solver for three-dimensional Euler equations used for compressible flow. There are four versions of this benchmark: a redundant flux computation and pre-computed fluxes, and both versions are also available with double precision computations. We use the redundant flux computation with the usage `./euler3d_cpu <input file>`. The number of used cores needs to be changed in the Makefile according to the documentation, but in our experiments we used explicit affinity mapping since more cores were used than we set in the Makefile. There are three input files given with three-dimensional Euler equations, and the smallest serial running time can be achieved by `fvcorr.donn.097K` with more than 100 seconds. Since this running time is quite high and does not fit into our input sizes, this benchmark will not be used in any scalability experiments.

heartwall

The `heartwall` application tracks the movement of a mouse heart over ultrasound images. The usage of this application is `./heartwall <input file> <number of frames> <number of threads>`. The input file is the file containing the mouse heart movement ultrasound images and is provided in the benchmark suite. The number of frames parameter is the parameter we used to adjust the application to different input sizes.

hotspot

The `hotspot` benchmark application creates power and thermal models [47] and estimates processor temperatures based on simulated power measurements and further characteristics. The usage of this application is `./hotspot <grid rows> <grid cols> <sim time> <threads> <temp file> <power file> <output file>` and the

number of grid rows and columns is related to the actual data given in the initial temperature and dissipated power input files. For generating additional temperature and power files to the given ones, a script is provided.

hotspot3d

The `hotspot3d` benchmark is like the `hotspot` application, a physical simulation in a structured grid, but in three dimensions. The application is executed by the command `./3D <rows / cols> <layers> <iterations> <power file> <temp file> <output file>`. As we can see, the used number of threads / cores can not be determined in the execution command, and the application uses all available or mapped threads. The given parameters `rows` (and implicitly `columns`) and `layers` match the given power file and temperature file, since both these files contain a grid with a specific number of rows and columns and an explicit amount of layers.

kmeans

The `kmeans` application is a clustering algorithm used in data mining by dividing a cluster of data objects into `k` sub clusters. Two implementations are provided, a serial variant `kmeans_serial` and `kmeans_openmp`. For all experiments, we use the parallelizable OpenMP applications and execute them with a single thread to simulate serial executions. We execute this OpenMP benchmark with the command `./kmeans_openmp/kmeans -n <cores> -i <input file>`, where the input file represents a file containing data to be clustered. Input files can be created with a provided input generation script.

lavaMD

The `lavaMD` application “calculates particle potential and relocation due to mutual forces between particles within a large 3D space [19, 20]”. The code can be executed with `./lavaMD -cores <cores> -boxes1d <num>`, where the second parameter `boxes1d` describes the number of boxes in one dimension.

leukocyte

Detecting and tracking rolling leukocytes, white blood cells, is the intention of the `leukocyte` benchmark. First, cells are detected in a video frame, which are then tracked through subsequent frames. The application’s usage is `./leukocyte <number of frames> <number of threads> <input file>`, where the input file is provided.

lud

The LU decomposition application `lud` describes an algorithm to calculate solutions of a set of linear equations. There are two ways to execute `lud`: by inputting either the matrix size or using an input file. In our experiments, we use the option of giving a number as the matrix size and execute `./lud_omp -n <cores> -s <matrix size>`.

mummergpu

The MUMmerGPU application `mummergpu` is a pairwise local sequence alignment program, which uses the GPU for query sequence alignments. This application requires Nvidia G80 or later graphics cards. An emulator is provided that does not run on the graphics card, but on the CPU. For compiling the benchmark, the Nvidia CUDA compiler is used. Since our benchmarking system is `hydra`, a machine with an Intel CPU that does not have Nvidia components, we exclude this benchmark from our experiments.

myocyte

The `myocyte` benchmark models and simulates the heart muscle cell's, the cardiac myocyte's, behavior. This benchmark is executed as `./myocyte.out <simulation time in ms> <instances> <parallelization method> <threads>`. The parallelization method is either 0 or 1, where the 0 stands for parallelization inside each simulation instance and 1 marks the parallelization across instances. We use 1, parallelization across instances, as the parallelization method.

nn

The nearest neighbor algorithm `nn` finds the k nearest neighbors from a data set. The application is executed as `./nn <file-list> <number of nearest neighbors> <target latitude> <target longitude>`, where the `file-list` file contains file names to the records to search the given number of nearest neighbors. Additionally, the target latitude and longitude determine the coordinate for the distance calculations. For the file-list generation, a program `hurricane_gen` is provided.

nw

The Needleman-Wunsch application is a method for DNA sequence alignment, where potential sequences are saved in a two-dimensional matrix. The application is executed as `./needle <max rows/max cols> <penalty> <threads>`, where the maximum number of rows and columns determines the size of the two-dimensional matrix.

particlefilter

The `particlefilter` application estimates the location of a target object by guessing frames and calculating the probability or likelihood of the guesses. The application's usage is `./particle_filter -x <x dimension> -y <y dimension> -z <frames> -np <particles>`, where the x and y dimension define the frame size.

pathfinder

The `pathfinder` application finds a path on a two-dimensional space from the bottom row to the top row by either going straight up or diagonally. It is executed as

`./pathfinder <width> <number of steps>`, where the `width` parameter determines the number of columns and the `number of steps` parameter determines the number of rows in the two-dimensional grid.

srad

The `srad` benchmark, Speckle Reducing Anisotropic Diffusion, is one of the first stages of the `heartwall` application, where mouse heart movements are tracked. Two different `srad` versions are given, `srad_v1` and `srad_v2`. Their usage differs, as `srad_v1` is executed as `./srad <iterations> <saturation coefficient> <rows> <columns> <threads>`, where the number of rows and columns defines the rows and columns in the input image. The other version `srad_v2` can be executed as `./srad <rows> <columns> <y1> <y2> <x1> <x2> <threads> <lambda> <iterations>`, where the rows and columns refer to the domain and the `y` and `x` coordinates determine the position of the speckle. In our experiments, we use `srad_v1`.

streamcluster

The `streamcluster` application is a program originating from the `streamcluster` application of the PARSEC [11] benchmark suite. “For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center [12].” It is executed as `./sc_omp <min centers> <max centers> <dim of data points> <data points> <chunk size> <cluster size> <input file> <output file> <threads>`.

Defining Input Instances

The Rodinia benchmark suite does not provide predefined input instances. Therefore, we define a *small* and *medium* input size for each application. These input sizes are chosen such that the *small* input size has a serial running time of approximately ten seconds, and the *medium* input size a serial running time of 50 seconds using `gcc` as the compiler.

Table 2.3 shows the definition of the *small* input size for Rodinia benchmarks, and the *medium* input size is defined in Table 2.4.

2.2.2 SPEC OMP2012 Applications

SPEC OMP2012 [35] is a benchmark suite by the Standard Performance Evaluation Corporation (SPEC). It contains 14 relevant parallel applications using OpenMP. These 14 applications are written in C, C++, or Fortran. The benchmark suite contains predefined input sizes by SPEC, therefore we will not discuss the individual benchmarks. More information about the applications can be found in the SPEC OMP2012 documentation [35].

Before executing the benchmarks, the applications need to be built. The SPEC suite provides a defined build-process, and the applications can be built with the

Table 2.3: Execution commands of the Rodinia applications with a *small* input size.

Benchmark	Command
b+tree	./b+tree.out cores <p> file input.txt command command.txt input.txt: 2000000,command.txt: j 1000000 50000,k 65535
backprop	./backprop 12500000
bfs	./bfs <p> graph5M.txt
heartwall	./heartwall test.avi 6 <p>
hotspot	./hotspot 1024 1024 4000 <p> temp_1024 power_1024 out.out
hotspot3d	./3D 512 2 1250 power_512x2 temp_512x2 out.out
kmeans	./kmeans -n <p> -i 290000_34.txt
lavaMD	./lavaMD -cores <p> -boxesld 12
leukocyte	./leukocyte 3 <p> testfile.avi
lud	./lud_omp/-s 4608 -n <p>
myocyte	./myocyte.out 500 100 1 <p>
nn	./nn list669k_64.txt 3000 30 90 list669k_64.txt generated by ./hurricanegen 655360 64
nw	./needle 36864 10 <p>
particlefilter	./particle_filter -x 1024 -y 1024 -z 55 -np 10000
pathfinder	./pathfinder 1000000 80
srad	./srad_v1/srad 1350 0.5 502 458 <p>
streamcluster	./sc_omp 10 20 256 14500 14500 1000 none output.txt <p>

command runspec -config=spec_config.cfg -action=build -tune=base. The given config file `spec_config.cfg` provides the necessary information like compiler, compile flags, and further flags to make the programs. Then, each application can be run with `runspec -config=spec_config.cfg -size=<size> -noreportable -tune=base -iterations=1 -threads=<t> <benchmark>`. Since we execute each application separately, we use the `-noreportable` option instead of running reportable runs containing all benchmarks. The number of iterations is set to one since this `runspec` command also can be called `x` times instead of setting `-iterations=x` directly. The intention is to ensure similar invocations with regard to other applications, e.g., the Rodinia benchmark suite.

As already visible in the build command, we only use the base tuning method, which means that the compiler options are consistent across all programs of a given language. The execution command contains several important parameters, like the `size` parameter that specifies one of the given input sizes *test*, *train*, or *ref*, where their magnitude can be described as *test* < *train* < *ref*. The *test* dataset provides data for simple tests, whereas the *ref* dataset is the real data set, and *train* can be used for feedback-directed optimization. The number of iterations used in all experiments is strictly set to one, since we can simply run this command several times to simulate several iterations.

Table 2.4: Execution commands of the Rodinia applications with a *medium* input size.

Benchmark	Command
b+tree	./b+tree.out cores <p> file input.txt command command.txt input.txt: 6000000,command.txt: j 6000000 50000,k 65535
backprop	./backprop 44000000
bfs	./bfs <p> graph21M.txt
heartwall	./heartwall test.avi 29 <p>
hotspot	./hotspot 2048 2048 4800 <p> temp_2048 power_2048 out.out
hotspot3d	./3D 512 8 1600 power_512x8 temp_512x8 out.out
kmeans	./kmeans -n <p> -i 1111500_34.txt
lavaMD	./lavaMD -cores <p> -boxes1d 202
leukocyte	./leukocyte 17 <p> testfile.avi
lud	./lud_omp/-s 8192 -n <p>
myocyte	./myocyte.out 2550 100 1 <p>
nn	./nn list669k_64.txt 18000 30 90 list669k_64.txt generated by ./hurricanegen 655360 64
particlefilter	./particle_filter -x 2048 -y 2048 -z 85 -np 10000
pathfinder	./pathfinder 1000000 290
srad	./srad_v1/srad 6650 0.5 502 458 <p>
streamcluster	./sc_omp 10 20 256 50000 50000 1000 none output.txt <p>

Looking at the serial running time of the *test* dataset, we already recognize different magnitudes. All benchmarks except *md*, *bt331*, *ilbdc*, and *applu331* have a serial running time of approximately 10 seconds or less. As an example, the *md* application with the smallest input size *test* already leads to a serial running time of more than four minutes. Therefore, we do not use all provided SPEC OMP2012 benchmarks for all experiments, simply to prevent distortion of result interpretations from plots and other measurement results.

2.3 Related Work

Co-location of applications is a large research field and considered in several studies. There are numerous frameworks and techniques when it comes to co-scheduling. SCALO, a Scalability-Aware Parallelism Orchestration [26], is such a framework for multicore machines. It can be used for orchestrating concurrent application executions with the goal of increasing throughput. They argue that executing applications concurrently and therefore dividing the resources among applications is often more reasonable in terms of resource utilization than running one application on all resources. They use performance indicators to build a speed-up model to predict the speed-up of co-scheduled programs. In comparison to many other work, they explicitly assume existing synchronization

constructs available in the runtime API to measure only time spent in parallel code regions. In this thesis, we show the need for such a synchronization constructs and present a small C library.

Another framework is SCAF [23], a Scheduling and Allocation system with Feedback for shared memory systems. The goal is making good allocation decisions based on each application's observed efficiency. Since the parallel efficiency is calculated with the execution time of both the parallel and serial execution, they try to circumvent measuring potentially non-optimal serial running times. Therefore, SCAF measures the parallel efficiency by executing the serial thread only until completion of the parallel executed task.

Harris et al. [28] propose a resource management layer, Callisto, for parallel runtime systems. Their goal is to implement a resource management layer responsible for scheduling applications with each other efficiently for improved hardware utilization. They provide Callisto, a user-mode shared library, that controls the use of hardware contexts, i.e., threads. The scheduler used by Callisto defines which hardware contexts are assigned to which jobs, and further defines the behavior when hardware contexts become idle unexpectedly. This hardware context assignment to jobs is similar to our procedure, since we are interested in finding good resource allocations to jobs.

Job striping is another method for improving performance by co-locating jobs of different users on one compute node [15]. Job striping maps processes to cores and interleaves distinct jobs to reduce system contention on supercomputing clusters. This striping is defined as an additional NUMA affinity mapping, like compact and spread/scatter mappings are. We only use a compact and scatter mapping, whereas this striped configuration is a combination of two spread configurations and the performance of job striping can be predicted by looking at the runtime behavior of an application when using a spread affinity mapping.

Other works consider the problem of how to collocate or co-schedule specific workloads. Mercier et al. [33] collocate classic HPC workloads with Big Data workloads on HPC infrastructure. Their goal is to benefit from holes in schedules created from HPC workloads, and fill these holes with Big Data workloads. Since these two workload types differently exploit resources, they seem to complement each other. Therefore, they create BeBiDa, a solution to collocate HPC and Big Data workloads. HPC workloads are mainly executed on the available resources, but when HPC resources become idle, these resources get attached to the Big Data resource pool, which may assign these dynamic resources to Big Data workloads.

Breitbart et al. [14] present an HPC scheduler, Poncos, making decisions based on applications' main memory bandwidth requirements to improve the overall system utilization. Poncos is not intended to replace well-established HPC schedulers but shows potential of co-scheduling applications while monitoring the main memory bandwidth utilization.

Another type of workload co-scheduling is scheduling CPU and GPU workloads on multicore machines with GPUs. Since these two workload types differ, it is possible to exploit performance advantages that GPUs provide. Wang et al. [46] propose CAP, a

Co-Scheduling Strategy Based on Asymptotic Profiling, which can be used for such CPU and GPU workload schedulings.

Tangram [48] is a framework for predicting co-locating potentials of applications in an HPC cluster. They use prior knowledge about applications, e.g., if applications are memory or compute intensive. The co-location of applications itself uses oversubscription of resources, i.e., jobs are assigned to the same resource while overlapping each other. Oversubscription is said to better utilize resources and therefore should improve the overall performance of the system. Tangram uses resource usage characteristics for prediction, performance counters, such as the PAPI performance events. It predicts whether co-location of MPI applications would improve the makespan or not. As this description already implies, this is an online scheduling framework, i.e., the co-location performance predictions are used while already scheduling. This is one difference to our approach: we are interested in predicting beforehand, whether applications should be co-scheduled or not. Additionally, we do not use oversubscription.

There are also approaches that analyze and monitor the co-scheduling process. Pascual and Rzadca [39] analyze side-effects of co-locating tasks on a single machine and provide a model describing effects of co-located tasks on machines. Breitbart et al. [13] provide a tool called `autopin+` for monitoring and optimizing co-scheduling. They explicitly choose a memory-bandwidth-bound and a compute-bound application to be co-scheduled and analyze performance and energy efficiency of such co-schedulings. They use CPU performance counters to find optimal resource allocations, i.e., thread to core mappings to increase the efficiency of the underlying system. They show that the memory bandwidth presents a limit for performance, whereas compute-bound applications do not have restrictions and can efficiently use all cores.

Kumar Pusukuri et al. [32] develop a scheduling framework ADAPT with the idea of continuously monitoring the resource usage of multithreaded programs. Their goal is very similar to our thesis goal: co-schedule applications such that they interfere with each other as little as possible. They predict this interference with other programs using supervised learning techniques and then adaptively co-schedule programs. This work was written in 2013, and the state of research at this time seemed to be that cache usage behaviors could influence the co-scheduling decision. But this ADAPT framework not only uses cache usage behaviors, but lock contention and thread latency for making co-scheduling decisions. They further show that lock contention and thread latency have to be considered additionally to cache misses per accesses for an effective co-scheduling of multithreaded programs. In contrast, we explore different performance metrics without focusing on specific predefined categories.

The ADAPT framework consists of a *Cores Allocator* and a *Policy Allocator*. The *Cores Allocator* predicts a performance loss of a program when being co-scheduled with another program on one hand, and the performance of a program when the used core/processor configuration is changed on the other hand. This means that the models created by the *Cores Allocator* predict possible performance losses or improvements. The *Policy Allocator* monitors resource allocation decisions from the *Cores Allocator* and dynamically

selects appropriate memory allocations. We can say that this ADAPT framework is predictor and scheduler in one.

All of these scheduling approaches have a common goal in mind: to utilize the system as good as possible. For co-scheduling itself, the goal is often to minimize the makespan, but another optimization goal emerges: energy efficiency [10, 25, 26, 34, 38]. Mirka et al. [34] propose a technique to optimize energy-efficiency of compute systems running OpenMP workloads. Their technique automatically detects OpenMP workload execution patterns by profiling chunks of a loop iteration, a fundamental OpenMP concept, and computes performance measurements, as the throughput chunks per second, from the measured chunks. These measurements are then used for a dynamic control system of parallel OpenMP workloads to find good configurations regarding the performance and energy efficiency. Benedict et al. [10] present TOEP, the Threshold Oriented Energy Prediction, that aims at predicting energy consumptions and execution times of MPI/OpenMP applications. They use random forest models for training the prediction model. A survey of different power and energy predictive models is provided by O'Brien et al. [38], since power and energy efficiency are increasingly important metrics in HPC systems. As they argue, data centers use so much energy to power servers that the annual power cost to operate the system itself is enormous. This makes energy an additional optimization criterion for HPC systems. There is usually a trade-off between energy use and performance, shown in Pareto fronts. Therefore, Endrei et al. [25] try to find a good balance between energy usage and performance. They predict these Pareto fronts using B-splines trained in a neural network to help identifying trade-offs between performance and energy.

Even though the following research topic is not related directly with our thesis, the general idea and goal is very interesting. Ates et al. [9] introduce HPAS, an HPC Performance Anomaly Suite, an anomaly generator for HPC systems. They want to reproduce performance variabilities occurring in program executions on HPC systems and develop an anomaly suite to reproduce such variability causes, e.g., bandwidth interferences or memory leaks.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Co-Scheduling Preparation: Parallel Scalability and Scheduling Potential

In this chapter, we will examine how large the co-scheduling potential on current multicore systems is. To that end, we first examine the scalability behavior of OpenMP applications from two important benchmark suites. The following questions will be answered in this chapter:

- How can we characterize an application's runtime behavior?
Does the runtime behavior change in a concurrent execution compared to sequential executions?
Can we use performance metrics to characterize an application's runtime behavior?
- What is the parallel scalability of applications, i.e., how do they scale using more cores? If an application has a perfect parallel scalability, we can use all available cores and do not need co-scheduling.
How many cores should we use for co-scheduling?
- We want to co-schedule several parallel applications. Are basic strategies sufficient or are there advantages of using more elaborate co-scheduling strategies?

In the following, we examine the runtime behavior of our applications on our parallel machine hydra. We explore single core and serial runtime characteristics, and later the parallel scalability of the benchmarks. These experiments have many configuration possibilities, e.g., what compiler we use. To make sure that such configurations do not influence the overall outcome, we experiment with different compilers and show: even

though the running time itself may change, the overall runtime behavior of OpenMP applications is similar for different compilers.

This chapter is structured as follows: we start with single core execution experiments. We want to find differences between executing an application using a single core on different sockets, and whether an application interferes with itself in case of executing it concurrently. By testing different sockets, we verify that different configurations do not change the overall observation.

Since we observe that some benchmarks have a relatively large time increase for their concurrent execution compared to a sequential execution, we use performance metrics, the arithmetic intensity, to find out, which performance counter correlates with the running time.

As already mentioned in Chapter 1, using a large number of cores does not necessarily lead to the fastest running time, nor do we get an acceptable parallel efficiency. With this statement in mind, we analyze the actual parallel scalability behavior of the Rodinia and SPEC OMP2012 benchmarks.

Lastly, we examine the overall potential of co-scheduling. We take a set of applications to be scheduled and assign one processor to each application, and all processors to one application. Then, we compare these two strategies with an integer program. This integer program represents the *Optimization Potential*, and we show that there is an optimization potential in scheduling with elaborate algorithms.

3.1 Novel Execution Framework for Co-Scheduling Applications

Before starting with runtime experiments, we introduce the execution framework used for our experiments. The idea was to create a resource manager, like SLURM [49], but for managing resources on one compute node, i.e., the cores. We use this framework¹ as resource manager and scheduler: different applications should be executed on one compute node with a given number of cores. As an example, we take a look at the execution of three `lavaMD` applications from the Rodinia benchmark suite in Listing 3.1.

Listing 3.1: Example input file for the resource manager and scheduler.

```
# JOBS
1;./lavaMD -cores 16 -boxes1d 40;0-15
2;./lavaMD -cores 16 -boxes1d 20;16-31
3;./lavaMD -cores 5 -boxes1d 15;0-10:2
# ORDER
0: 1,3
3: 2,1
```

There are two macros `# JOBS` and `# ORDER` that are used for separating the application commands from the execution ordering. After `# JOBS`, we can name execution commands

¹https://gitlab.com/bsarkoez/resource-manager_scheduler

with the syntax `id;command;cores` of applications. The `cores` part may be a single core number, a range of cores, an enumeration of cores, or an interleaved range of cores. In our example, we execute three jobs using the `lavaMD` application with different sizes, i.e., different `boxes1d`, and on different cores. Job 3 uses the interleaved core range syntax and is executed on the cores 0,2,4,6,8. The `# ORDER` represents the order in which jobs are submitted to the resource manager. This simulates live behaviors on multicore machines, where jobs are submitted at different times to the compute node. As in the example, jobs 1 and 3 are submitted at time 0, i.e., without delay. If we look at the core assignment, we notice that these two jobs cannot be executed concurrently, since they demand same cores. Which one of job 1 and job 3 will be executed first, is the scheduler's assignment. It is possible to configure different scheduling strategies, and in this example we use the `create-schedule-in-order` strategy, i.e., we execute job 1 first, and after it has finished, we execute job 3. After a time delay of three seconds, job 2 and job 1 are submitted. Assuming that job 1 takes 1.5 seconds, and job 3 takes 2.5 seconds, we can already start executing job 2, while job 3 is running, since there are no resource sharing conflicts. After job 3 finishes, we execute job 1 again, while job 2 may still be running. This scenario shows the need for this resource manager: if we use `create-schedule-in-order`, while the jobs at timestamp 3 are submitted as 1,2, there would be a resource conflict.

To make sure that applications are only executed on the given number of cores, the resource manager executes application commands using an affinity mapping with `numactl --physcpubind=<binding> <command>`. This means that job 1 would be executed as `numactl --physcpubind=0-15 ./lavaMD -cores 16 -boxes1d 40`.

Using this execution framework, we can provide one input file, which contains the jobs that need to be executed and the job ordering. Assuming that the input file is called `launching_order` and we want to execute it on one compute node of the `hydra` cluster, we can submit the following command to SLURM on `hydra`: `main.py -c 32 -f launching_order`. Then, our list of applications from the file `launching_order` will be executed on one compute node with 32 cores.

Some further examples clarify the usage and behavior of our execution framework. For the runtime experiments we use the ordering given by `# ORDER`. This means that the scheduling policy used by the internal scheduler is `create-schedule-in-order`.

Listing 3.2: Executing `lavaMD` three times sequentially on core 0.

```
# JOBS
1;./lavaMD -cores 1 -boxes1d 15;0
2;./lavaMD -cores 1 -boxes1d 15;0
3;./lavaMD -cores 1 -boxes1d 15;0
# ORDER
0: 1,2,3
```

Listing 3.2 shows the execution of three `lavaMD` applications executed serially on core 0. Since all three jobs are submitted at time 0 and they all require the same core, they

will be executed one after another, i.e., job 2 starts after job 1 finishes, job 3 starts after job 2 finishes.

Listing 3.3: Executing `lavaMD` three times concurrently with one core.

```
# JOBS
1; ./lavaMD -cores 1 -boxes1d 15;0
2; ./lavaMD -cores 1 -boxes1d 15;1
3; ./lavaMD -cores 1 -boxes1d 15;2
# ORDER
0: 1,2,3
```

Listing 3.3 shows the execution of three `lavaMD` applications executed concurrently using a single core. Since all three jobs are submitted at time 0, but can be executed on different cores, they all start their execution at time 0 and will be executed concurrently.

Listing 3.4: Pausing between two executions.

```
# JOBS
1; ./lavaMD -cores 20 -boxes1d 30;0-19
2; ./lavaMD -cores 20 -boxes1d 30;12-32
# ORDER
0: 1
10: 2
```

Sometimes, we want to make sure that jobs are not interleaving one another. Then, we can add a higher time delay for the start of the execution. For the job executions in Listing 3.4, we assume that this `lavaMD` execution using 20 cores takes around 2 seconds. Even though they share cores, we could start job 2 after job 1 finishes. But using this order, the execution framework waits ten seconds before submitting and executing job 2.

3.2 Single Core Execution Behaviors

We start by executing our applications on a single core. We examine differences between the running time of sequential executions with one core and a concurrent execution of an application with itself using a single core.

As already mentioned in Chapter 2, `hydra` has two sockets. We want to make sure that the execution time of any application is not influenced by the socket on which it is executed. Further on, executing the same application concurrently on all available cores leads to the matter of resource sharing, since the same application requires the same resources. With this motivation, we define the following three execution strategies for one application using a single core:

Case Seq 1

We run the application on one thread of socket one in a serial way. For socket one, we use core 0 to run one application 32-times one after another. Each execution of the application is called a job. The corresponding Gantt chart of this strategy is shown in Figure 3.1.

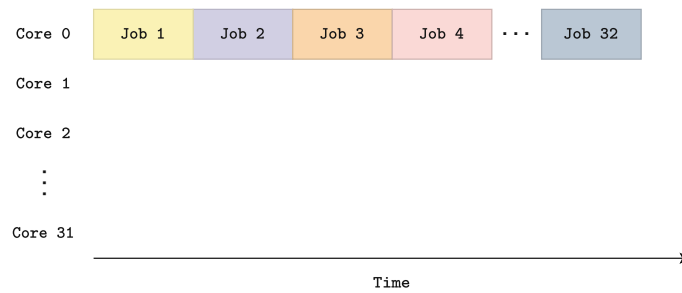


Figure 3.1: Gantt Chart of the execution strategy **Case Seq 1** with 32 jobs and cores.

Case Seq 2

We run the application on one thread of socket two in a serial mode. The first core of socket two is core 1, as the affinity mapping for hydra shows in Listing 2.1. We run an application 32-times one after another on core 1. The corresponding Gantt chart of this strategy looks like Figure 3.2.

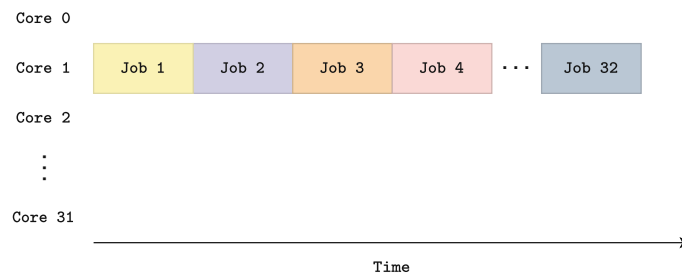


Figure 3.2: Gantt Chart of the execution strategy **Case Seq 2** with 32 jobs and cores.

Case Concur

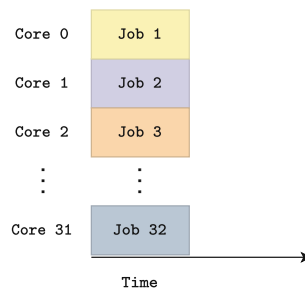


Figure 3.3: Gantt Chart of the execution strategy **Case Concur** with 32 jobs and cores.

Since our testing machine hydra has 32 cores on one compute node, we take all available cores and execute the same application concurrently on all available cores. The corresponding Gantt chart is shown in Figure 3.3.

We compare these three defined strategies `case seq 1`, `case seq 2`, and `case concur` for both benchmark suites Rodinia and SPEC OMP2012. We use `gcc` as the default compiler, but also examine the sequential and concurrent case execution using `icc`, `pgcc`, and `clang`. Table 3.1 shows the used compiler versions and commands used for the compilation.

The running times shown in the following are the average execution times of a job, where a job corresponds to the notion of a job in the execution strategies from Figures 3.1 – 3.3.

Table 3.1: Compiler overview.

	gcc	icc	pgcc	clang
Version	10.2.0	19.1.304	20.7-0	11.1.0
C	gcc	icc	pgcc	clang
C++	g++	icc	pgc++	clang++
Fortran	gfortran	ifort	pgfortran	
OpenMP flag	-fopenmp	-qopenmp	-mp	-fopenmp
Optimization flags	-O2	-O2	-O2	-O2
	-O3	-O3	-O3	-O3
	-ffast-math	-fast	-Mfprelaxed	-ffast-math
	-ffree-form	-FR	-Mfree	

3.2.1 Execution Experiments with Rodinia

Rodinia has no provided input instances for its benchmarks, therefore we defined a *small* and *medium* input size for the running time experiments in Table 2.3 and Table 2.4. For the single core execution experiments with different compilers, we only use the *small* input size. We measure the **kernel time** of each benchmark, the time of the computational kernel, which excludes the time needed for pre- and postprocessing steps. Additionally, we measure the **real user time**, i.e., the wall clock time. This kernel time measurement is added manually to almost all benchmarks, only a few contain a time measurement around their kernel section. The kernel section is clearly visible for the Rodinia benchmarks due to a simple structure of the code.

Differences between Sequential and Concurrent Cases

In the following, we see results obtained from the comparison of the different execution strategies `case seq 1`, `case seq 2`, and `case concur` using the Rodinia benchmark suite applications. Each application is executed 32 times, in either a sequential or concurrent way. The plots show the mean time, while error bars indicate the deviation (minimum and maximum times) of these 32 time measurements.

Figure 3.4 shows the achieved kernel times on `hydra` for the three different execution strategies using `gcc` as the compiler.

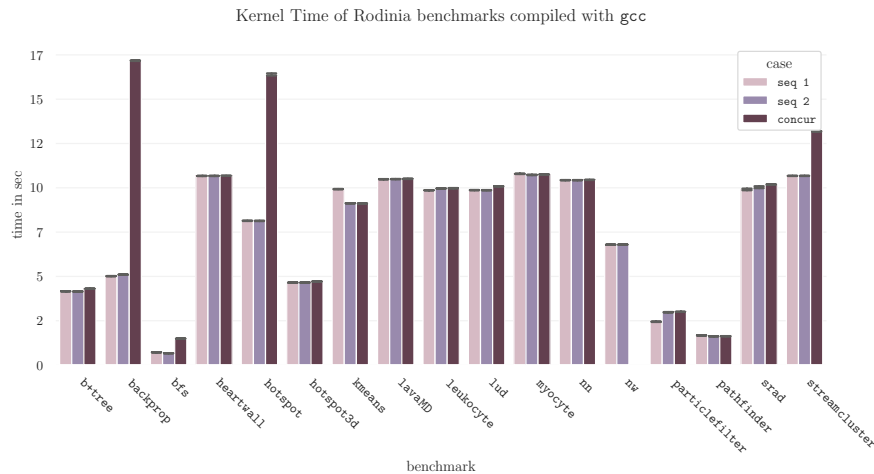


Figure 3.4: Kernel runtime of the *small* Rodinia benchmarks compiled with `gcc` from 32 runs.

One assumption was that using different sockets for the same execution order does not affect the running time. The results of these experiments confirm this assumption, since there are hardly run time differences between the cases `seq 1` and `seq 2`. Looking at the application execution times closer, we see that some applications, like `backprop` or `hotspot`, perform badly in the concurrent execution. This could mean that these applications use many resources which are rare when executed concurrently with itself.

For the `nw` application, we did not obtain valid results for the case `concur` execution. The results obtained from these runs are invalid, since the execution exited with failure due to memory allocation errors. Such invalid runs are excluded from our results by ignoring their results, as the empty bar indicates.

Additionally to the kernel time, Figure 3.5 shows the user time of the Rodinia applications compiled with `gcc`. We observe similar differences between the sequential and concurrent execution for the user times. Interestingly, `pathfinder` only seems to have concurrent runtime issues with itself in the user time measurements, but not in the kernel time measurements, which leads to the assumption that the pre- and postprocessing steps are resource conflicting.

As this clearly shows, it is indeed important to differentiate between kernel and user time measurements, since some artifacts may occur in pre- or postprocessing steps, but are irrelevant for the kernel section itself.

Arithmetic Intensity

Since there are big differences for some applications between a sequential and concurrent execution, we take a closer look on possible reasons behind this. For this purpose, we measure performance counters for the sequential case `seq 1` using `likwid`. We measure

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

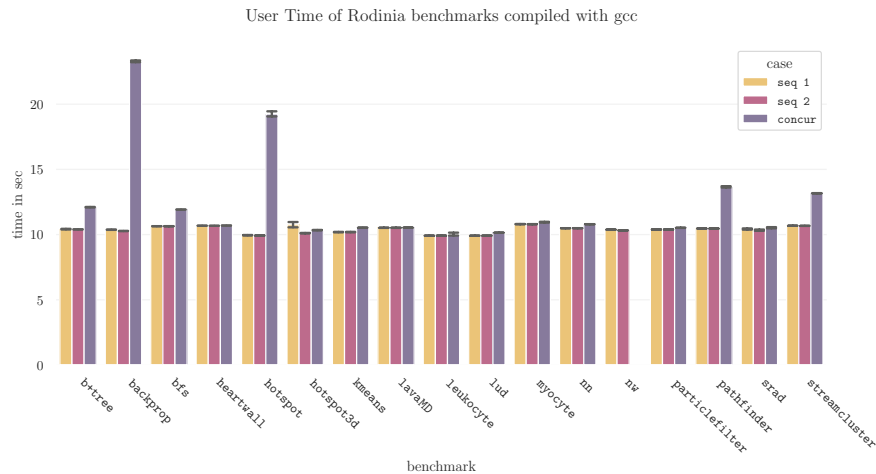


Figure 3.5: User time of the *small* Rodinia benchmarks compiled with `gcc` from 32 runs.

three runs and use the median values of the kernel assessment to calculate the arithmetic intensity. Table 3.2 gives an overview of the used performance counters, where each counter value was measured on core 0, since we look at case seq 1.

Table 3.2: Performance counters obtained with `likwid-perfctr`.

	Group	Performance Counter
Memory Bandwidth	MEM_SP and MEM_DP	Memory bandwidth [MBytes/s]
Flops Single Prec.	MEM_SP	SP [MFLOP/s]
Flops Double Prec.	MEM_DP	DP [MFLOP/s]
Arithmetic Intensity	MEM_SP or MEM_DP	Operational intensity
L3 Cache Misses	L3CACHE	MEM_LOAD_RETIRED_L3_MISS
L3 Cache Miss Ratio	L3CACHE	L3 miss ratio

The arithmetic intensity is already calculated for both the single and double precision performance groups. For a better visualization of the arithmetic intensity, a roofline model is helpful to differentiate between compute- and memory-bound applications.

Figure 3.6 shows the roofline model for the Rodinia benchmark applications with a *small* input size. The peak bandwidth and peak performance used to plot the roofline model are taken from all measured performance counters, including the performance counters obtained from the SPEC OMP2012 benchmark suite too (see later). Applications below the *Bandwidth Roof* are called *memory-bound* and applications below the *Performance Roof* are called *compute-bound*. Benchmarks that are not plotted in this roofline model have an arithmetic intensity of less than $1/8$ and are memory-bound.

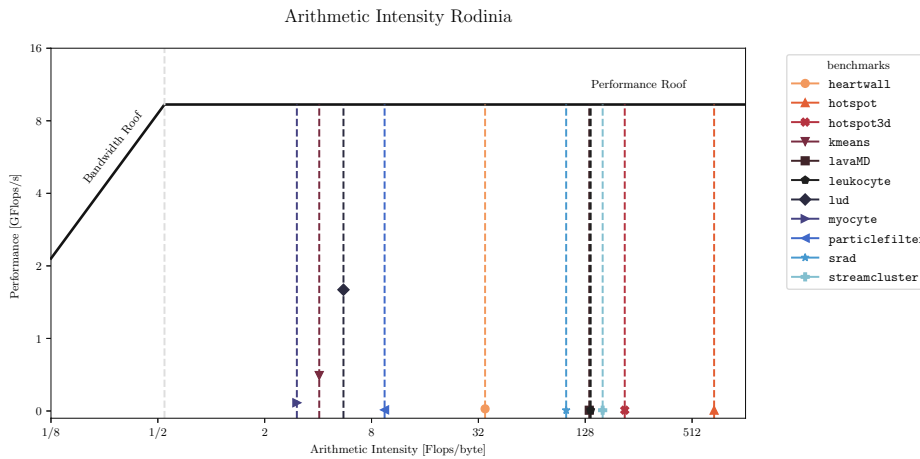


Figure 3.6: Arithmetic intensity of the *small* Rodinia benchmarks compiled with `gcc` for the `seq 1` case. `b+tree`, `backprop`, `bfs`, `nn`, `nw`, and `pathfinder` are outliers with an AI $< 1/8$.

The roofline model for the *small* Rodinia benchmarks is plotted in Figure 3.6. There are no applications depicted under the bandwidth roof, but some applications are not even represented in this model: `b+tree`, `backprop`, `bfs`, `nn`, `nw`, and `pathfinder` are not plotted because their arithmetic intensity is too small, $AI < 1/8$. These applications are therefore under the bandwidth roof and thus *memory-bound*. The other applications are correspondingly *compute-bound*.

Table 3.3 gives a detailed overview for the executions with a *small* input size of the running times, the proportional running time difference, the measured arithmetic intensity, whether an application is memory-bound (MB) or compute-bound (CB) and the L3 cache misses and cache ratios. The entries are sorted by the proportional time difference in descending order.

Using `likwid` to get performance counters for a specific region, i.e., using the Marker API, there are some unresolved problems with `particlefilter` and `gcc`. Therefore, we compile `particlefilter` with `clang` to get these performance metrics. All other benchmarks are analyzed with the performance groups `MEM_SP` and `MEM_DP`, but these performance groups lead to memory allocation problems with `particlefilter`. Therefore, we measure the memory bandwidth and the floating point operations using the performance groups `MEM` and `FLOPS_DP` instead of `MEM_DP`, from which we can calculate the same metrics used from `MEM_DP`.

As we can see in this overview in Table 3.3, a large time difference between the sequential and concurrent execution does not necessarily mean a high arithmetic intensity implying compute-bound applications, like `backprop` shows, but also does not always indicate a memory-bound application, like `hotspot` shows.

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

Table 3.3: Arithmetic intensity overview of the *small* Rodinia apps compiled with `gcc` for the kernel measurements. Time Diff ... time difference between `seq 1` and `concur`, MB ... memory-bound, CB ... compute-bound, AI in Flops/Byte

App	Seq 1	Concur	Time Diff	SP DP	AI	MB CB	L3 cache misses	L3 miss ratio
backprop	5.04 s	17.18 s	240.87%	SP	0.01	MB	381 594 900	1.00
bfs	0.67 s	1.49 s	122.39%	DP	0.00	MB	1 544 947	0.09
hotspot	8.12 s	16.41 s	102.09%	SP	681.30	CB	4 507	0.00
streamcluster	10.67 s	13.18 s	23.52%	SP	160.45	CB	965	0.00
b+tree	4.15 s	4.30 s	3.73%	SP	0.00	MB	1 679 451	0.36
lud	9.87 s	10.09 s	2.23%	SP	5.56	CB	2 002 875	0.48
srad	10.11 s	10.22 s	1.09%	SP	99.94	CB	630	0.00
hotspot3d	4.66 s	4.71 s	1.07%	SP	213.68	CB	88	0.00
pathfinder	1.62 s	1.63 s	0.62%	SP	0.00	MB	73 362	0.62
particlefilter	2.99 s	3.00 s	0.33%	DP	9.48	CB	2 481	0.07
myocyte	10.72 s	10.75 s	0.28%	SP	3.04	CB	3 498	0.11
leukocyte	9.96 s	9.97 s	0.10%	DP	137.33	CB	2 643	0.01
nn	10.43 s	10.44 s	0.10%	SP	0.03	MB	107	0.04
lavaMD	10.49 s	10.50 s	0.10%	DP	134.86	CB	3 400	0.32
heartwall	10.67 s	10.68 s	0.09%	SP	34.91	CB	7 669	0.03
nw	6.78 s	-	-	SP	0.00	MB	192 215 300	1.00
kmeans	9.12 s	9.11 s	-0.11%	SP	4.06	CB	432 122	0.88

Since some sequential kernel times are small, we also take a look at the running times and performance metrics of the *medium* input size of the Rodinia applications. Table 3.4 shows an overview of the achieved sequential and concurrent kernel times for the *medium* input size of the Rodinia benchmarks and the corresponding performance counter values. The `backprop` application was not executed 32-times concurrently with a single core (as the other benchmarks), but only 14-times concurrently. The reason for this behavior is that an execution of 16 or more concurrent `backprop` applications leads to an execution time of more than one hour for each concurrently executed `backprop` program on `hydra`. Therefore, the relative time difference is even much larger than shown in this table for the case of executing `backprop` 32-times concurrently on `hydra`.

The results from Table 3.4 show similar results regarding the relative time differences for the top four applications `backprop`, `bfs`, `hotspot`, and `streamcluster`. We keep these applications in mind because they seem to have problems with resource sharing.

Table 3.4: Arithmetic intensity overview of *medium* Rodinia apps compiled with `gcc`. Time Diff ... time difference between `seq 1` and `concur`, MB ... memory-bound, CB ... compute-bound, AI in Flops/Byte

App	Seq 1	Concur	Time Diff	SP DP	AI	MB CB	L3 cache misses	L3 miss ratio
backprop	17.15 s	164.89 s	861.49%	SP	0.01	MB	1 361 258 000	1.00
myocyte	50.13 s	101.45 s	102.37%	SP	2.67	CB	14 409	0.10
bfs	3.64 s	7.12 s	95.74%	DP	0.00	MB	23 560 420	0.35
hotspot	42.55 s	78.96 s	85.57%	SP	1.53	CB	39 901 740	0.60
streamcluster	51.19 s	55.67 s	8.75%	SP	0.47	MB	74 594 010	0.77
srad	48.12 s	50.08 s	4.07%	SP	102.99	CB	1 514	0.00
hotspot3d	23.65 s	23.84 s	0.82%	SP	7.50	CB	69 020	0.11
pathfinder	5.75 s	5.78 s	0.52%	SP	0.00	MB	260 591	0.60
nn	48.77 s	48.88 s	0.22%	SP	0.01	MB	977	0.00
heartwall	49.71 s	49.76 s	0.10%	SP	35.97	CB	22 833	0.02
b+tree	26.59 s	26.61 s	0.08%	SP	0.00	MB	227 557	0.70
leukocyte	51.60 s	51.62 s	0.04%	DP	139.73	CB	22 302	0.06
lavaMD	51.88 s	51.90 s	0.04%	DP	109.07	CB	20 789	0.35
kmeans	43.08 s	43.05 s	-0.08%	SP	3.90	CB	891 965	0.90
particlefilter	3.81 s	3.75 s	-1.57%	DP	11.86	CB	4 495	0.06
lud	46.29 s	45.51 s	-1.70%	SP	3.80	CB	61 019 080	0.07

Compiler Comparison

For the previous experiments, we used `gcc` as the compiler, since it is also the default *setting* in the makefiles of the Rodinia benchmarks. But nonetheless, it should also be possible to use other compilers. For this reason, we explore the differences when other compilers are used. We compare the `gcc` kernel runtime of the case `seq 1` and the case `concur` with the runtime compiled with `icc`, `pgcc`, and `clang`. The other sequential case, case `seq 2`, can be left out from these experiments since our assumption of equal running times on different sockets was already confirmed. We have already seen the used versions of compilers, compile commands, and compile flag for OpenMP in Table 3.1.

Figure 3.7 shows the comparison of the compilers `gcc`, `icc`, `pgcc`, and `clang` used for the sequential case `seq 1`, and Figure 3.8 shows this comparison for the concurrent case `concur`. From both these comparisons, we cannot see a clear winner in terms of constantly smaller running times, but we see that using another compiler may affect the runtime of some applications. Still, the `icc` compiler performs better for many applications like `kmeans`, `myocyte`, `nn`, `srad`, and `streamcluster` since the running time differences compared to `gcc` are improved by at least a factor of two. The `pgcc`

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

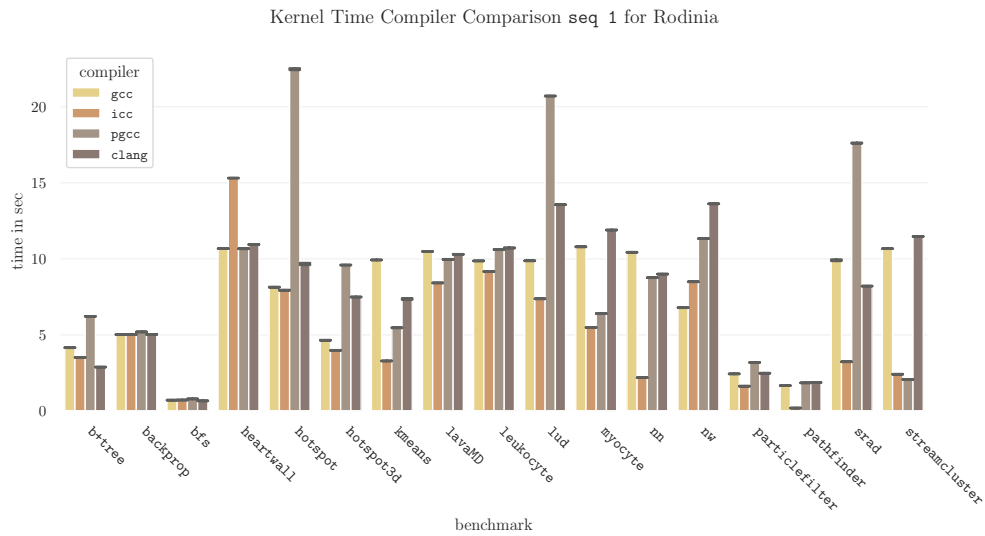


Figure 3.7: Kernel times of the *small* Rodinia benchmarks compiled with gcc, icc, pgcc, and clang for the seq 1 case (32 executions).

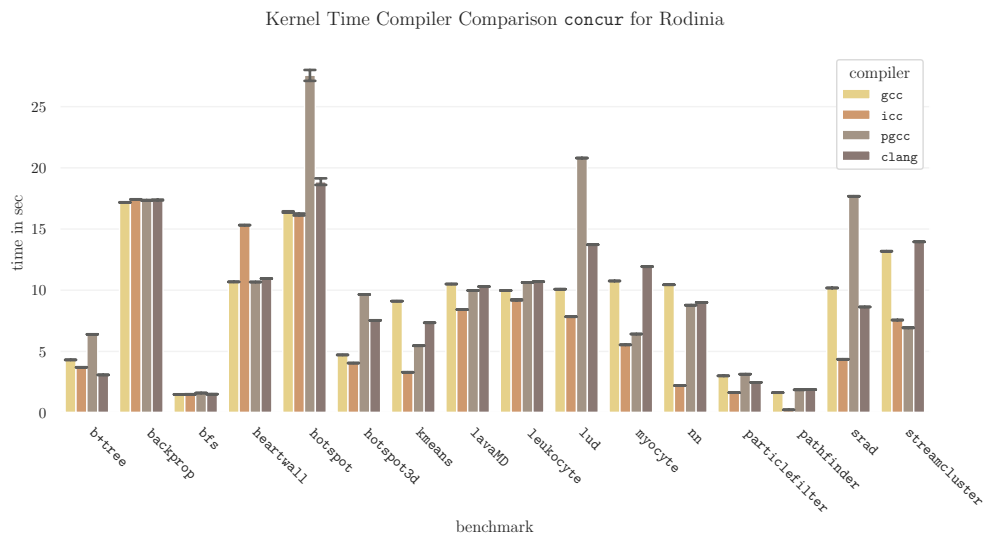


Figure 3.8: Kernel times of the *small* Rodinia benchmarks compiled with gcc, icc, pgcc, and clang for the concur case (32 concurrent executions).

compiler also leads to some faster running times than gcc, but hardly outperforms icc and often even leads to slower execution times. An interesting observation is that higher concurrent runtimes are not influenced by changing compiler settings, as the concurrent running time of, e.g., backprop is high for all compilers, while the sequential time is comparatively smaller. Outliers coming from compiler differences are also reflected

in Figures 3.7 and 3.8, like the `pgcc` measurement for `hotspot`. So we see that it does make a difference what compiler we use, but the compiler does not change the runtime behavior of applications. To double check our findings so far, we will do the same experiments with the SPEC OMP2012 benchmark suite.

3.2.2 Execution Experiments with SPEC OMP2012

The SPEC OMP2012 benchmark suite already provides input instances, therefore we do not need to define them. In the following experiments, we use the `test` dataset size, which corresponds to the smallest provided input size. Still, the serial user running time of this `test` data size varies for the 14 provided applications, as many applications are executed with a single core in a few seconds, and others take up to four minutes. Therefore, we exclude `md`, `bt331`, `ilbdc`, and `applu331` from the following experiments due to their different magnitude of serial execution time.

Regarding analyzing the execution time, SPEC OMP2012 already provides a very stable time measurement shown as `runtime=<time>` in the log output file of an execution. This provided time corresponds to the *user time* as described for Rodinia, the time needed for the application's main function. The SPEC benchmarks are not designed for measuring pure kernel time of parallel constructs, since source code changes are not intended and not accepted as official benchmarking results. For measuring performance counters with the Marker-API of `likwid`, we need to add the start and end of the marker to the start and end of the main function. Many SPEC OMP2012 do not have a single kernel section, but do some kernel calculations, then some pre- and postprocessing operations, then kernel calculations again. This makes it hard to narrow down the measurements to the kernel section solely. Therefore, we will only analyze the benchmark execution times provided in the log output files with `runtime=<time>`.

We do not use the real input size of the SPEC OMP2012 benchmarks, i.e., the *ref* input size, due to its long running time, and more importantly long serial running time. Regarding measuring kernel times, this long runtime of the SPEC OMP2012 benchmarks renders the time needed for pre- and post-processing steps irrelevant, which may be a reason that kernel sections are not explicitly clocked. Still, this input size is too large for our use cases.

Differences between Sequential and Concurrent Cases

We start by comparing the three execution strategies `case seq 1`, `case seq 2`, and `case concur` using a single core for the benchmark execution time. As for the Rodinia benchmarks, we assume that there is no clear time difference between `case seq 1` and `case seq 2` since both underlying sockets are equal. An interesting question is if we find similar relative time differences for some benchmarks between a purely sequential and purely concurrent execution.

Figure 3.9 shows the execution time measurements for the single core execution cases of the SPEC OMP2012 benchmarks. In the plot, we see almost no difference between

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

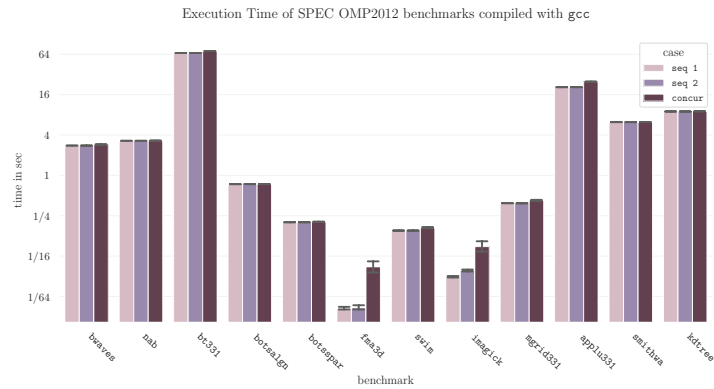


Figure 3.9: Benchmark execution times of the *test* SPEC OMP2012 benchmarks compiled with gcc.

the two serial execution cases *seq 1* and *seq 2*, which confirms the assumption of no runtime differences between the two sockets on *hydra*. Regarding the difference between *seq 1* and *concur*, we observe small differences, like for *bt331* or *applu331*. Even though there seems to be a big relative difference for the concurrent case for *fma3d* and *imagick*, this difference is so small in absolute values that it can be considered irrelevant.

Arithmetic Intensity

The roofline model for the SPEC OMP2012 *test* size benchmarks is shown in Figure 3.10.

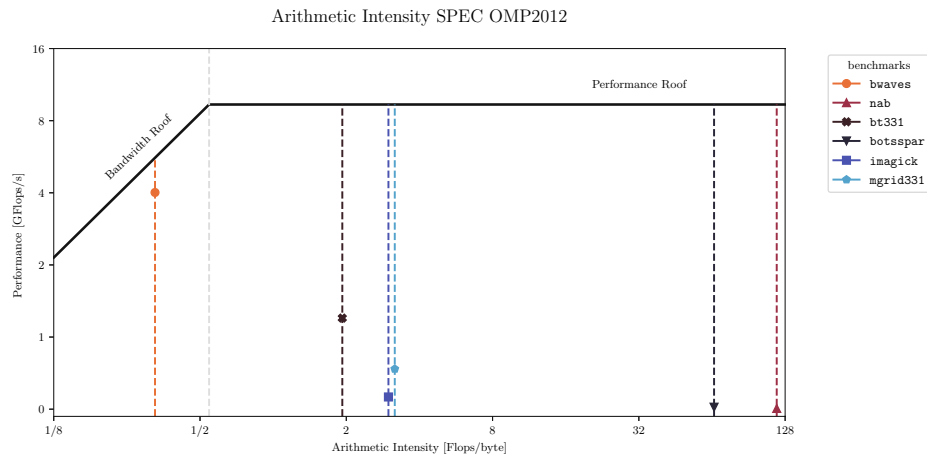


Figure 3.10: Arithmetic intensity of the *test* SPEC benchmarks compiled with gcc for the *seq 1* case. *botsalgn*, *fma3d*, *swim*, *applu331*, *smithwa*, and *kdtree* are outliers with an AI < 1/8.

For the calculation of the arithmetic intensity, we used the median values of the performance counters obtained from three executions. In Figure 3.10, we see one memory-bound application, *bwaves*, but the applications *botsalgn*, *fma3d*, *swim*, *applu331*, *smithwa*, and *kdtree*, which are not plotted, are also memory-bound with an arithmetic intensity of less than $1/8$ flops per byte. The other applications are compute-bound.

More details about the arithmetic intensity and performance metrics are shown in Table 3.5. We see an overview of the running time and arithmetic intensity for the *test* size of the SPEC OMP2012 benchmarks. We observe the running times, the proportional running time difference, the measured arithmetic intensity, whether an application is memory-bound (MB) or compute-bound (CB), and the L3 cache misses and miss ratios. The entries are sorted by the proportional time difference in descending order.

Table 3.5: AI overview of *test* SPEC OMP2012 applications compiled with `gcc`. Time Diff ... time difference between `seq 1` and `concur`, MB ... memory-bound, CB ... compute-bound, AI in Flops/Byte

App	Seq 1	Concur	Time Diff	SP DP	AI	MB CB	L3 cache misses	L3 miss ratio
<i>fma3d</i>	0.01 s	0.04 s	300.00%	DP	0.01	MB	344	0.50
<i>imagick</i>	0.03 s	0.07 s	133.33%	DP	2.98	CB	444	0.21
<i>applu331</i>	20.66 s	25.01 s	21.03%	SP	0.00	MB	1 765 208	0.02
<i>mgrid331</i>	0.39 s	0.42 s	9.09%	DP	3.17	CB	1 038	0.01
<i>bt331</i>	66.88 s	71.63 s	7.12%	DP	1.93	CB	110 281 300	0.51
<i>swim</i>	0.15 s	0.16 s	6.67%	SP	0.00	MB	20 796	0.81
<i>bwaves</i>	2.78 s	2.89 s	3.96%	DP	0.33	MB	1 035 522	0.94
<i>kdtree</i>	8.97 s	9.00 s	0.39%	DP	0.00	MB	16 338	0.00
<i>nab</i>	3.28 s	3.29 s	0.30%	DP	118.20	CB	471	0.40
<i>smithwa</i>	6.23 s	6.24 s	0.16%	DP	0.00	MB	60	0.04
<i>botsalgn</i>	0.74 s	0.74 s	0.00%	DP	0.00	MB	99	0.28
<i>botsspar</i>	0.20 s	0.20 s	0.00%	SP	65.28	CB	708	0.06

From these measurements in Table 3.5, we cannot clearly say if there is a significant relative difference between sequential and concurrent executions, since this relative difference is only significant for *fma3d* and *imagick*, where the running times are so small that no reasonable conclusions can be drawn. Therefore, we measure these performance metrics for the *train* input size, which expectedly leads to a higher sequential kernel time. The measured performance values for the *train* input size are presented in Table 3.6.

We see a change in the ordering of Table 3.6, which means that the relative differences between sequential and concurrent executions change. For the *train* input size, we now see an increased relative difference for *swim*, *mgrid331*, and *applu331*. Even though

Table 3.6: AI overview of *train* SPEC OMP2012 applications compiled with `gcc`. Time Diff ... time difference between `seq 1` and `concur`, MB ... memory-bound, CB ... compute-bound, AI in Flops/Byte

App	Seq 1	Concur	Time Diff	SP DP	AI	MB CB	L3 cache misses	L3 miss ratio
swim	29.38 s	111.29 s	278.80%	SP	0.00	MB	65 578	0.92
mgrid331	2.96 s	7.20 s	143.24%	DP	0.47	MB	51 415	0.25
applu331	38.46 s	61.56 s	60.08%	SP	0.00	MB	38 080 840	0.22
botsspar	1.49 s	2.00 s	34.23%	SP	89.79	CB	3 544	0.05
fma3d	56.33 s	68.62 s	21.83%	DP	0.44	MB	384 351 000	0.65
bwaves	24.91 s	28.20 s	13.23%	DP	0.29	MB	11 007 920	0.97
bt331	282.13 s	292.57 s	3.70%	DP	1.50	CB	635 414 200	0.79
imagick	175.34 s	177.37 s	1.15%	DP	165.57	CB	12 045	0.02
nab	274.18 s	275.76 s	0.58%	DP	124.14	CB	2 599	0.00
kdtree	117.31 s	117.56 s	0.22%	DP	0.00	MB	562 424	0.02
botsalgn	18.85 s	18.82 s	-0.13%	DP	0.00	MB	123	0.15
smithwa	23.92 s	23.88 s	-0.17%	DP	0.00	MB	166	0.00

these two input sizes do not correlate for the relative difference between `seq 1` and `concur`, we can at least observe similarly as for Rodinia that some benchmarks have resource conflicts when executed concurrently with itself.

The running time of the different benchmarks is still problematic for comparison since the magnitudes of the runtimes are so different. Additionally, using the big input size *ref*, we need at least 15 minutes for one sequential run, which makes this big input size non-applicable for our purposes of getting to know applications in a short period of time.

Compiler Comparison

Similarly to Rodinia, we compare the running times using different compilers for the cases `seq 1` and `concur`, shown in Figures 3.11 and 3.12. We execute the applications 32 times either sequentially or concurrently and therefore use the median values of these 32 measurements for plotting, where error bars indicate minimum and maximum obtained values.

The observation from Rodinia, that `icc` often leads to better running times, is still valid, while also `pgcc` seems to be a good choice for fast running times. As we can see in both plots, for some applications like `bwaves` or `bt331` the `clang` compiler measurements are missing. Since these applications (`bwaves`, `bt331`, `fma3d`, `swim`, `mgrid331`, or `applu331`) are written in Fortran, we cannot use `clang`, which only supports C and C++.

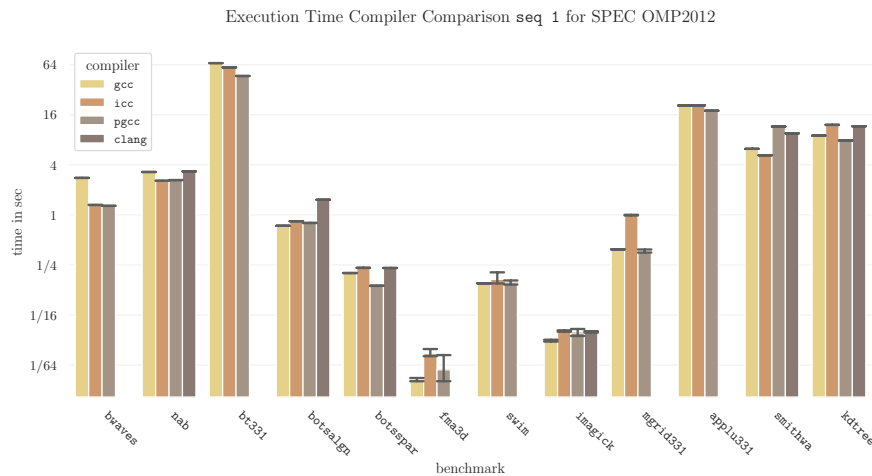


Figure 3.11: Benchmark execution times of the SPEC OMP2012 benchmarks compiled with gcc, icc, and pgcc for the seq 1 case.

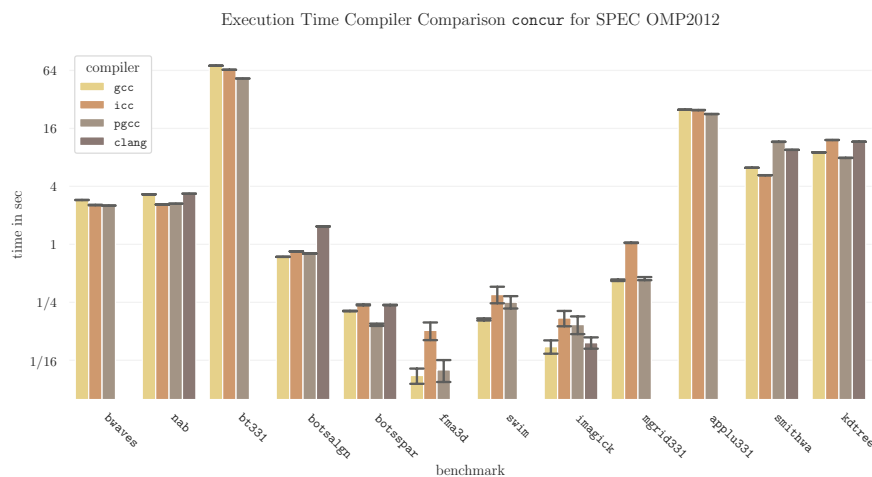


Figure 3.12: Benchmark execution times of the SPEC OMP2012 benchmarks compiled with gcc, icc, and pgcc for the concur case.

Looking back at our examination of the Rodinia and SPEC OMP2012 benchmarks, we notice some benchmarks that seem to have resource sharing problems when executed concurrently with itself. The arithmetic intensity additionally does not help figuring out whether an application might have problems with resource sharing or not. Further, we need to pay attention, which compiler is used since the compiler does affect the runtime itself.

3.3 Parallel Scalability

We look at the scalability of our benchmarks from the Rodinia and SPEC OMP2012 suite. Additionally to the *small* and *test* input size, we also use the *medium* and *train* input size. We measure the kernel time five times for 1, 2, 4, 8, 16, and 32 cores. First, we take a look at the mean kernel running times, how they scale with the number of cores. Then, we additionally plot the parallel efficiency and strengthen our statement from Chapter 1: using all available cores often leads to an *insufficient parallel efficiency*, i.e., the parallel efficiency is smaller than an *acceptable threshold*.

3.3.1 Scalability of Rodinia Benchmarks

For the Rodinia benchmark suite's parallel scalability tests, we measure kernel times achieved with different compilers, `gcc`, `icc`, and `clang`.

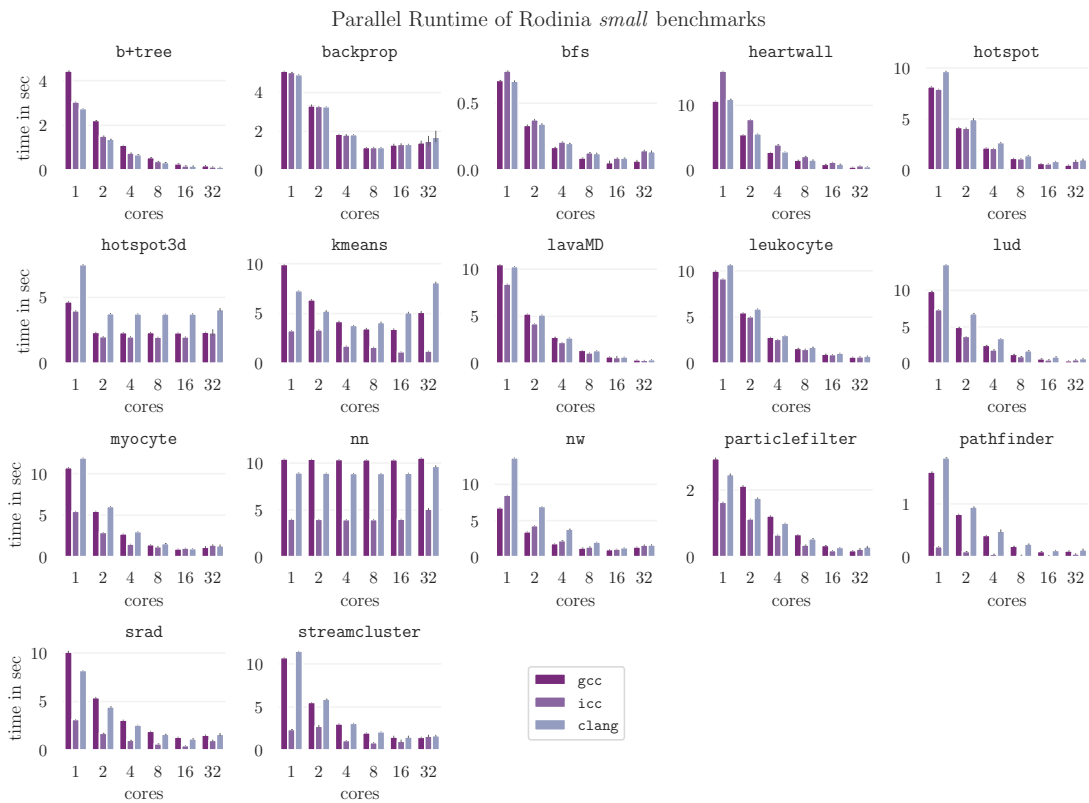


Figure 3.13: Parallel scalability of the Rodinia benchmark suite for the *small* input size applications on hydra with 5 repetitions.

The obtained kernel running times for the parallel scalability of the Rodinia benchmark suite with a *small* input size are shown in Figure 3.13. We see that most of the applications scale well, e.g., `b+tree`, `heartwall`, `lavaMD`. In comparison, `nn` does not scale well for

our defined input size (even though we set the environment variable `OMP_NUM_THREADS` accordingly to the number of cores). Looking at the usage of different compilers, we notice runtime differences. Still, the scaling behavior is similar for the compilers. As an example, we look at `kmeans` and observe the same scaling behavior for `gcc`, `icc`, and `clang`: the kernel time decreases using more cores until we reach 8 or 16 cores, afterwards the kernel time increases again.

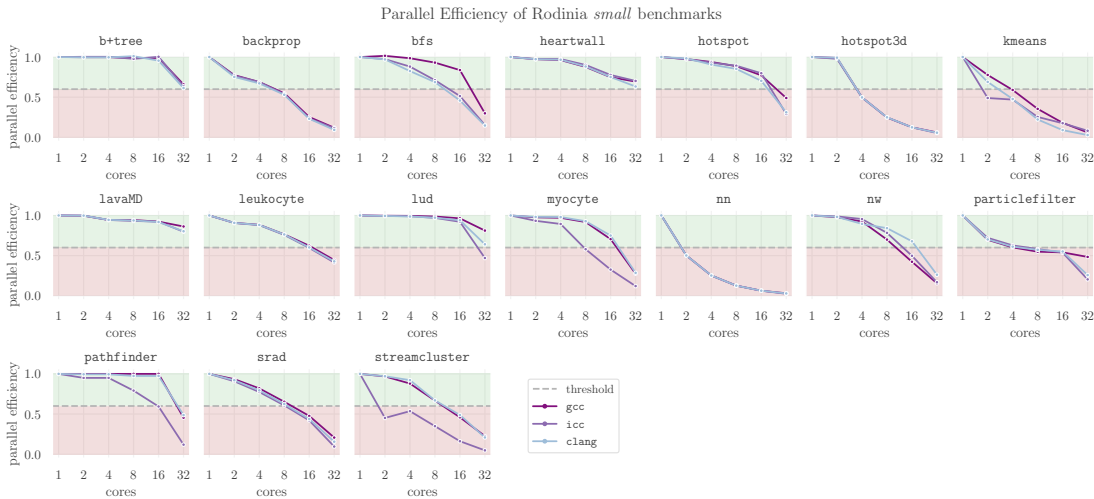


Figure 3.14: Parallel efficiency of the *small* sized Rodinia benchmark suite. The green area marks *acceptable* efficiencies over 60%, while the red area marks too low values.

The parallel efficiency of the Rodinia benchmarks with a *small* input size is plotted in Figure 3.14. For each compiler separately, we calculate the parallel efficiency. This means that the parallel efficiencies are relative to their corresponding compiler. We mark an acceptable threshold for the parallel efficiency as 60%. Since the optimal parallel efficiency would be one for any number of cores, we notice some benchmarks that scale well: `b+tree`, `heartwall`, and `lavaMD`. Regarding well scaling applications, we would assume that they are compute-bound, since memory-bound applications often point out memory limitations. Interestingly, not all well scaling applications are compute-bound, e.g., `b+tree` is a memory-bound application. Still, these three well-scaling applications are a minority considering a total of 18 applications. Nonetheless, this observation shows that it is often reasonable to run an application with less cores instead of fully parallelizing it.

Next, we look at the *medium* input size of the Rodinia benchmarks and capture the scaled kernel times in Figure 3.15, and the parallel efficiencies of the benchmarks in Figure 3.16. As mentioned earlier, we do not define `nw` for the *medium* input size, which is why `nw` is missing from the plots for the *medium* size.

The conclusions we draw from both the parallel scalability and parallel efficiency for the *medium* input instances are almost equal: some applications scale fairly well and therefore

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

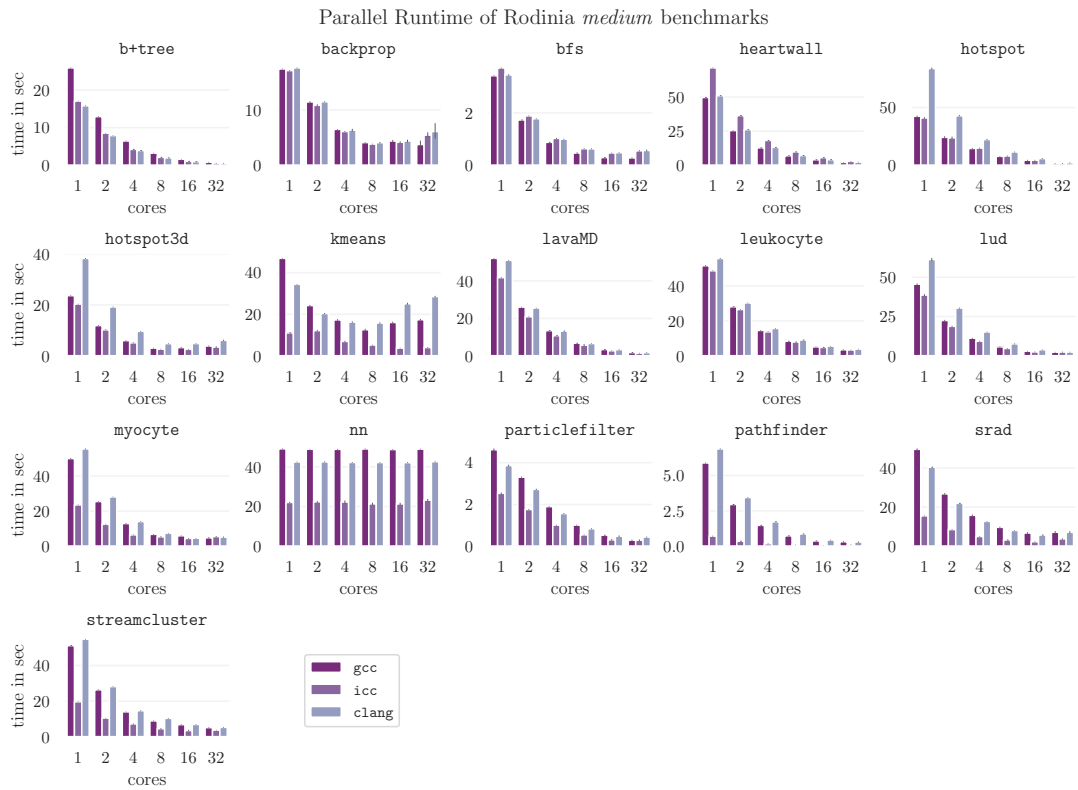


Figure 3.15: Parallel scalability of the *medium* sized Rodinia benchmarks (5 repetitions).

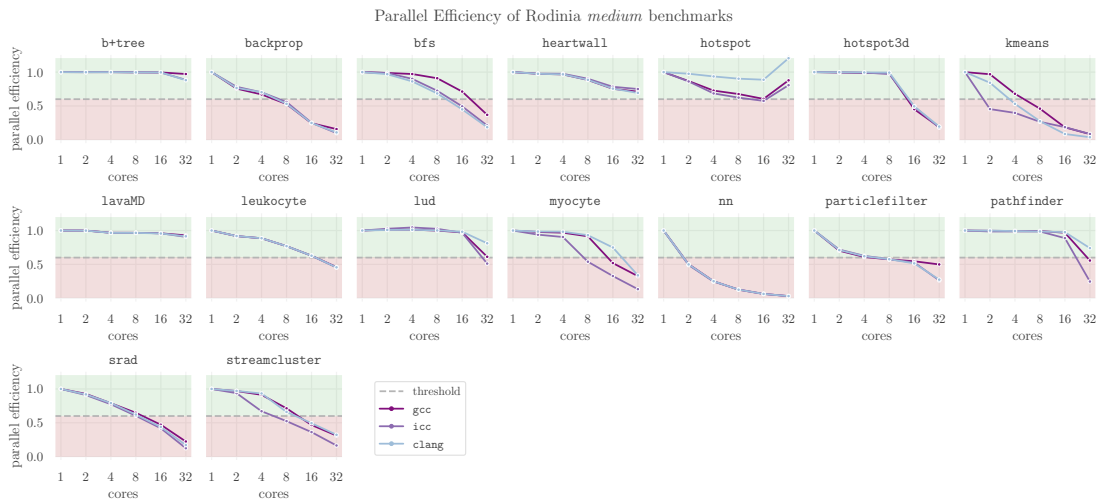


Figure 3.16: Parallel efficiency of the *medium* sized Rodinia benchmarks. The green area marks *acceptable* efficiencies over 60%, while the red area marks too low values.

have an acceptable parallel efficiency for any number of cores, but this statement does not hold for all benchmarks. Using different compilers leads to different measured times, but the compiler does not change the scaling behavior, nor the universal runtime behavior of an application.

3.3.2 Scalability of SPEC OMP2012 Benchmarks

We are also interested in the parallel scalability of the SPEC OMP2012 benchmarks. In comparison to the Rodinia experiments, we choose `gcc` as our default compiler since the compiler does not affect the behavior of applications in general.

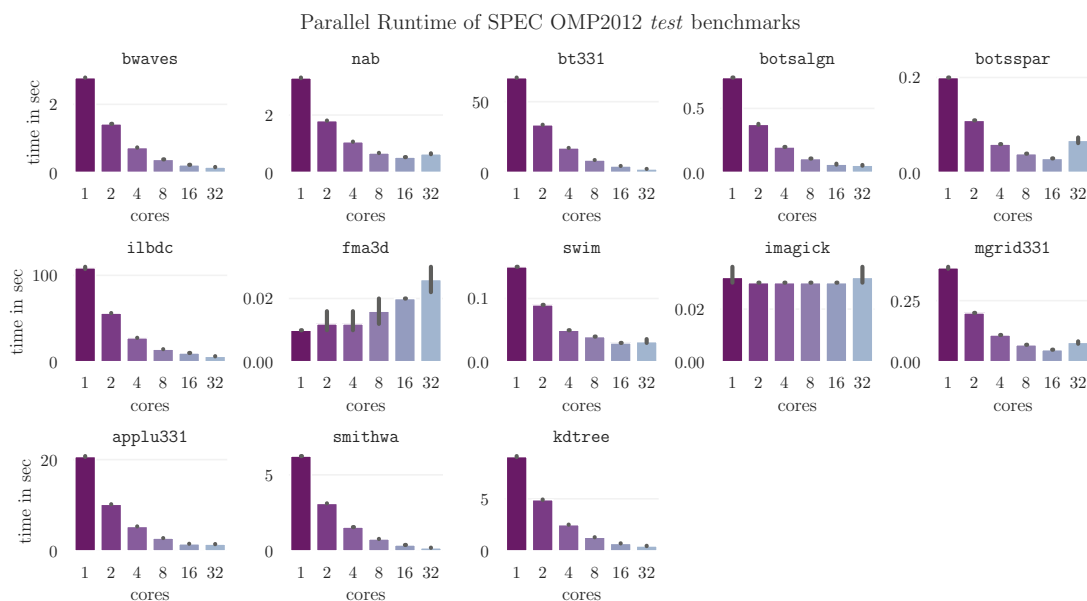


Figure 3.17: Parallel scalability of the SPEC OMP2012 benchmark suite for the *test* input size with 5 repetitions.

Figures 3.17 and 3.18 show the parallel scaling behaviors and parallel efficiencies of the SPEC OMP2012 benchmarks for the *test* input size. It seems like almost all apps scale well, with an exception of `fma3d` and `imagick`. But the execution time of these two applications is also very small, which makes time differences seem relatively large, even though there is only a small absolute difference.

Therefore, it is necessary to take a look at a bigger input size, like the *train* input size, to avoid artifacts. We see the results in Figures 3.19 and 3.20. As we can see for the parallel scalability of the *train* size, the execution time for `fma3d` and `imagick` does not lead to such artifacts as in Figure 3.17. Further, all benchmarks seem to scale fairly well.

Regarding the parallel efficiency of the *train* input size, shown in Figure 3.20, we see that several applications, e.g., `bwaves`, `bt331`, `ilbdc`, `smithwa`, `kdtree`, have an

3. CO-SCHEDULING PREPARATION: PARALLEL SCALABILITY AND SCHEDULING POTENTIAL

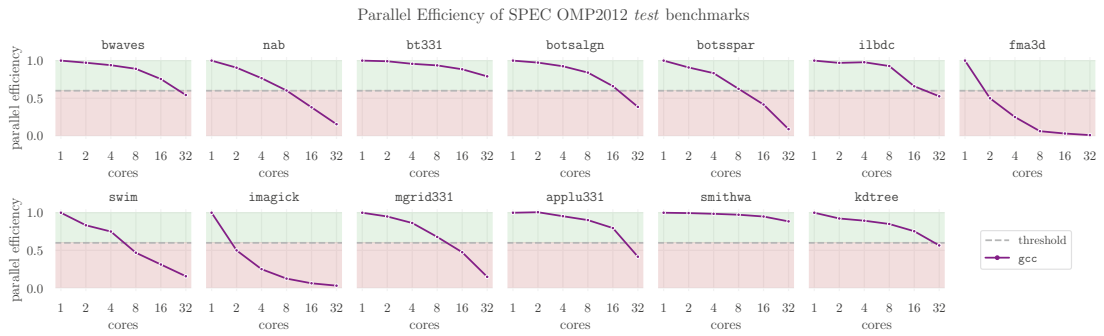


Figure 3.18: Parallel efficiency of the SPEC OMP2012 benchmark suite for the *test* input size. The green area marks *acceptable* efficiencies over 60%, while the red area marks too low values.

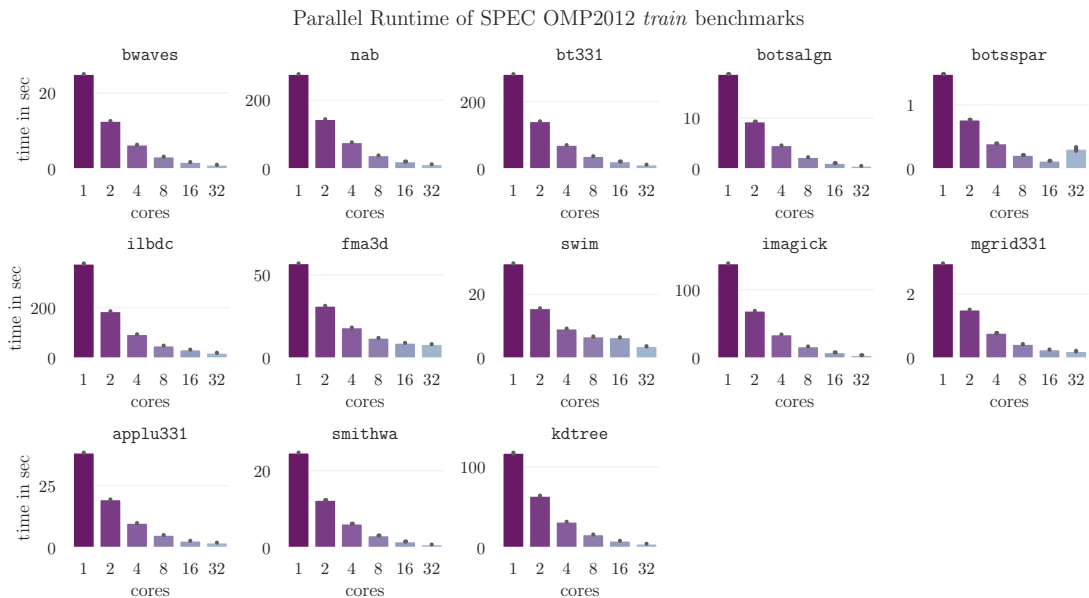


Figure 3.19: Parallel scalability of the SPEC OMP2012 benchmark suite for the *train* input size with 5 repetitions.

acceptable parallel efficiency for all cores or are closely there. Still, other benchmarks, like *imagick* or *fma3d*, do not have such a good parallel efficiency, and these applications benefit from co-scheduling.

By exploring the Rodinia and SPEC OMP2012 benchmarks, we notice that applications often do not perfectly scale, which makes it unreasonable to execute them in a fully parallel mode using all available cores of a machine. This means that we can benefit from co-scheduling regarding the parallel scalability.

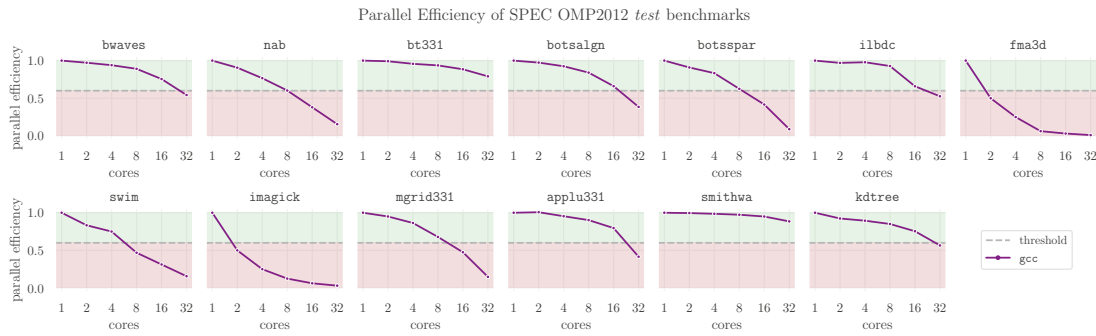


Figure 3.20: Parallel efficiency of the SPEC OMP2012 benchmark suite for the *train* input size. The green area marks *acceptable* efficiencies over 60%, while the red area marks too low values.

3.4 Performance Potential of Co-Scheduling

Next, we are interested in a possible potential of co-scheduling. Executing many applications with an execution strategy implies creating or having a schedule. To see the potential of co-scheduling, we create three schedules with the goal to minimize the overall makespan. We call these three schedules *Sequential*, *Parallel*, and *Optimization Potential*.

The SEQUENTIAL schedule assigns one core to each program. We use *hydra* with 32 cores. If there are at most 32 applications, then the applications will be executed concurrently using a single core each. Having more than 32 applications means that some programs will be assigned to the same core and executed one after another.

The PARALLEL schedule executes each program with the full number of cores, 32 on *hydra*. Thus, only one program runs concurrently scheduled on all cores.

For the OPTIMIZATION POTENTIAL, we implement an integer linear program. The assumption is that we have measured the runtime of each application for each number of used cores. There are two versions of the integer program: one only returns which job should be placed on which core/machine and how many cores should be used for the parallelization (the basic integer program), the other integer program is more sophisticated and additionally returns a concrete order, i.e., the starting times of each job. We limit the integer program to a time limit of 600 seconds. Since the more sophisticated program does not always find a solution in this time limit for big input sizes, i.e., many programs, we also use the basic IP without a concrete order. To create a concrete schedule from the basic IP, we try to find a good schedule with the smallest possible makespan by testing different orderings of jobs.

3.4.1 Definition of the Integer Programs

We assume that we know the running times for $k \in \{1, 2, 4, 8, 16, 32\}$ cores of our n jobs, which will be scheduled on m machines or cores.

Basic Integer Program (BIP) that does not return a Specific Job Order

Table 3.7 presents a description of the variables used for the basic IP. The constraints of the minimization problem are shown in Equation 3.1.

Table 3.7: Basic Integer Program (BIP) variables.

Variable	Type	Description
i	Integer	Represents the number of jobs.
j	Integer	Represents the total number of machines/cores.
k	Integer	Represents the number of cores with known runtimes.
C_{max}	Continuous	Represents the makespan of the schedule.
$jobNprocs_{i,k}$	Binary	Does job i use k cores? E.g., $jobNprocs_{1,4} = 1$ means that job 1 uses 4 cores.
$jobMachNprocs_{i,j,k}$	Binary	Is job i scheduled on machine/core j with k cores? E.g., $jobMachNprocs_{1,4,2} = 1$ means that job 1 is assigned to core 4 and will be executed with 2 cores in total. This also means that there has to be another assignment $jobMachNprocs_{1,x,2}$ representing the other core x assigned to job 1.
$timeEst_{i,k}$	Continuous	Estimated Time for job i using k cores.

Minimize Makespan C_{max}

Subject to: Each job uses either 1, 2, 4, 8, 16, or 32 cores:

$$\sum_k jobNprocs_{i,k} = 1, \forall i$$

A job using x cores has to be assigned to x machines/cores:

$$\sum_j jobMachNprocs_{i,j,k} = jobNprocs_{i,k} \times k, \forall i \forall k$$

If a job is assigned to a machine with k cores, the estimated running time of that job with k cores has to be smaller or equal to the makespan:

$$\sum_{i,k} jobMachNprocs_{i,j,k} \times timeEst_{i,k} \leq C_{max}, \forall j$$

Binary variables:

$$jobNprocs_{i,k} \in \{0, 1\}, \quad jobMachNprocs_{i,j,k} \in \{0, 1\}$$

Possible values for i, j, k :

$$i \in \{1, 2, \dots, n\}, \quad j \in \{1, 2, \dots, m\}, \quad k \in \{1, 2, 4, 8, 16, 32\}$$

(3.1)

This basic integer program returns the assignments $jobNprocs_{i,k}$ and $jobMachNprocs_{i,j,k}$, which means that we know how many cores a job should use and on which cores it should be executed. From these two variables, we cannot derive an ordering of jobs inside one machine/core. Therefore, we test six different orderings of the jobs and take the one with the smallest overall makespan. The six orderings are: the original order, the reversed order, a sorted order according to the number of used cores and this sorted order reversed, an order, where the jobs are sorted according to their assumed running time and this order reversed. Since the integer program also returns a value for the makespan, we can compare the IP's makespan with the makespan of the different orders. In our experiments, these six orders always find a very similar makespan to the IP's minimized makespan.

Advanced Integer Program (AIP) that returns a Specific Job Order

The advanced integer program solves the issue of the ordering of jobs. Thus, we need more variables, which leads to more constraints in the integer program. Therefore, this integer program cannot solve larger instances in a small time period. Table 3.8 shows the used variables of the integer program, where we already notice three relevant variables for this method: the starting time of a job $jobStime_i$, the end time of a job $jobCtime_i$, and an ordering of two jobs $jobBefJob_{i_1,i_2,j}$. Equation 3.2 shows the constraints used for this advanced integer program.

Table 3.8: Advanced Integer Program (AIP) variables.

Variable	Type	Description
i	Integer	Represents the number of jobs.
j	Integer	Represents the total number of machines/cores.
k	Integer	Represents the number of cores with known runtimes.
C_{max}	Continuous	Represents the makespan of the schedule.
$jobNproc_{i,k}$	Binary	Does job i use k cores? E.g., $jobNprocs_{1,4} = 1$ means that job 1 uses 4 cores.
$jobMach_{i,j}$	Binary	Is job i assigned to machine/core j ? E.g., $jobMach_{1,4} = 1$ means that job 1 is assigned to machine/core number 4.
$jobStime_i$	Continuous	Start time of job i .
$jobCtime_i$	Continuous	End time of job i .
$timeEst_{i,k}$	Continuous	Estimated Time for job i using k cores.
$jobBefJob_{i_1,i_2,j}$	Binary	Is job i_1 executed before job i_2 on machine/core j ? E.g., $jobBefJob_{1,9,4} = 1$ means that job 1 is executed before job 9 on machine/core number 4.
$seqTime_i$	Continuous	Sequential running time of job i , i.e., using 1 core.

Minimize Makespan C_{max}

Subject to: Each job is executed with either 1, 2, 4, 8, 16, or 32 cores:

$$\sum_k jobNproc_{i,k} = 1, \forall i$$

The assignments of jobs to machines/cores has to correspond to the number of used cores k of job i :

$$\sum_j jobMach_{i,j} = \sum_k k \times jobNproc_{i,k}, \forall i$$

The end time of a job corresponds to the starting time plus the time estimation for the used number of cores:

$$jobStime_i + \sum_k jobNproc_{i,k} \times timeEst_{i,k} = jobCtime_i, \forall i$$

All jobs must not end after the makespan:

$$jobCtime_i \leq C_{max}, \forall i$$

One and the same job cannot be ordered with itself:

$$jobBefJob_{i_1, i_2, j} = 0, \forall i_1 \in i \forall i_2 \in i \forall j, \text{ if } i_1 = i_2$$

If two jobs are executed on the same machine, one job will be executed before the other one:

$$jobBefJob_{i_1, i_2, j} + jobBefJob_{i_2, i_1, j} \geq jobMach_{i_1, j} + jobMach_{i_2, j} - 1, \quad \forall i_1 \in i \forall i_2 \in i \forall j, \text{ if } i_1 \neq i_2$$

Only one of two jobs can be executed before the other one:

$$jobBefJob_{i_1, i_2, j} + jobBefJob_{i_2, i_1, j} \leq 1, \forall i_1 \in i \forall i_2 \in i \forall j, \quad \text{if } i_1 \neq i_2$$

The ending time of a previously executed job must not be later than the start time of a following job:

$$jobCtime_{i_1} \leq jobStime_{i_2} + \sum_i seqTime_i - jobBefJob_{i_1, i_2, j} \times \sum_i seqTime_i, \forall i_1 \in i \forall i_2 \in i \forall j, \text{ if } i_1 \neq i_2$$

Binary variables:

$$jobMach_{i,j} \in \{0, 1\}, jobNproc_{i,k} \in \{0, 1\}, jobBefJob_{i_1, i_2, j} \in \{0, 1\}$$

Continuous variables:

$$jobStime_i \in \mathbb{R}, jobCtime_i \in \mathbb{R}$$

Possible values for i, j, k :

$$i \in \{1, 2, \dots, n\}, j \in \{1, 2, \dots, m\}, k \in \{1, 2, 4, 8, 16, 32\}$$

This AIP also returns the assignments $jobNproc_{i,k}$ and $jobMach_{i,j}$, i.e., how many cores a job should use and on which cores the job will be executed. But in comparison to the basic integer program, there are additional variables $jobStime_i$, $jobCtime_i$, and $jobBefJob_{i_1,i_2,j}$ that return ordering information. In our schedule implementation, we use the $jobBefJob_{i_1,i_2,j}$ variable as information about the ordering to create the schedule.

3.4.2 Evaluating the Optimization Potential of Co-Scheduling

We implement the sequential, parallel, and optimization potential strategies and submit the created schedules to our execution framework, which then executes these schedules on a `hydra` compute node. Since our execution framework is written in Python, we show a possible, but still easily understandable way to represent schedules using Python: the schedule is a dictionary with keys and values. A key corresponds to one core of a `hydra` compute node, and the value to the job assignments of that core. Since our execution framework only executes jobs if all requested resources are available, we only submit the order of jobs on a core. This means that the value to one key in the dictionary corresponds to a list of job IDs, and the list from left to right determines the job ordering. As an example of this schedule structure, we assume a compute node with four cores and five jobs, and present the parallel schedule in Listing 3.5.

Listing 3.5: Parallel schedule dictionary of a four-core machine with five jobs.

```

schedule = {
    '0': [1,2,3,4,5], # core 0
    '1': [1,2,3,4,5], # core 1
    '2': [1,2,3,4,5], # core 2
    '3': [1,2,3,4,5] # core 3
}

```

For the optimization potential, we implement the basic and advanced integer program, such that our execution framework can execute the resulting schedule. We create these four schedules SEQ, PAR, BIP, and AIP for multiple collections of jobs and execute each schedule for each application group three times on `hydra`, to get an average value for the makespan. These collections of jobs are either benchmarks from one benchmark suite with a specific input size, or a combination thereof. As a small summary of these experiments, Table 3.9 presents the mean overall makespans achieved with schedules created from the sequential, parallel, or the linear program strategies on `hydra`.

The advanced integer program does not return any schedule for both input sizes of SPEC OMP2012, and the combination of the Rodinia and SPEC OMP2012 applications within the given time limit of 600 seconds. We notice this in Table 3.9, where no average time is shown in the rows of *Rodinia and SPEC OMP2012* and *SPEC OMP2012 - Test and Train*.

We see that sometimes a simple scheduling strategy like SEQ may suffice, as for small input sizes, i.e., schedules created for a small number of applications. But having several applications to be co-scheduled, there is a great optimization potential, e.g., for *Rodinia*

Table 3.9: Overview of the schedule comparison mean makespan results (3 repetitions).

Applications Used From	SEQ	PAR	BIP	AIP
Rodinia and SPEC OMP2012	419.96 s	511.64 s	123.78 s	timeout
Rodinia - Small and Medium	52.40 s	350.17 s	52.58 s	53.80 s
Rodinia - Medium	53.31 s	278.80 s	52.31 s	54.92 s
Rodinia - Small	11.70 s	71.05 s	13.15 s	12.28 s
SPEC OMP2012 - Test and Train	380.80 s	159.06 s	80.51 s	timeout
SPEC OMP2012 - Test	113.27 s	48.89 s	16.19 s	14.83 s
SPEC OMP2012 - Train	373.09 s	113.12 s	72.62 s	111.40 s

and *SPEC OMP2012* we could achieve a more than three times faster makespan with co-scheduling compared to executing each application sequentially.

For a better visualization of the possible *optimization potential*, we plot the expected and computed schedules of the three strategies `SEQUENTIAL`, `PARALLEL`, and `OPTIMIZATION POTENTIAL`. The computed executed schedule in the plots is the execution with the smallest makespan out of our three measurements.

Figure 3.21 shows expected and real schedules for all Rodinia benchmarks, i.e., both the *small* and *medium* input sized applications. For the optimization potential, we use the advanced integer program, which returns an order for the schedule. We see the visualization of the strategies themselves: using the `SEQUENTIAL` strategy, each application is run in sequential mode, i.e., on one core out of the 32 cores on `hydra`. In comparison, the `PARALLEL` schedule executes each application on all available cores and creates a serial schedule, one application after the other one. For the `OPTIMIZATION POTENTIAL`, we do not recognize a pattern, because it is not a *basic* strategy.

The expected schedules of 31 Rodinia benchmarks is plotted in the first row of the plot in Figure 3.21, the real execution is plotted in the second row. We see that the schedule on `hydra` corresponds to our created schedule, but sometimes the time assumptions do not fit perfectly. The left column presents the schedule using the `SEQUENTIAL` scheduling, and we notice that each benchmark is assigned to one core and therefore executed in a serial mode, but concurrently with the other applications. This `SEQUENTIAL` execution leads to a makespan of 52.40 seconds. In the middle, we see the `PARALLEL` strategy, where each benchmark is executed on all 32 cores of a `hydra` compute node. This strategy leads to a high expected and real makespan, the execution on `hydra` leads to 350.17 seconds. Thus, this `PARALLEL` execution is about seven times slower than the `SEQUENTIAL` strategy. Taking a look at the `OPTIMIZATION POTENTIAL` using the AIP on the right, we do not see any logic behind the job-to-core assignments. Our integer program leads to a makespan of 52.58 seconds. We see that there is an optimization potential for co-scheduling, since the `PARALLEL` execution leads to an insufficient execution, meaning that the `PARALLEL` strategy is insufficient for the creation of a schedule.

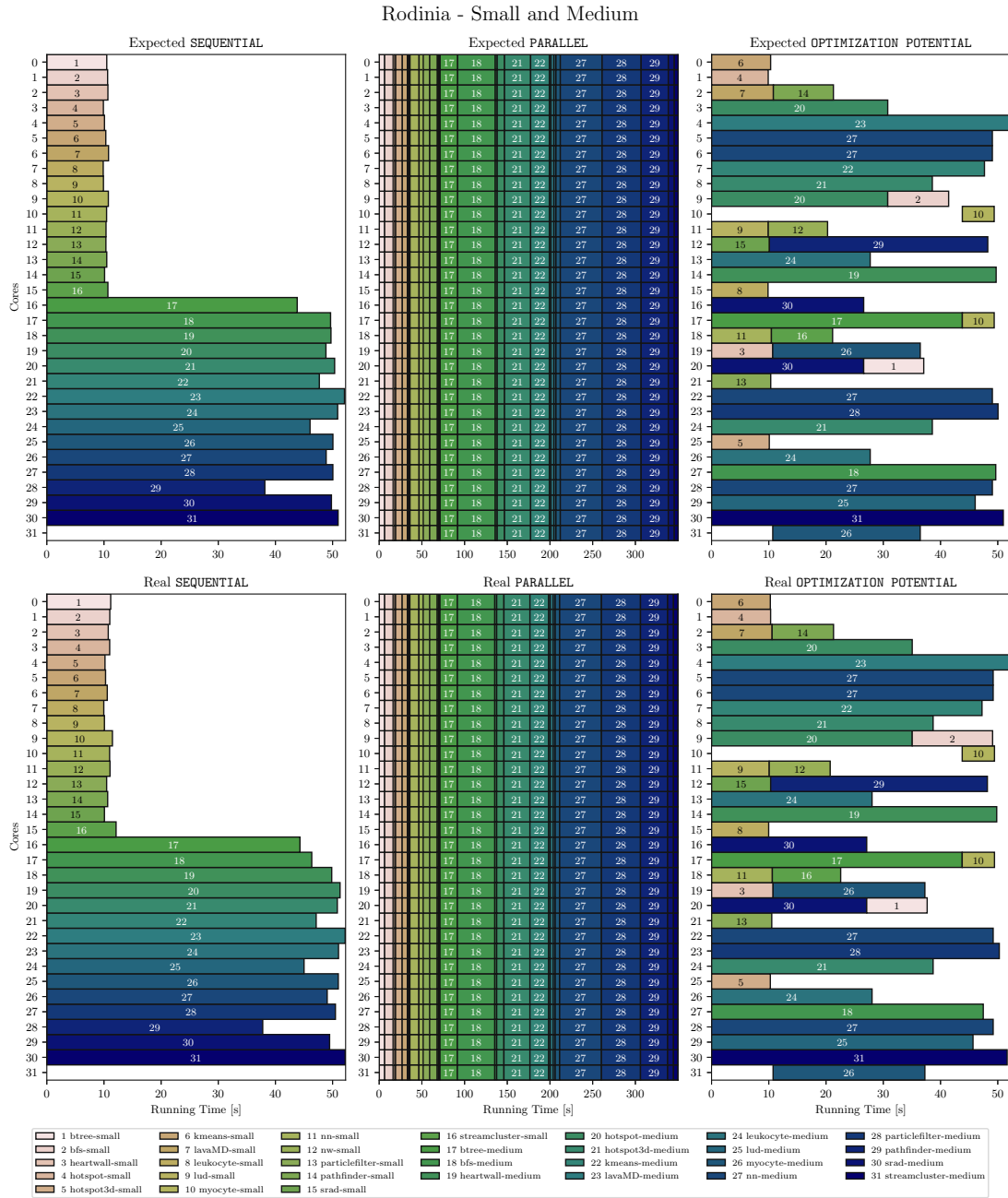


Figure 3.21: Scheduling approach comparison for applications of the Rodinia suite with a *small* and *medium* input size using the AIP (top: schedule computed by the integer program; bottom: actual schedule obtained by executing the schedule on real hardware, i.e., a hydra node).

Comparing the `SEQUENTIAL` and `OPTIMIZATION POTENTIAL`, the makespan is very similar, and for the execution of these 31 Rodinia benchmarks we can even say that a basic sequential allocation strategy suffices.

We also take a look at the scheduling strategy comparison using 57 Rodinia and SPEC OMP2012 benchmarks in Figure 3.22. For the optimization potential, we use the `BIP`, since the `AIP` does not find a solution in the given time limit.

Figure 3.22 shows the expected schedules in the first row, and the real execution in the second row of the plot. Taking a look at the left column, the `SEQUENTIAL` strategy, we now see that each application is executed on a single core. Since this is a basic scheduling approach, we do not look at how long an application takes and do not make a scheduling decision based on such observations. Therefore, we take the jobs in their order. There are three jobs which have a very long sequential running time: job 46 (`nab` with the *train* size), job 47 (`bt331` with the *train* size), and job 50 (`ilbdc` with the *train* size). These long sequential executions lead to the big makespan of 419.96 seconds. Executing all applications in a `PARALLEL` mode also leads to a big makespan of 511.64 seconds. Comparing these observations to the schedules created for the Rodinia benchmarks in Figure 3.21, we notice that the runtime behavior of the applications determines whether a basic co-scheduling is sufficient or not. For the combination of Rodinia and SPEC OMP2012 benchmarks, these basic approaches are insufficient. A clever scheduling approach, as shown by our `OPTIMIZATION POTENTIAL`, can improve the makespan significantly. Our integer program results in a makespan of 123.78 seconds, which means that this advanced co-scheduling improves the makespan by almost a factor of five.

We summarize our main observations:

1. The arithmetic intensity itself does not imply resource sharing conflicts of an application with itself.
2. Even though the kernel time differs for different compilers, neither the runtime behavior, nor the parallel scalability change by using another compiler.
3. Applications do not have a perfect scalability, i.e., it is possible to find an optimal number of cores to be used and it is reasonable to choose less cores than all available cores for a parallel execution.
4. An advanced scheduling algorithm leads to an overall better makespan compared to basic scheduling strategies, like a sequential or parallel schedule.

Rodinia and SPEC OMP2012

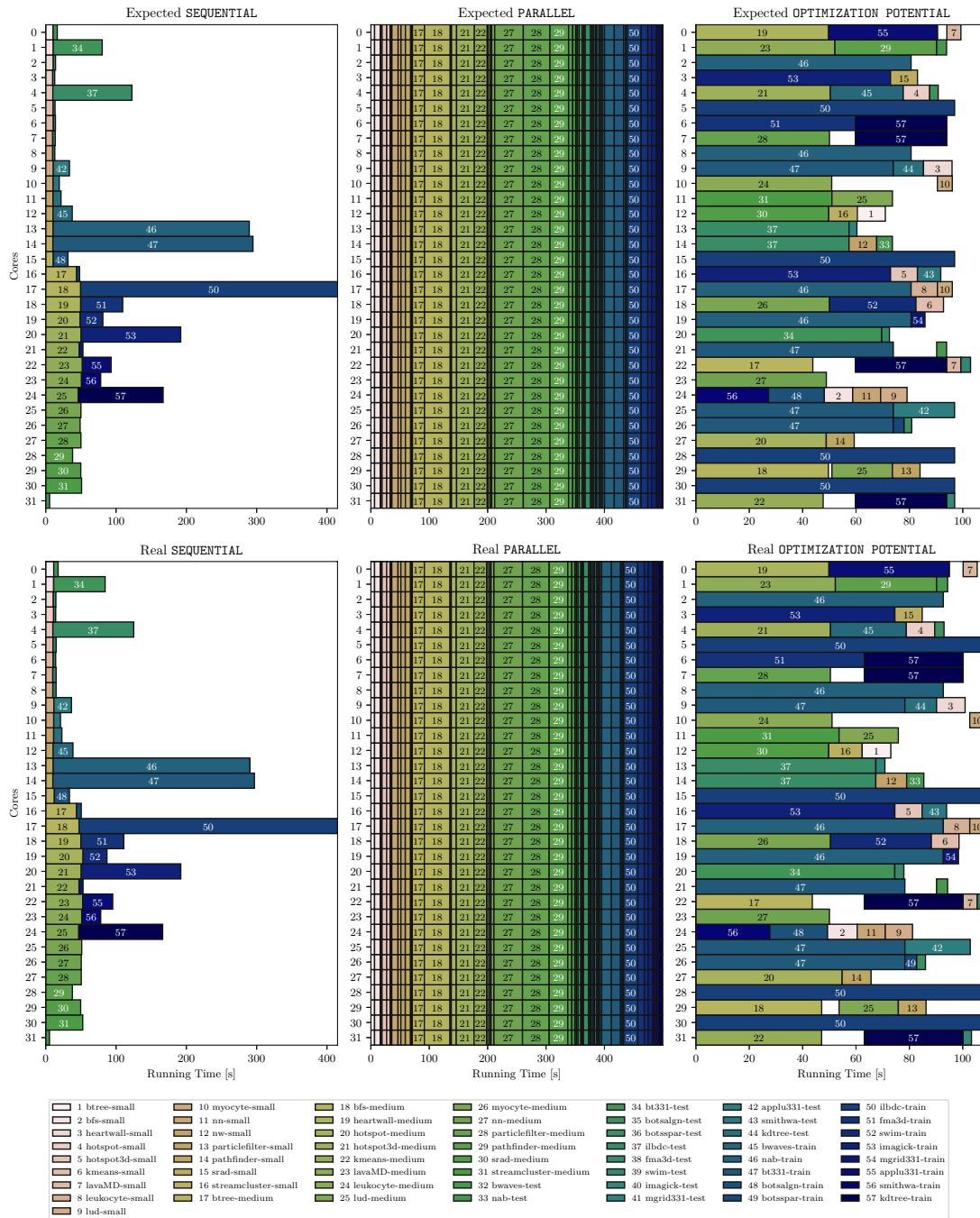


Figure 3.22: Scheduling approach comparison for applications of both the Rodinia and SPEC OMP2012 suite for all provided input sizes using the BIP (top: schedule computed by the integer program; bottom: actual schedule obtained by executing the schedule on real hardware, i.e., a hydra node).



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Characterizing Co-Scheduled Applications With Hardware Performance Counters

This chapter is led by the following motivation questions:

- How can we efficiently co-schedule two applications on one multicore machine?
How can we share resources on a multicore machine?
- Are the running times affected if we schedule two applications on two distinct sockets or NUMA domains of the multicore machine?
How do two applications interfere with each other if we schedule them on the same socket or on the same NUMA domain?

We start by creating configurations or strategies to co-schedule two applications. There are different possibilities to co-schedule two applications on the two sockets with 32 cores in total on hydra. The main goal is to discover how we can co-schedule two applications on these available resources and which configurations do or do not lead to increased kernel times with co-scheduling. If we see increased co-scheduled kernel times, we imply some resource sharing conflicts. We are interested to explore such cases.

Then, we will see a problem occurring with measuring kernel times of co-scheduled applications: kernel sections of two independent programs do not start and stop at the same time, which makes it difficult to measure only the time needed in the kernel section. After presenting and discussing the problem, we propose a solution: a library for synchronizing programs.

Using this synchronization method for the kernel section of two programs, we measure the influence of co-scheduling with the different configurations for co-scheduling two

applications. This demonstrates which configuration of concurrent executions of two applications may lead to resource sharing conflicts.

4.1 Configurations for Measuring the Influence of Co-Scheduling

At first, we define possible co-scheduling strategies that can be used to explore runtime behaviors of co-scheduled applications. To find out whether there are conflicts or not, we co-schedule two applications A and B. We measure the dedicated kernel time of A and the kernel time of A while B is executed concurrently with A, namely the co-scheduled time of A.

As we have already seen in Figure 2.3, `hydra` has two sockets with 16 cores on each socket. With this prerequisite, we define three configurations for measuring the influence of co-scheduling:

- Sharing a compute node by assigning the separate sockets to the applications.
- Sharing one socket by assigning half the cores to each application.
- Sharing both sockets by giving each application the same number of cores on each socket.

These strategies are depicted in Figures 4.1 – 4.3.

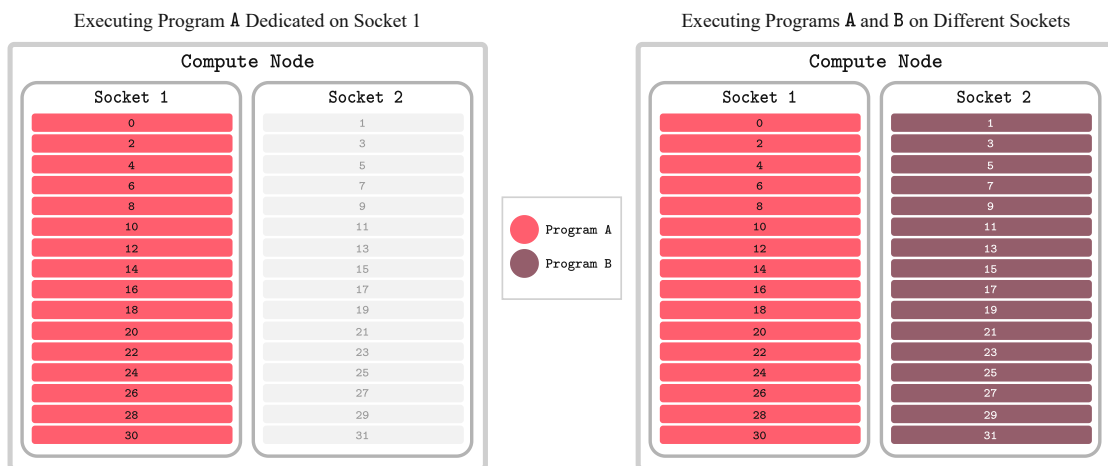


Figure 4.1: Visualization of the Different Sockets strategy.

As we see in Figure 4.1, sharing a compute node for co-scheduling means assigning *different sockets* to each application. Each application is assigned to a socket, and



Figure 4.2: Visualization of the Compact Socket 1 strategy.

therefore to a NUMA domain. Our hypothesis is that the co-scheduled time of A with B does not differ greatly from the dedicated time of A.

If two programs share a socket, share a NUMA domain, as shown in Figure 4.2, we call this a *compact* thread mapping. Sharing a NUMA domain means sharing compute resources of this one socket, which could lead to time penalties. Therefore, our hypothesis is that there are differences between a dedicated and co-scheduled execution of A using this *compact* scheduling strategy.

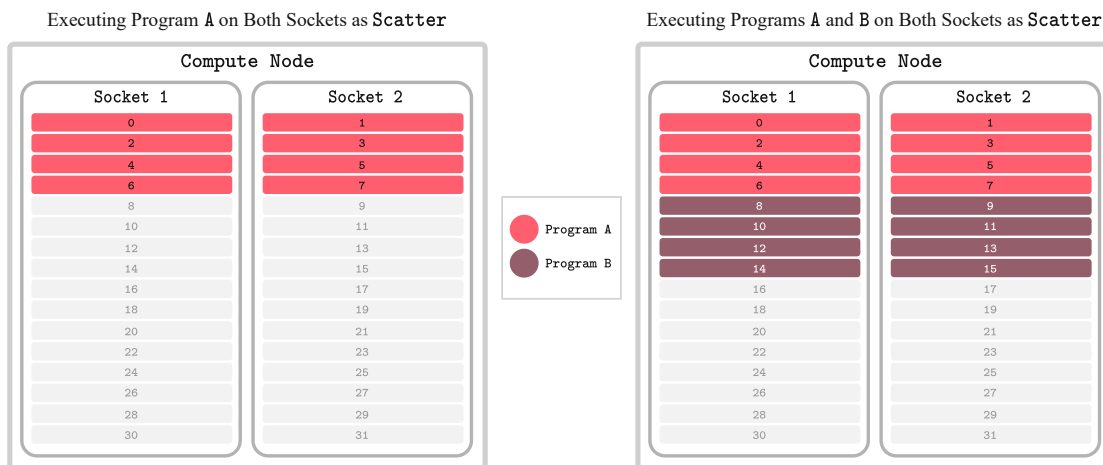


Figure 4.3: Visualization of the Scatter Socket 1+2 strategy.

The last configuration is shown in Figure 4.3. We see two applications sharing both sockets when being co-scheduled. Since we use cores of all sockets, in our case two sockets, this strategy behaves like a *scatter* thread mapping. This case is interesting

to observe since already the dedicated execution of A should have some overhead due to the communication between the two sockets. For the co-scheduling we assume time differences between the dedicated and co-scheduled execution of A, as already for the *compact* strategy, since both applications share the same resources, now on both sockets.

4.2 The Problem of Measuring Co-scheduled Applications

In the previous section, we discussed measuring the *dedicated kernel time of A* and the *kernel time of A while being co-scheduled with B*. This sentence already triggers the problem of measuring co-scheduled applications:

- How can we measure the kernel time of A while A is being co-scheduled with B?
- How can we co-schedule the kernel sections of A and B?

The problem is depicted in Figure 4.4 as `No Sync`. We see that the kernel sections of two concurrently executed applications might not start and end at the same time. Therefore, while one kernel section already starts, the other application might do some pre- or postprocessing concurrently. This behavior is problematic, since the time needed for preprocessing or postprocessing steps often fluctuates. As a result, the measured kernel time results are not deterministic and often depend on external factors that have nothing to do with the parallel computing part. As we can see, it is problematic to measure the kernel time of two concurrently executed applications, and further not possible to measure the co-scheduled time of kernel sections without changing the source code of the applications.

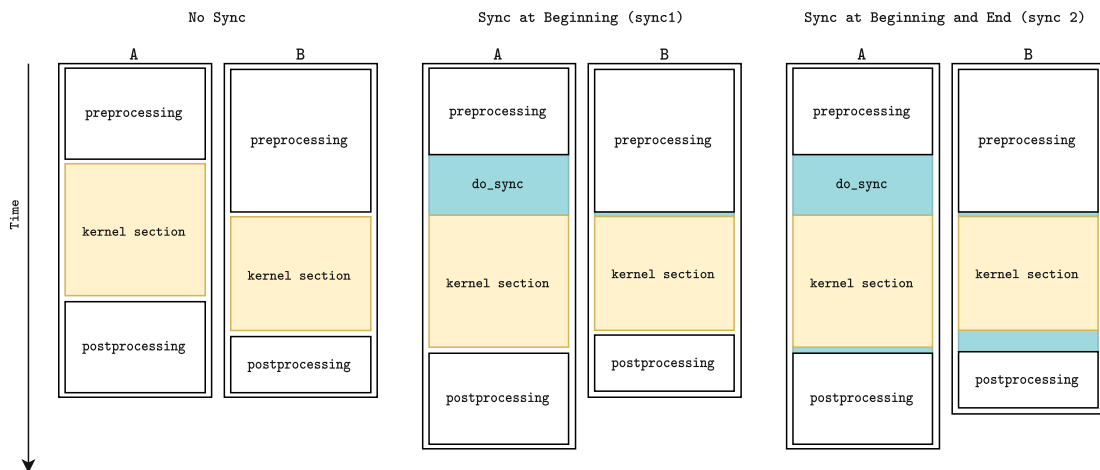


Figure 4.4: Synchronization possibilities of two programs A and B. The kernel section is the area of interest.

4.2.1 Synchronization of Programs as the Solution

Therefore, we need a mechanism for synchronizing co-located applications, before they enter their kernel section. As a solution, we propose a synchronizer `sync`, a static C and C++ library that synchronizes a defined number of programs.

Figure 4.4 visualizes the synchronization as the solution. In this example, we have two applications that are co-located. We assume that each program has some kind of preprocessing (e.g., reading in a file, allocating memory, etc.), a kernel section, and a postprocessing part (e.g., writing results to an output file, deallocating memory, etc.).

Without synchronization (`No Sync`), the two programs are run the way they are. As we can see, the preprocessing time of these two applications varies, which leads to one kernel section already starting while the other application still does its preprocessing. Therefore, we can call the synchronizer after the preprocessing by adding the call `do_sync` after each program's preprocessing part (`sync 1`). Then, both kernel sections start concurrently and if both kernel sections take equally long, we can guarantee a non-distortion of co-scheduled kernel section measurements. But it is still possible and very likely that two kernel sections are not identical and therefore have different running/kernel times, whereas we need a second synchronization call `do_sync` of both programs after their kernel section (`sync 2`). With these two synchronization calls for applications with a preprocessing, kernel section, and postprocessing part, we can assure non-distortion of measurements regarding the kernel section.

The synchronization library (published on GitLab¹) can synchronize as many applications as needed. An environment variable `NUM_SYNC_PROGRAMS` is responsible for communicating the number of applications to be synchronized and therefore needs to be set correspondingly, as shown in Listing 4.1.

Listing 4.1: Setting the environment variable `NUM_SYNC_PROGRAMS`.

```
# synchronize 2 programs
# NOTICE: number written in quotes
export NUM_SYNC_PROGRAMS="2"
```

Each application participating in the synchronization process needs to call the functions for initializing the synchronizer, doing the synchronization, and cleaning up the synchronizer: `init_sync()`, `do_sync()`, and `cleanup_sync()`. The synchronization call `do_sync()` can be written at any program point where a synchronization should happen. A program can contain several `do_sync()` calls. Internally, we use semaphores that allow implicit waiting until all participating programs reach their synchronization point.

Overhead of the Synchronization Library

An important question is if the usage of the synchronizer creates an overhead. The assumption is that there is almost no noticeable overhead in the runtime, as the `sync`

¹<https://gitlab.com/bsarkoez/program-synchronizer>

library is a very lightweight library. To analyze the overhead, we compare the median kernel times of sequentially executing the Rodinia benchmarks on one core. We only *synchronize* this one application with itself, i.e., wrapping synchronizer code around the kernel section (`init_sync()`, `do_sync()`, `cleanup_sync()`), and setting the environment variable for the number of programs as `NUM_SYNC_PROGRAMS="1"`. Then, we compare the kernel time without synchronizing code and the time for the kernel section wrapped with synchronizer elements around it.

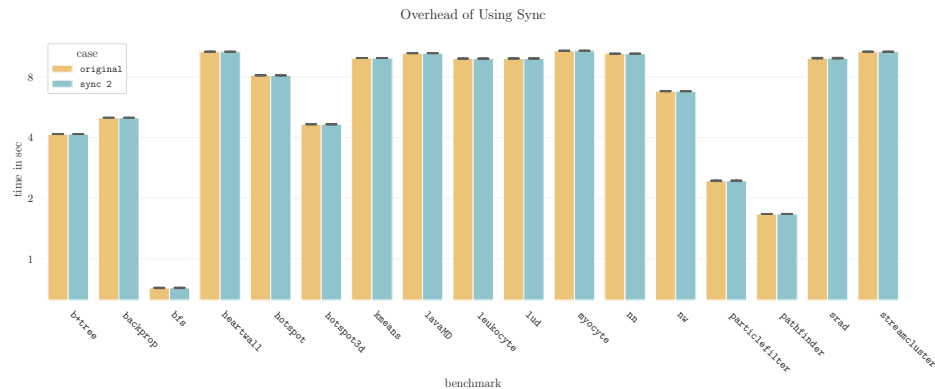


Figure 4.5: Overhead of using synchronization for the *small* Rodinia benchmarks.

Figure 4.5 shows the difference of these two differently measured times, and we notice no obvious time difference in the plot. In fact, there is a small time difference of approximately 0.2 ms, which is minimal and supports the hypothesis of an imperceptible overhead.

Relevance of Synchronizing Kernel Sections

Furthermore, we want to show the relevance of using a synchronizer for synchronizing the kernel sections of concurrently executed applications. For this, we need to compare the concurrent kernel time of applications without synchronization with the co-scheduled kernel time of synchronized applications. It does not matter which synchronization method (no sync, sync 1, sync 2) yields the smallest running time. To show the relevance of using sync for concurrent executions, it is sufficient to show that there are time differences between running the programs with and without synchronization.

We take the first memory- and compute-bound benchmarks from the Rodinia benchmark suite ranked in Table 3.3: `backprop` and `myocyte`. Then, we use these two applications, `backprop` and `myocyte`, as our program A, while all other programs of the benchmark suite represent program B, and we co-schedule A with B as shown in Figures 4.1 – 4.3. This means, we co-schedule A and B, where both programs use a *whole socket*, share a socket with a *compact mapping*, or share both sockets using a *scatter mapping*. Each of these co-scheduling configurations is executed without synchronization of A and B (no sync), with synchronization before the kernel sections of A and B (sync 1), and with synchronization before and after their kernel sections (sync 2).

The *compact mapping* leads to similar results as using a *whole socket*. Therefore, we only take a closer look at the time differences between no synchronization and synchronization for the *whole socket* and *scatter mapping* strategies.

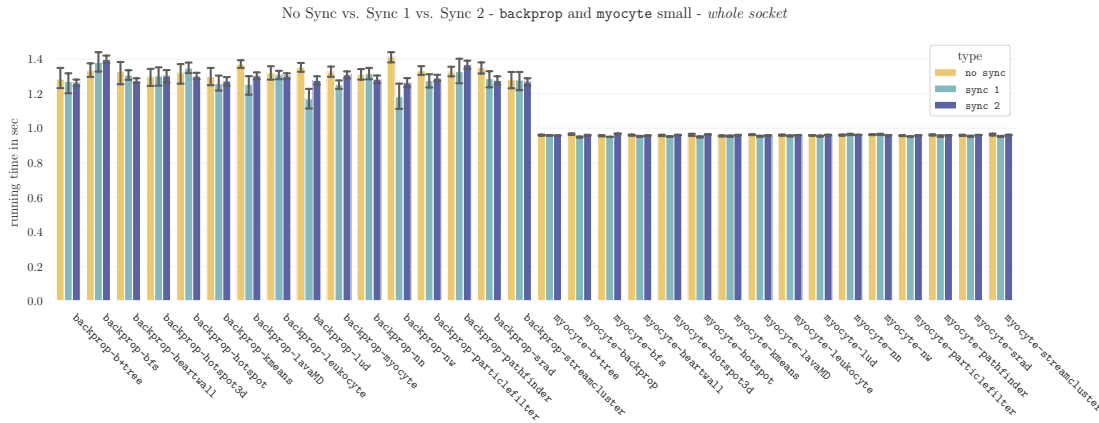


Figure 4.6: Differences in the concurrent execution time with and without synchronization for the *small* input size of backprop and myocyte assigned to a socket (9 repetitions for no sync, 9 repetitions for sync 1, 30 repetitions for sync 2).

Figure 4.6 shows the time difference between using no synchronization and using synchronizations when two applications are executed on a whole socket, i.e., on 16 cores each. As we can see, for backprop there are small differences, but for myocyte, the synchronization does not seem to change the runtime.

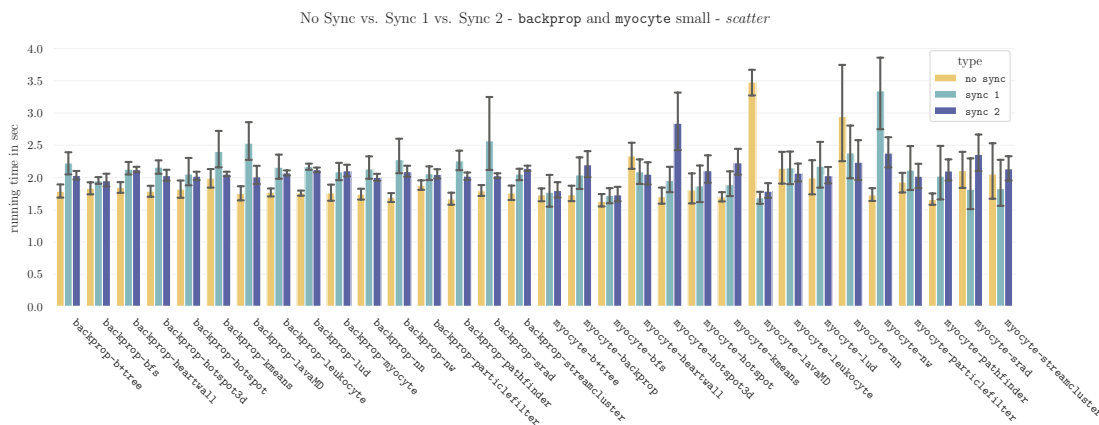


Figure 4.7: Differences in the concurrent execution time with and without synchronization for the *small* input size of backprop and myocyte using a scatter affinity mapping (9 repetitions for no sync, 9 repetitions for sync 1, 30 repetitions for sync 2).

Looking at Figure 4.7, we see executions of two programs mapped to two sockets with a scatter affinity mapping, where each application is executed on eight cores. In this

plot, we notice time differences between no synchronization and synchronization for all benchmark executions.

Regarding the relevance of `sync`, both Figures 4.6 and 4.7 show, that it is possible to achieve other running times when using synchronization. Since the motivation of the synchronizer is measuring only concurrently executed kernel sections without pre- and postprocessing steps, we now see that there is indeed a time difference, which suggests using a synchronization for comparing kernel sections.

Another motivating reason for using `sync` is getting more reasonable and meaningful performance counter measurements, which should help characterizing phenomena seen in concurrent executions. Therefore, it is interesting to see if there are indeed differences regarding performance counter values with and without synchronization.

Hence, we pick two concurrent executions: `hotspot-kmeans (small)` and `hotspot-lud (small)` with the `scatter` strategy and compare performance values measured with `likwid` of synchronized and non-synchronized kernel sections. Table 4.1 shows the performance counter differences of the concurrent execution of `hotspot` and `kmeans` while being synchronized and not synchronized. We see some differences, e.g., L1 DTLB load misses or L3 bandwidth.

Table 4.1: No Sync vs. Sync 1 vs. Sync 2 for a `scatter` co-location of `hotspot` and `kmeans`.

Counter	No Sync	Sync 1	Sync 2	No Sync vs. Sync 2
L1 DTLB load misses STAT	123271	149614	70680	-42.66%
L1 DTLB store misses STAT	142685	222986	88438	-38.02%
L1 ITLB misses STAT	29249	113854	41228	+40.96%
L2 bandwidth [MBytes/s] STAT	928490.23	1144265.00	1413717.00	+52.26%
L2 data volume [GBytes] STAT	1221.25	1475.98	1688.02	+38.22%
L2 miss rate STAT	0.12	0.14	0.14	+18.91%
L2 miss ratio STAT	0.92	1.10	1.02	+10.26%
L2 request rate STAT	0.76	0.89	0.97	+26.56%
L3 bandwidth [MBytes/s] STAT	485640.38	704378.30	303925.26	-37.42%
L3 data volume [GBytes] STAT	636.25	845.23	546.06	-14.18%
L3 miss rate STAT	0.00	0.00	0.00	+179.35%
L3 miss ratio STAT	0.05	0.14	0.08	+50.00%
L3 request rate STAT	0.00	0.01	0.01	+38.64%

Looking at the differences in performance counters at the co-location of `hotspot` and `lud`, Table 4.2 also shows differences, but on different performance counters, e.g., L1 ITLB misses.

Table 4.2: No Sync vs. Sync 1 vs. Sync 2 for a scatter co-location of hotspot and lud.

Counter	No Sync	Sync 1	Sync 2	No Sync vs. Sync 2
L1 DTLB load misses STAT	115305	95726	98054	-14.96%
L1 DTLB store misses STAT	147263	67558	139440	-5.31%
L1 ITLB misses STAT	82520	35294	44128	-46.52%
L2 bandwidth [MBytes/s] STAT	949651.38	1412271.00	1404189.00	+47.86%
L2 data volume [GBytes] STAT	1258.93	1685.59	1688.62	+34.13%
L2 miss rate STAT	0.12	0.09	0.14	+18.51%
L2 miss ratio STAT	0.91	0.77	1.03	+12.16%
L2 request rate STAT	0.76	0.60	0.95	+24.10%
L3 bandwidth [MBytes/s] STAT	473628.44	246494.81	700631.77	+47.93%
L3 data volume [GBytes] STAT	624.58	559.27	826.34	+32.30%
L3 miss rate STAT	0.00	0.00	0.00	+0.00%
L3 miss ratio STAT	0.14	0.13	0.09	-31.79%
L3 request rate STAT	0.00	0.01	0.01	+62.50%

Although it is not important now to investigate, which performance counter differs significantly with and without synchronization, the fact that there are differences is very relevant. We see that synchronizing the kernel section of co-located applications can improve the quality of the measurements.

Therefore, we use the synchronization mechanism `sync 2` for all subsequent co-scheduling experiments to achieve results explicitly showing observations of the co-scheduling of kernel sections.







4.3 The Influence of Co-Scheduling with Different Scheduling Configurations

We are interested if our defined co-scheduling configurations from Figures 4.1 – 4.3 lead to time differences between a dedicated and co-scheduled execution, i.e., have an influence on co-scheduling. Since we synchronize the kernel sections of applications with `sync 2`, we need to modify the source code of the benchmarks. Therefore, we stick with the Rodinia benchmark suite, since the SPEC OMP2012 suite is stricter regarding changes in the source code. Additionally, Rodinia does not have fixed input sizes, which allows the definition of fine granularity input sizes.

For these experiments, we choose three Rodinia benchmarks to represent our program A, the program for which we measure differences between a dedicated and co-scheduled kernel time: `backprop` (memory-bound), `hotspot` (compute-bound), and `myocyte` (compute-

bound). Since these benchmarks had the biggest time difference between a pure sequential and concurrent execution in Tables 3.3 and 3.4, they seem to lead to resource sharing conflicts when being co-scheduled. Accordingly, we are interested in exploring, which scheduling configuration leads to such conflicts and which configuration avoids them.

Table 4.3: Overview of the socket test notations and icons.

Notation	Description
 $A@S1$	Runtime of A when executing A dedicated on socket 1 (Figure 4.1).
 $A@S1 \mid B@S2$	Runtime of A when executing A and B on different sockets (Figure 4.1).
 $A@1/2@S1$	Runtime of A when executing A on half of socket 1 dedicated (Figure 4.2).
 $A@1/2@S1 \mid B@1/2@S1$	Runtime of A when executing A and B on the same socket compact (Figure 4.2).
 $A@1/4@S1+S2$	Runtime of A when executing A on both sockets dedicated scatter (Figure 4.3).
 $A@1/4@S1+S2 \mid B@1/4@S1+S2$	Runtime of A when executing A and B on both sockets scatter (Figure 4.3).

For the presentation and discussion of the results, we use the notations given in Table 4.3.

Sharing a compute node by assigning separate sockets to the two applications A and B is depicted in Figure 4.1. We say that application A is executed on socket 1 ($A@S1$) and application B on socket 2 ($B@S2$). This constellation compares time differences that can occur by sharing a compute node. Our hypothesis is that this leads to no runtime differences between a dedicated and co-scheduled execution of A, since both applications are executed on different compute resources, namely the sockets, which do not share resources with one another.









Sharing one socket, where each application gets half the number of cores of one socket, i.e., eight cores, is depicted in Figure 4.2. We say that A is executed on half the number of cores of socket 1 ($A@1/2S1$) and B on the other half of socket 1 ($B@1/2S1$). As we can see, the granularity of the resource sharing got smaller by comparing runtime differences when sharing the same socket. This sharing of one socket also corresponds to sharing the same NUMA domain on *hydra*. Our hypothesis is that there are resource sharing conflicts if both A and B are executed concurrently on the same socket. This can be explained by the fact that we choose applications with a much higher concurrent running time than a sequential running time as our program A, which is an indication that this program A allocates many resources.

The third introduced resource-sharing possibility, sharing both sockets with a *scatter* mapping, is depicted in Figure 4.3. We therefore say that A is executed on a quarter of cores of both sockets 1 and 2 ($A@^{1/4}S1+S2$), and the same applies to B ($B@^{1/4}S1+S2$). Since both programs A and B share the same resources, i.e., the same sockets, our hypothesis is that there is a difference between the execution time of executing program A dedicated in a *scatter* way and the execution time of the concurrent execution of A and B.

4.3.1 Co-Scheduling Experiments

We use the Rodinia benchmarks *backprop*, *hotspot*, and *myocyte* as program A and compare the dedicated kernel time of A with the co-scheduled kernel time of A. Each co-scheduled run of any A with B is executed 30 times and we present the median kernel time achieved.

Table 4.4: Influence of co-scheduling for *backprop* by sharing a compute node [A@S1 | B@S2] (30 repetitions for the dedicated *backprop* and all co-scheduled executions).

	small			medium		
	 	 	Ratio	 	 	Ratio
backprop - b+tree	1.33 s	1.27 s	95.85%	4.35 s	4.24 s	97.36%
backprop - bfs	1.33 s	1.40 s	105.28%	4.35 s	4.34 s	99.89%
backprop - heartwall	1.33 s	1.28 s	96.60%	4.35 s	4.31 s	99.20%
backprop - hotspot3d	1.33 s	1.31 s	99.25%	4.35 s	4.12 s	94.83%
backprop - hotspot	1.33 s	1.31 s	98.49%	4.35 s	4.10 s	94.25%
backprop - kmeans	1.33 s	1.28 s	96.60%	4.35 s	4.21 s	96.67%
backprop - lavaMD	1.33 s	1.31 s	98.49%	4.35 s	4.34 s	99.77%
backprop - leukocyte	1.33 s	1.31 s	98.87%	4.35 s	4.17 s	95.75%
backprop - lud	1.33 s	1.27 s	96.23%	4.35 s	4.11 s	94.48%
backprop - myocyte	1.33 s	1.31 s	98.49%	4.35 s	3.91 s	89.77%
backprop - nn	1.33 s	1.29 s	96.98%	4.35 s	4.25 s	97.70%
backprop - particlefilter	1.33 s	1.29 s	97.74%	4.35 s	4.21 s	96.78%
backprop - pathfinder	1.33 s	1.36 s	102.64%	4.35 s	4.17 s	95.75%
backprop - srad	1.33 s	1.28 s	96.60%	4.35 s	4.26 s	98.05%
backprop - streamcluster	1.33 s	1.27 s	96.23%	4.35 s	4.22 s	97.01%
min	1.14 s	0.97 s	85.09%	3.80 s	3.35 s	88.16%
mean	1.32 s	1.30 s	98.19%	4.29 s	4.16 s	97.11%
median	1.33 s	1.30 s	98.11%	4.35 s	4.19 s	96.32%
max	1.47 s	1.54 s	104.76%	4.63 s	5.38 s	116.20%

Before discussing the influence of co-scheduling, we take a look at Table 4.4 as an example. We see the mean value of the 30 dedicated and co-scheduled kernel times of *backprop*,

where `backprop` and any other benchmark is executed on a separate socket, i.e., the $A@S1 \mid B@S2$ co-scheduling configuration. The ratio shown refers to the ratio between the mean kernel time of the co-scheduled and dedicated run. If the ration x/y of two numbers x and y is 100%, then x and y are the same number. This means that if these two numbers differ greatly, the ratio would be significantly different from 100%. As the results from Table 4.4 show, there is no significant ratio, i.e., a ratio $\geq 110\%$. We are interested in significant ratios and therefore exclude insignificant ratios, i.e., ratios $< 110\%$, from the following result presentations.

backprop - MB

We use `backprop` as our program A and compare the dedicated and co-scheduled kernel times of the three defined co-scheduling strategies.

The results in Table 4.5 show that there are no significant ratios for $A@S1$ and $A@S1 \mid B@S2$, nor for $A@1/2S1$ and $A@1/2S1 \mid B@1/2S1$. Since $A@S1$ and $A@S1 \mid B@S2$ lead to very similar results, our hypothesis is strengthened: executing applications on different sockets helps avoiding resource sharing problems, since both applications do not share resources this way.













Additionally, as the results from Table 4.4 show, co-scheduling even seems to improve the dedicated kernel time, which is somehow contradictory. It seems that resource sharing, even if it is only resources of one compute node, may lead to faster kernel times compared to no resource sharing at all. Still, we see this phenomenon in several experiments.

Interestingly, there is significant ratio comparing $A@1/2S1$ and $A@1/2S1 \mid B@1/2S1$, even though both A and B are executed on the same socket, and therefore share resources, since the cores inside of one socket do not have separate resources available.

Then, we look at the ratio of the dedicated $A@1/4S1+S2$ and the co-scheduled $A@1/4S1+S2 \mid B@1/4S1+S2$ execution and notice relevant relative time differences for almost all execution combinations. If we compare the mean or median kernel time of the dedicated *compact mapping* ($A@1/2S1$) with the mean or median kernel time of the dedicated *scatter mapping* ($A@1/4S1+S2$), we already notice a difference, even though both strategies use eight cores. The question is: what resources are shared that lead to these kernel time differences? Figure 2.3 helps answering this question: since $A@1/4S1+S2$ uses both sockets, and $A@1/2S1$ only uses one socket, there is an additional communication overhead for $A@1/4S1+S2$. This communication across the two sockets might be a bottleneck, as the increased kernel times of $A@1/4S1+S2 \mid B@1/4S1+S2$ indicate.

4.3. The Influence of Co-Scheduling with Different Scheduling Configurations













Table 4.5: Influence of co-scheduling with backprop as program A (30 repetitions).

	small			medium		
			Rel. Diff.			Rel. Diff.
min	1.14 s	0.97 s	85.09%	3.80 s	3.35 s	88.16%
mean	1.32 s	1.30 s	98.19%	4.29 s	4.16 s	97.11%
median	1.33 s	1.30 s	98.11%	4.35 s	4.19 s	96.32%
max	1.47 s	1.54 s	104.76%	4.63 s	5.38 s	116.20%
			Rel. Diff.			Rel. Diff.
min	1.15 s	1.13 s	98.26%	4.12 s	3.89 s	94.42%
mean	1.17 s	1.19 s	102.31%	4.22 s	4.30 s	101.92%
median	1.17 s	1.17 s	100.00%	4.22 s	4.25 s	100.71%
max	1.19 s	1.57 s	131.93%	4.26 s	5.16 s	121.13%
			Rel. Diff.			Rel. Diff.
backprop - b+tree	1.72 s	1.98 s	115.12%	5.61 s	6.40 s	114.18%
backprop - bfs	1.72 s	1.90 s	110.76%	5.61 s	7.21 s	128.55%
backprop - heartwall	1.72 s	2.12 s	123.55%	5.61 s	6.86 s	122.39%
backprop - hotspot3d	1.72 s	1.97 s	114.53%	5.61 s	7.71 s	137.64%
backprop - hotspot	1.72 s	2.02 s	117.73%	5.61 s	7.37 s	131.49%
backprop - kmeans	1.72 s	2.02 s	117.44%	5.61 s	7.06 s	125.96%
backprop - lavaMD	1.72 s	1.90 s	110.76%	5.61 s	7.70 s	137.38%
backprop - leukocyte	1.72 s	2.07 s	120.35%	5.61 s	7.23 s	128.99%
backprop - lud	1.72 s	2.12 s	123.55%	5.61 s	7.63 s	136.22%
backprop - myocyte	1.72 s	2.15 s	124.71%	5.61 s	7.54 s	134.61%
backprop - nn	1.72 s	2.02 s	117.73%	5.61 s	6.86 s	122.30%
backprop - particlefilter	1.72 s	2.02 s	117.44%	5.61 s	7.13 s	127.30%
backprop - pathfinder	1.72 s	2.03 s	118.02%	5.61 s	7.17 s	127.83%
backprop - srاد	1.72 s	2.00 s	116.28%	5.61 s	7.29 s	129.97%
backprop - streamcluster	1.72 s	2.16 s	125.58%	5.61 s	7.55 s	134.70%
min	1.47 s	1.51 s	102.72%	5.16 s	5.08 s	98.45%
mean	1.73 s	2.05 s	118.92%	5.63 s	7.59 s	134.71%
median	1.72 s	2.03 s	118.02%	5.61 s	7.25 s	129.35%
max	2.04 s	4.10 s	200.98%	6.62 s	14.96 s	225.98%

hotspot - CB

The measured kernel times for hotspot are presented in Table 4.6.

Table 4.6: Influence of co-scheduling with hotspot as program A (30 repetitions).

	small			medium		
			Rel. Diff.			Rel. Diff.
min	0.64 s	0.64 s	100.00%	4.07 s	3.98 s	97.79%
mean	0.65 s	0.66 s	100.54%	4.38 s	4.32 s	98.68%
median	0.65 s	0.65 s	100.00%	4.38 s	4.29 s	97.95%
max	0.68 s	0.73 s	107.35%	4.73 s	5.01 s	105.92%
			Rel. Diff.			Rel. Diff.
hotspot - lud	1.14 s	1.15 s	100.88%	8.04 s	9.79 s	121.70%
hotspot - streamcluster	1.14 s	1.17 s	102.63%	8.04 s	11.23 s	139.61%
min	1.13 s	1.13 s	100.00%	7.78 s	7.61 s	97.81%
mean	1.15 s	1.16 s	100.72%	8.07 s	8.47 s	105.02%
median	1.14 s	1.15 s	100.88%	8.04 s	8.11 s	100.87%
max	1.19 s	1.46 s	122.69%	8.44 s	11.83 s	140.17%
			Rel. Diff.			Rel. Diff.
hotspot - b+tree	1.26 s	1.79 s	142.46%	5.86 s	9.77 s	166.64%
hotspot - backprop	1.26 s	2.12 s	168.25%	5.86 s	10.10 s	172.27%
hotspot - bfs	1.26 s	1.48 s	117.86%	5.86 s	6.52 s	111.26%
hotspot - heartwall	1.26 s	1.46 s	116.27%	5.86 s	6.38 s	108.87%
hotspot - hotspot3d	1.26 s	2.27 s	180.16%	5.86 s	9.14 s	155.97%
hotspot - kmeans	1.26 s	2.33 s	185.32%	5.86 s	15.08 s	257.34%
hotspot - lavaMD	1.26 s	1.66 s	131.75%	5.86 s	6.54 s	111.60%
hotspot - leukocyte	1.26 s	1.68 s	133.33%	5.86 s	6.31 s	107.76%
hotspot - lud	1.26 s	1.30 s	103.17%	5.86 s	7.94 s	135.49%
hotspot - particlefilter	1.26 s	2.20 s	174.60%	5.86 s	7.14 s	121.93%
hotspot - pathfinder	1.26 s	1.59 s	126.19%	5.86 s	6.77 s	115.53%
hotspot - srاد	1.26 s	2.33 s	184.52%	5.86 s	6.04 s	103.16%
hotspot - streamcluster	1.26 s	1.29 s	102.78%	5.86 s	8.04 s	137.29%
min	1.21 s	1.16 s	95.87%	5.54 s	5.35 s	96.57%
mean	1.26 s	1.83 s	145.05%	5.94 s	8.16 s	137.42%
median	1.26 s	1.77 s	140.08%	5.86 s	7.26 s	123.98%
max	1.32 s	3.18 s	240.91%	6.53 s	17.50 s	267.99%

There is no significant ratio for $A@S1$ and $A@S1 \mid B@S2$. Interestingly, we see a difference between $A@1/2S1$ and $A@1/2S1 \mid B@1/2S1$ for the combinations *hotspot-lud* and *hotspot-streamcluster* for the *medium* input size, and the ratio is significant, i.e., $> 120\%$. This means that for some co-scheduling combinations of applications or benchmarks the resource sharing on one socket might be problematic. This observation is important since we have not seen this behavior for *backprop* in Table 4.5.

For the *scatter* strategy $A@1/4S1+S2$ and $A@1/4S1+S2 \mid B@1/4S1+S2$, we notice less displayed combinations, e.g., *hotspot-nn* is missing, and this implies that there is no significant ratio present. Additionally, there are some co-scheduling combinations, *hotspot-srad medium* as an example, with an insignificant ratio. This is a new observation, since all combinations of both the *small* and *medium* input size for *backprop* imply relevant ratios and therefore relevant differences between a dedicated and co-scheduled execution. This phenomenon shows that the difference between dedicated and co-scheduled execution might not always correlate for both our input sizes.

Still, we notice a similar influence of co-scheduling with a *scatter mapping* on both sockets, the $A@1/4S1+S2 \mid B@1/4S1+S2$ co-scheduling strategy. As already discussed for *backprop*, this again might be derived from resource sharing problems regarding the communication between the two sockets.

myocyte - CB

For our last co-scheduling experiments exploring the influence of co-scheduling, we use *myocyte* as our application A. The obtained results are shown in Table 4.7.

As already seen for *backprop* and *hotspot*, there is no obvious resource sharing problem when each application uses a different socket, i.e., $A@S1 \mid B@S2$.













For the co-scheduling combination $A@1/2S1 \mid B@1/2S1$, we notice that the *small* and *medium* input size of *myocyte-lavaMD* do not correlate since there is no significant ratio for the *small* input size, but a big relative difference between the dedicated and co-scheduled execution time of the *medium* input size.

This non-correlation between the *small* and *medium* instances is also present for $A@1/4S1+S2 \mid B@1/4S1+S2$. Even though we are not interested in the correlation, but in which co-scheduling configurations might lead to resource sharing conflicts, this is an important observation.

Since several combinations of co-scheduling with *myocyte* as program A lead to an increased kernel time for $A@1/4S1+S2 \mid B@1/4S1+S2$ without a significant kernel time increase for $A@1/2S1 \mid B@1/2S1$, we conclude that the communication between the two sockets leads to this resource conflict, as we already observed for *backprop* in Table 4.5 and *hotspot* in Table 4.6.

4. CHARACTERIZING CO-SCHEDULED APPLICATIONS WITH HW PERFORMANCE COUNTERS

Table 4.7: Influence of co-scheduling with `myocyte` as program A (30 repetitions).

	small			medium		
			Rel. Diff.			Rel. Diff.
min	0.95 s	0.95 s	100.00%	4.40 s	4.40 s	100.00%
mean	0.96 s	0.96 s	99.95%	4.43 s	4.44 s	100.24%
median	0.96 s	0.96 s	100.00%	4.43 s	4.44 s	100.23%
max	0.97 s	0.99 s	102.06%	4.55 s	4.59 s	100.88%
			Rel. Diff.			Rel. Diff.
myocyte - lavaMD	1.47 s	1.47 s	100.00%	6.82 s	11.20 s	164.15%
min	1.46 s	1.46 s	100.00%	6.78 s	6.78 s	100.00%
mean	1.47 s	1.48 s	100.62%	6.83 s	7.17 s	105.03%
median	1.47 s	1.47 s	100.00%	6.82 s	6.86 s	100.59%
max	1.54 s	1.58 s	102.60%	6.96 s	11.46 s	164.66%
			Rel. Diff.			Rel. Diff.
myocyte - backprop	1.73 s	2.04 s	117.97%	7.49 s	7.63 s	101.87%
myocyte - hotspot3d	1.73 s	2.45 s	142.03%	7.49 s	7.71 s	102.94%
myocyte - hotspot	1.73 s	1.98 s	115.07%	7.49 s	7.76 s	103.67%
myocyte - kmeans	1.73 s	2.04 s	118.55%	7.49 s	7.49 s	100.00%
myocyte - leukocyte	1.73 s	1.94 s	112.46%	7.49 s	7.64 s	102.00%
myocyte - lud	1.73 s	1.90 s	110.14%	7.49 s	7.68 s	102.54%
myocyte - nn	1.73 s	2.13 s	123.48%	7.49 s	7.66 s	102.27%
myocyte - pathfinder	1.73 s	2.00 s	116.23%	7.49 s	7.36 s	98.26%
myocyte - srad	1.73 s	2.09 s	121.16%	7.49 s	7.62 s	101.67%
myocyte - streamcluster	1.73 s	1.97 s	114.20%	7.49 s	7.48 s	99.87%
min	1.51 s	1.50 s	99.34%	7.04 s	7.06 s	100.28%
mean	1.82 s	2.13 s	116.79%	7.62 s	7.81 s	102.48%
median	1.73 s	1.93 s	111.88%	7.49 s	7.54 s	100.67%
max	3.67 s	5.82 s	158.58%	9.41 s	19.07 s	202.66%

As a summary of gained information from this chapter, we conclude:

- There are several ways to co-schedule two applications on `hydra`. We choose three types of resource sharing granularities to find resource sharing conflicts: sharing the compute node by using different sockets, sharing one socket, and sharing both sockets.

- To measure the kernel time of co-scheduled applications, it is necessary to synchronize kernel sections with one another, otherwise pre- and postprocessing steps can be executed concurrently with the kernel section, which often leads to distorted measurement results.

The synchronization itself can be accomplished with a lightweight library using semaphores as the synchronization mechanism.

- Sharing a compute node, i.e., where each co-scheduled application is assigned to a separate socket or NUMA domain, does not lead to increased co-scheduled kernel times and therefore can be considered *resource conflict free*.
- Sharing a socket with a *compact* core mapping often does not lead to increased kernel times, which also implies a *resource conflict free* co-scheduling. Even though two applications share one socket with the same resources, we only observe occasional resource conflicts.
- Sharing both sockets with a *scatter* core mapping means more communication between the sockets/NUMA domains. This is already visible for dedicated executions because the kernel time is increased for $A@^{1/4}S1+S2$ compared to $A@^{1/2}S1$, even though both configurations use eight cores. Since the co-scheduled times of $A@^{1/4}S1+S2$ | $B@^{1/4}S1+S2$ are increased, we assume resource sharing conflicts regarding the communication between the sockets/NUMA domains.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Prerequisites for Predicting Co-Scheduling Behaviors

We are interested in predicting the co-scheduling potential of two applications, i.e., to answer our question: *To co-schedule or not to co-schedule?* For this, we have to analyze what we need for this prediction, i.e., what data we can use for training a prediction model. Therefore, these questions motivate the work of this chapter:

- What do we need for creating a prediction model to predict the co-scheduling potential of two applications?
- Are performance counters reliable and suitable for a prediction?
Which hardware counter changes its value between dedicated and co-scheduled executions, i.e., which hardware counter helps identifying the quality or goodness of a co-scheduling?

5.1 Correlation of Performance Metrics with Time

We assess whether it is possible to predict the co-scheduling potential by using performance or hardware counters. For this, we need two types of measurements: co-scheduled executions that do not increase the dedicated kernel time and co-scheduled executions that increase the dedicated kernel time. These two types are necessary to find correlations between performance metrics and the kernel time, to further predict whether applications should be co-scheduled or not by means of performance counters. Therefore, we use the *scatter* strategy from the previous chapter, where the co-scheduling of two programs A and B takes place on both sockets: $A@1/4S1+S2 \mid B@1/4S1+S2$. Since these *scatter* results often show an increased co-scheduled kernel time compared to the dedicated runs, we hope to see these observations reflected in performance counter values. We pose our

essential hypothesis: hardware performance counters correlate with the kernel time, e.g., if the kernel time increases, also one of the counter values has to increase, say the L3 cache miss ratio.

5.1.1 Difficulties with likwid Performance Counters

We had already analyzed the arithmetic intensity of Rodinia and SPEC OMP2012 benchmarks with the Marker API of `likwid`. Therefore, we continued using `likwid` to measure performance metrics.

There are several performance groups provided by `likwid`, where a single execution can measure values that are part of a specific performance group. We execute all performance groups to get an overview of which groups might show a correlation for increased kernel times, and choose the following four `likwid` performance groups: `CYCLE_STALLS`, `L2CACHE`, `TLB_DATA`, `TLB_INSTR`. Table 5.1 shows the measured performance counters of these groups.

Table 5.1: `likwid` performance counters and their corresponding group. For all performance counters, we use the aggregated `STAT` value for measuring counter values on all used cores.

Group	Performance Counter	Our Notation
CYCLE_STALLS	Stalls caused by L1D misses [%]	<code>stalls.l1.misses</code>
	Stalls caused by L2 misses [%]	<code>stalls.l2.misses</code>
	Stalls caused by L1D misses rate [%]	<code>stalls.l1d.miss.rate</code>
	Stalls caused by L2 misses rate [%]	<code>stalls.l2.miss.rate</code>
L2CACHE	L2 request rate	<code>l2.request.rate</code>
	L2 miss rate	<code>l2.miss.rate</code>
	L2 miss ratio	<code>l2.miss.ratio</code>
TLB_DATA	L1 DTLB load misses	<code>l1.dtlb.load.misses</code>
	L1 DTLB load miss rate	<code>l1.dtlb.load.miss.rate</code>
	L1 DTLB load miss duration [Cyc]	<code>l1.dtlb.load.miss.duration</code>
	L1 DTLB store misses	<code>l1.dtlb.store.misses</code>
	L1 DTLB store miss rate	<code>l1.dtlb.store.miss.rate</code>
	L1 DTLB store miss duration [Cyc]	<code>l1.dtlb.store.miss.duration</code>
TLB_INSTR	L1 ITLB misses	<code>l1.itlb.misses</code>
	L1 ITLB miss rate	<code>l1.itlb.miss.rate</code>
	L1 ITLB miss duration [Cyc]	<code>l1.itlb.miss.duration</code>

We pick one example from the $A@^{1/4}S1+S2$ vs. $A@^{1/4}S1+S2 \mid B@^{1/4}S1+S2$ experiments above that shows a big relative difference between the dedicated and co-scheduled execution: `hotspot-backprop` with a *small* input size. This big relative difference is necessary such that it is possible to see a correlation between increasing kernel

times and increasing counter values. For each `likwid` performance group, we execute 1000 dedicated `hotspot@1/4S1+S2`, and 1000 co-scheduled `hotspot@1/4S1+S2 | backprop@1/4S1+S2` runs.

For each `likwid` performance group, we plot a correlation matrix shown in Figure 5.1.

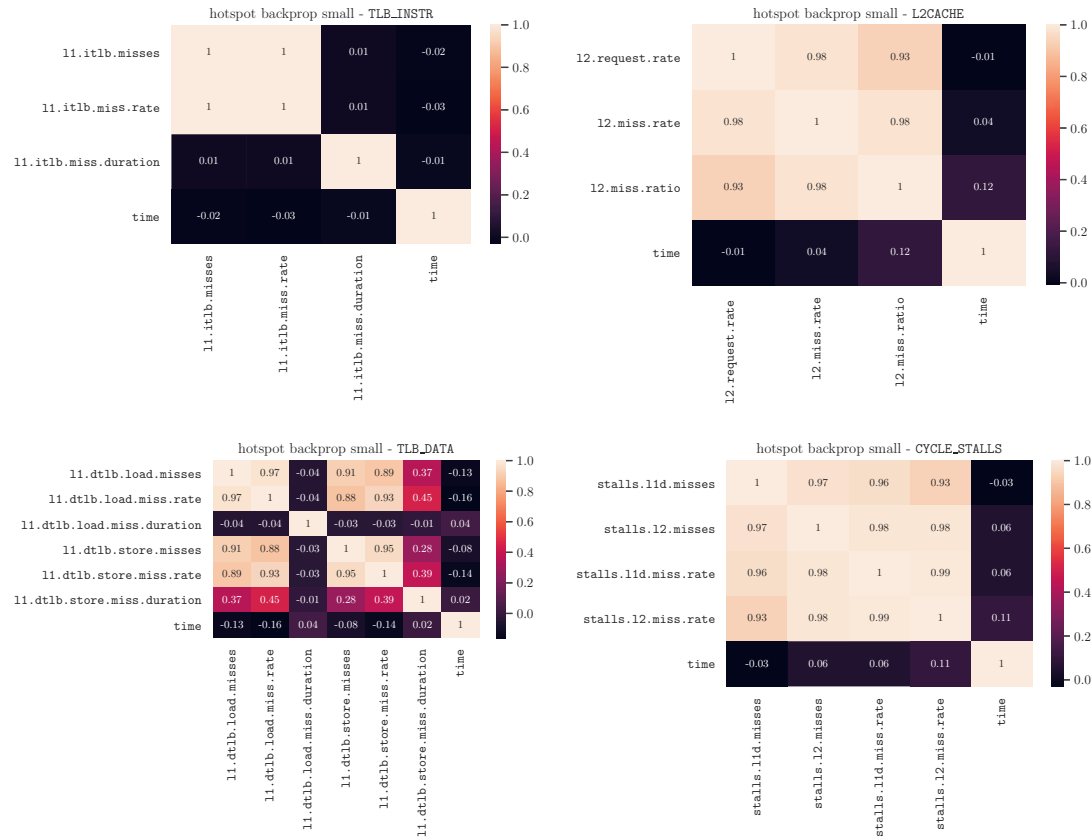


Figure 5.1: Correlation matrices of `likwid` performance counters of `hotspot` and `backprop` (correlation from 1000 repetitions).

Since a correlation between any counter and kernel time is necessary to predict whether an application with similar performance metric values should be co-scheduled or not, we notice a problem: there is no significant correlation between the metrics and the time. This non-correlation of performance values with the time is not a phenomenon appearing solely for the `hotspot-backprop` combination, but for several other execution combinations too.

Additionally to this correlation problem, we notice another complication with our measurements: measuring performance values for the same code region using the `likwid` Marker API leads to different kernel times of this section, i.e., the kernel time of one and the same code region fluctuates for different executions. These kernel time measurements fluctuate highly. Without measuring performance counter values using the

likwid Marker API, we do not notice this kind of strong fluctuation. Therefore, we deem the usage of likwid to measure performance values of kernel sections for our prediction unreasonable.

5.1.2 Correlation of PAPI Performance Events

As an alternative, we try using PAPI instead of likwid. We do the same experiments as described above, i.e., 1000 dedicated and 1000 co-scheduled runs of some application combinations while measuring performance metrics. As above, we use the co-scheduling combination `hotspot-backprop` with a *small* input size, and additionally take a look at metrics of `hotspot-hotspot3d` with a *small* input size. For evaluating PAPI events, we use the PAPI Low Level API.

We want to measure all performance events available on `hydra` that are not derived from any other event. Therefore, we use the `papi_avail` command and take all non-derived performance events. On `hydra`, there are 37 such events. Since these performance events use registers to count the metrics, it is not possible to measure all 37 events altogether in one run. To determine, which events can be measured concurrently, we use the command `papi_event_chooser PRESET <event> ... <event>`. With this, we create four custom performance event groups Cycles, Branches, Load/Store misses, Instructions, and Data. These groups with the corresponding PAPI events are listed in Table 5.2.

Table 5.2: Performance groups from the set of PAPI performance events.

Group	Event	Description
Cycles	PAPI_MEM_WCY	Cycles Stalled Waiting for memory writes
	PAPI_STL_ICY	Cycles with no instruction issue
	PAPI_STL_CCY	Cycles with no instructions completed
	PAPI_FUL_CCY	Cycles with maximum instructions completed
	PAPI_RES_STL	Cycles stalled on any resource
	PAPI_TOT_CYC	Total cycles
	PAPI_REF_CYC	Reference clock cycles
Branches, Load/Store misses	PAPI_BR_CN	Conditional branch instructions
	PAPI_BR_NTK	Conditional branch instructions not taken
	PAPI_BR_MSP	Conditional branch instructions mispredicted
	PAPI_L3_LDM	Level 3 load misses
	PAPI_L1_LDM	Level 1 load misses
	PAPI_L1_STM	Level 1 store misses
	PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses	

Table 5.2 continued

Group	Event	Description
Instructions	PAPI_TOT_INS	Instructions completed
	PAPI_LD_INS	Load instructions
	PAPI_SR_INS	Store instructions
	PAPI_BR_INS	Branch instructions
	PAPI_L1_ICM	Level 1 instruction cache misses
	PAPI_L2_ICM	Level 2 instruction cache misses
	PAPI_L2_ICH	Level 2 instruction cache hits
	PAPI_L2_ICA	Level 2 instruction cache accesses
	PAPI_L3_ICA	Level 3 instruction cache accesses
	PAPI_L2_ICR	Level 2 instruction cache reads
	PAPI_L3_ICR	Level 3 instruction cache reads
	PAPI_TLB_IM	Instruction translation lookaside buffer misses
	PAPI_PRF_DM	Data prefetch cache misses
Data	PAPI_L1_DCM	Level 1 data cache misses
	PAPI_L2_TCM	Level 2 cache misses
	PAPI_L3_TCM	Level 3 cache misses
	PAPI_L2_DCA	Level 2 data cache accesses
	PAPI_L2_DCR	Level 2 data cache reads
	PAPI_L3_DCR	Level 3 data cache reads
	PAPI_L3_DCW	Level 3 data cache writes
	PAPI_L3_TCA	Level 3 total cache accesses
PAPI_L3_TCW	Level 3 total cache writes	

As mentioned above, we pick two execution combinations for getting an overview of the correlation between PAPI events and the kernel time: `hotspot-backprop` and `hotspot-hotspot3d`. These two combinations lead to higher running times when being co-scheduled compared to `hotspot` executed in a dedicated mode.

Since we are interested in the correlation between performance events and the running time, the data set used for the correlation matrix contains both the dedicated events and running times of `hotspot`, as well as the co-scheduled events and running times of `hotspot` with `backprop` or `hotspot3d` correspondingly.

Correlation Matrices of `hotspot-backprop`

We use the defined PAPI groups from Table 5.2 and plot the correlation matrices for `hotspot-backprop` with a *small* input size in Figure 5.2. We are interested in the correlation between the `time`, i.e., the kernel time, and PAPI performance events. This correlation matrices display a high correlation with a light color, while small correlations are displayed as dark areas. When we look at the `time` columns of all four matrices,

5. PREREQUISITES FOR PREDICTING CO-SCHEDULING BEHAVIORS

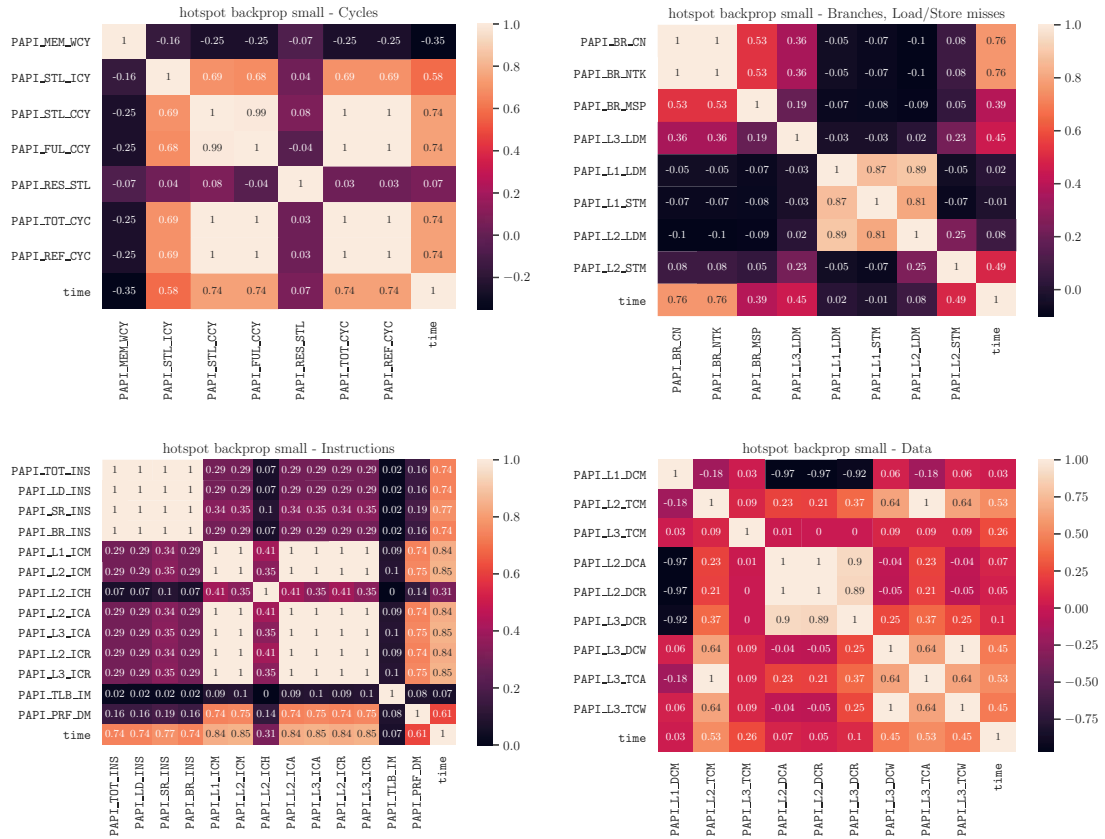


Figure 5.2: Correlation matrices of PAPI performance events of hot spot and backprop (correlation from 1000 repetitions).

we notice several PAPI performance events showing a correlation with the time, e.g., `PAPI_TOT_CYC`, `PAPI_BR_CN`, `PAPI_TOT_INS`, `PAPI_L1_ICM`, and several others.

Correlation Matrices of hotspot-hotspot3d

We also take a look at another execution combination, `hotspot-hotspot3d`, to make sure that the observed high correlations from Figure 5.2 are not artifacts.

The correlation matrices of our PAPI groups for `hotspot-hotspot3d` with a *small* input size are plotted in Figure 5.3. Again, we see a high correlation for several PAPI events, e.g., `PAPI_TOT_CYC`, `PAPI_BR_CN`, `PAPI_TOT_INS`, `PAPI_L1_ICM`, and others. Interestingly, the correlation between the counter values and the time is not that high as for `hotspot-backprop` shown in Figure 5.2, but we notice that similar performance counters show a good correlation with the kernel time.

5.2. Limited Group of Performance Metrics Relevant for Prediction

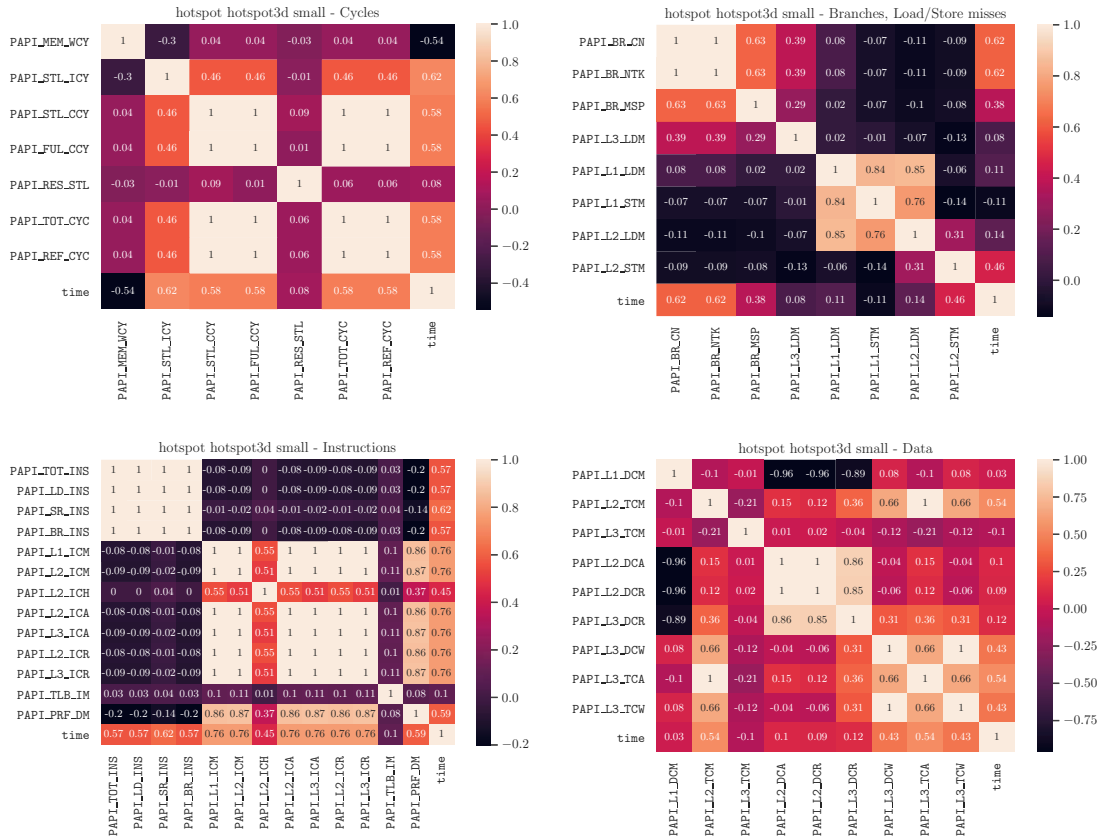


Figure 5.3: Correlation matrices of PAPI performance events of hotspot and hotspot3d (correlation from 1000 repetitions).

Therefore, we can say that an increased kernel time is reflected in several increased PAPI performance events for both hotspot-backprop and hotspot-hotspot3d. This correlation between performance events and the kernel time implies that it is possible to predict increased kernel times using performance and hardware counters. Thus, it is possible to predict the co-scheduling potential of applications.

5.2 Limited Group of Performance Metrics Relevant for Prediction

As we see in Figures 5.2 and 5.3, not all PAPI performance events correlate with the kernel time. Additionally, we measure all non-derived PAPI counters. Since hardware counters are measured by counting events in registers, some PAPI performance events cannot be measured concurrently, because they share the same register for counting. This means that we need four executions to measure all non-derived events. Therefore, we are interested in limiting these performance counters to measure them in a single execution.

5.2.1 Finding Good Representatives

The goal is to find a group of representatives, such that we can measure relevant performance events with a single execution. To determine, which metrics are useful, we analyze the correlation matrices from Figures 5.2 and 5.3. As we can see, some of these PAPI events show a high correlation with the kernel time. Additionally, there are several events that show a similar correlation with the time and further on correlate perfectly with one another.

Therefore, we remove strongly correlating performance events from our set of performance events. These events are those performance events that correlate perfectly with one another, i.e., they have a correlation of one. Such events represent a similar metric and measuring any counter out of these similar events gives us the same information regarding the correlation with the kernel time.

Taking a closer look at the correlation matrices from Figures 5.2 and 5.3, we see a perfect correlation between each one of `PAPI_TOT_CYC`, `PAPI_STL_CYC`, `PAPI_FUL_CYC`, and `PAPI_REF_CYC`. Therefore, we choose one of these four metrics since all four of them are strongly correlating. With this procedure, we eliminate strongly correlating performance events and minimize the set of hardware counters.

Still, there are many events after this elimination and we want to know, which performance counter values grow when the base application is co-scheduled. Therefore, we measure this smaller set of performance events with PAPI and compare performance values when an application is executed dedicated and when being co-scheduled.

For this experimental setup, we choose an application A as our base application. We then need to find two other applications: one that changes the runtime behavior of A when being co-scheduled, application B, and one that does not change the runtime behavior of A, application C.

For each of these executions A-B and A-C, we measure the performance events 1000 times in two ways:

- A being executed dedicated, and
- A being co-scheduled with B or C.

These measured performance values are then inspected using a *one-way analysis of variance tests* (anova) in R. We are interested if there is a significant difference between the average performance counter values in the two scenarios of dedicated and co-scheduled executions of A.

For each performance counter separately, we perform an anova (called `aov` in R) for the execution of A-B and an anova for the execution of A-C, as shown in Listing 5.1.

Listing 5.1: Anova in R to find significant performance events.

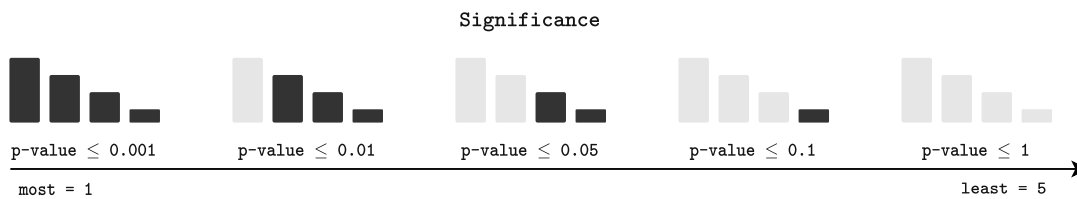
```

data_a_b # data of execution of A-B
data_a_c # data of execution of A-C
# both data frames have a column for each performance counter
# both data frames have a column ded_co which contains either the value
# 'dedicated' or 'co-scheduled'

for (perf_counter in all_performance_counter_names) {
  aov_a_b <- aov(data_a_b[[perf_counter]] ~ data_a_b[[ded_co]])
  aov_a_c <- aov(data_a_c[[perf_counter]] ~ data_a_c[[ded_co]])
}

```

The results of an *analysis of variance* provide a p-value for each test. This p-value can then be used to get knowledge whether there are significant differences between the groups, namely between dedicated and co-scheduled executions. To determine the significance of differences between dedicated and co-scheduled executions, we use the significance levels shown in Figure 5.4.

Figure 5.4: Significance levels of an *analysis of variance*.

As Figure 5.4 shows, we can say that the difference between a dedicated and a co-scheduled execution is significant, if $p\text{-value} \leq 0.05$. Still, there are different gradations of significance, like a $p\text{-value} \leq 0.001$ indicates the most significant difference between mean values of groups.

If we encode the significance to integer numbers, i.e., the most significant difference is 1, the least significant difference is 5, then we could measure the absolute difference between two significance levels and use this as a *relevance* measurement, as shown in Figure 5.5.

The intention for this *relevance* is to show whether a performance counter measured for A-B and A-C has a similar significance level, or if there is a difference in significance for A-B and A-C.

As in our example, the co-scheduling of A-B changes the runtime behavior of A, while this does not happen with the co-scheduling of A-C. Hence, if there is a significant difference of a performance counter when being executed dedicated and co-scheduled for A-B, but there is no significant difference for A-C, we assume that this performance event may be a good coefficient for any prediction regarding co-scheduled runtime behaviors.

We stick with `hotspot` as program A, and therefore need to find suitable programs B and C for this significance and relevance analysis of performance events. Therefore, we

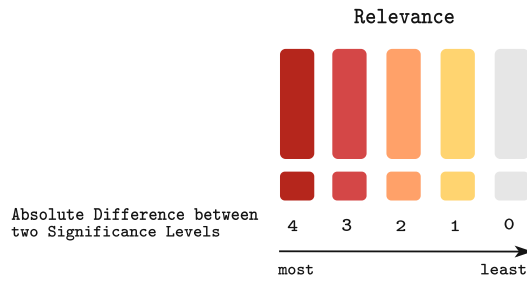


Figure 5.5: Relevance of the difference between two significance levels.

measure the `Cycles` group of performance events (any group would be a valid choice), and monitor the kernel times. Figure 5.6 presents the observed kernel times of 1000 runs.

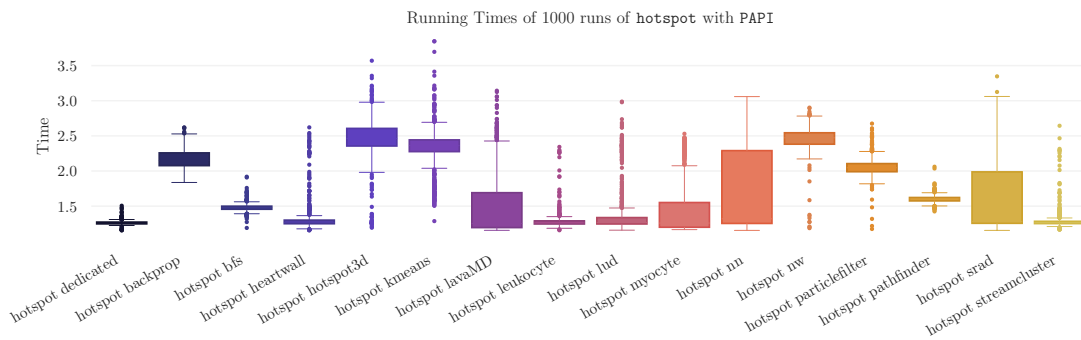


Figure 5.6: Running times of 1000 dedicated and co-scheduled runs of `hotspot` using PAPI as the performance measurement tool.

Using these observed kernel times, we choose the three combinations listed in Table 5.3 to observe different significance behaviors of performance events.

Table 5.3: Experimental setup for finding relevant performance counters.

Base Application A	Does not affect A = Application C	Affects A = Application B
hotspot (A)	streamcluster (C)	hotspot3d (B)
hotspot (A)	leukocyte (C)	nw (B)
hotspot (A)	heartwall (C)	backprop (B)

As we can see in Figure 5.6, the co-scheduled execution of `hotspot-streamcluster`, `hotspot-leukocyte`, and `hotspot-heartwall` lead to similar median kernel times as the dedicated execution of `hotspot`. Therefore, we choose `streamcluster`, `leukocyte`, and `heartwall` as our program C. In comparison, `hotspot-hotspot3d`, `hotspot-nw`, and `hotspot-backprop` lead to significantly increased kernel times com-

pared to the dedicated execution of `hotspot`, hence `hotspot3d`, `nw`, and `backprop` represent program B. In case we see a non-significant difference of any performance event between the dedicated and co-scheduled execution of A-C, but a significant difference between the dedicated and co-scheduled execution of A-B for the same performance event, we conclude that this event might be relevant for predicting the co-scheduling potential of applications.

Case: `hotspot` (A) | `streamcluster` (C) | `hotspot3d` (B)

The co-scheduled runtime of `hotspot-streamcluster` is not significantly increased compared to the dedicated execution of `hotspot`. But the co-scheduled runtime of `hotspot-hotspot3d` is significantly higher than the dedicated runtime of `hotspot`. Therefore, we compare the significance of the differences of the performance events between dedicated and co-scheduled executions. The results obtained from the analysis of variance for 1000 measurements are shown in Table 5.4.

Table 5.4: Significance and relevance of PAPI performance events for `hotspot` co-scheduled with `streamcluster` and `hotspot3d`.

Performance Event	Significance of hotspot-streamcluster	Significance of hotspot-hotspot3d	Relevance
PAPI_MEM_WCY	0.83	0.00	!
PAPI_STL_ICY	0.00	0.00	!
PAPI_RES_STL	0.04	0.00	!
PAPI_TOT_CYC	0.00	0.00	!
PAPI_BR_CN	0.42	0.00	!
PAPI_BR_MSP	0.00	0.00	!
PAPI_L3_LDM	0.00	0.00	!
PAPI_L1_LDM	0.09	0.00	!
PAPI_L1_STM	0.07	0.00	!
PAPI_L2_LDM	0.00	0.00	!
PAPI_L2_STM	0.00	0.00	!
PAPI_TOT_INS	0.66	0.00	!
PAPI_L1_ICM	0.00	0.00	!
PAPI_L2_ICH	0.57	0.00	!
PAPI_TLB_IM	0.24	0.00	!
PAPI_PRF_DM	0.07	0.00	!
PAPI_L1_DCM	0.06	0.21	!
PAPI_L2_TCM	0.10	0.00	!
PAPI_L3_TCM	0.00	0.00	!
PAPI_L2_DCR	0.08	0.00	!
PAPI_L3_DCR	0.02	0.00	!
PAPI_L3_DCW	0.00	0.00	!

As Table 5.4 shows, the following PAPI performance events can be considered relevant: PAPI_MEM_WCY, PAPI_BR_CN, PAPI_L1_LDM, PAPI_L1_STM, PAPI_TOT_INS, PAPI_L2_ICH, PAPI_TLB_IM, PAPI_PRF_DM, PAPI_L2_TCM, and PAPI_L2_DCR.

Case: hotspot (A) | leukocyte (C) | nw (B)

To prevent artifacts, we also look at other co-scheduled executions. The co-scheduled execution of hotspot-leukocyte does not significantly increase compared to the dedicated execution of hotspot, whereas hotspot-nw does. The significance and relevance of the PAPI performance events observed by the analysis of variance are shown in Table 5.5.

Table 5.5: Significance and relevance of PAPI performance events for hotspot co-scheduled with leukocyte and nw.

Performance Event	Significance of hotspot-leukocyte	Significance of hotspot-nw	Relevance
PAPI_MEM_WCY	0.00	0.00	■
PAPI_STL_ICY	0.33	0.00	■
PAPI_RES_STL	0.62	0.00	■
PAPI_TOT_CYC	0.00	0.00	■
PAPI_BR_CN	0.00	0.00	■
PAPI_BR_MSP	0.53	0.00	■
PAPI_L3_LDM	0.00	0.00	■
PAPI_L1_LDM	0.25	0.00	■
PAPI_L1_STM	0.18	0.00	■
PAPI_L2_LDM	0.19	0.00	■
PAPI_L2_STM	0.01	0.00	■
PAPI_TOT_INS	0.00	0.00	■
PAPI_L1_ICM	0.01	0.00	■
PAPI_L2_ICH	0.00	0.00	■
PAPI_TLB_IM	0.01	0.00	■
PAPI_PRF_DM	0.29	0.00	■
PAPI_L1_DCM	0.00	0.46	■
PAPI_L2_TCM	0.97	0.00	■
PAPI_L3_TCM	0.00	0.00	■
PAPI_L2_DCR	0.14	0.00	■
PAPI_L3_DCR	0.00	0.00	■
PAPI_L3_DCW	0.78	0.00	■

Table 5.5 shows possibly relevant PAPI events: PAPI_STL_ICY, PAPI_RES_STL, PAPI_BR_MSP, PAPI_L1_LDM, PAPI_L1_STM, PAPI_L2_LDM, PAPI_PRF_DM, PAPI_L2_TCM, PAPI_L2_DCR, PAPI_L3_DCW. We have observed some of these events

already as being *relevant* in the combination `hotspot-streamcluster-hotspot3d`, but other events appear to be *relevant* for this combination too.

Case: hotspot (A) | heartwall (C) | backprop (B)

As another attempt to prevent artifacts, we look at the significance differences between `hotspot-heartwall` and `hotspot-backprop` shown in Table 5.6.

Table 5.6: Significance and relevance of PAPI performance events for hotspot co-scheduled with heartwall and backprop.

Performance Event	Significance of hotspot-heartwall	Significance of hotspot-backprop	Relevance
PAPI_MEM_WCY	0.68	0.00	!
PAPI_STL_ICY	0.02	0.00	!
PAPI_RES_STL	0.97	0.00	!
PAPI_TOT_CYC	0.00	0.00	
PAPI_BR_CN	0.00	0.00	
PAPI_BR_MSP	0.00	0.00	
PAPI_L3_LDM	0.32	0.00	!
PAPI_L1_LDM	0.82	0.41	
PAPI_L1_STM	0.06	0.75	
PAPI_L2_LDM	0.13	0.00	!
PAPI_L2_STM	0.00	0.00	
PAPI_TOT_INS	0.00	0.00	
PAPI_L1_ICM	0.00	0.00	
PAPI_L2_ICH	0.04	0.00	!
PAPI_TLB_IM	0.01	0.00	
PAPI_PREF_DM	0.00	0.00	
PAPI_L1_DCM	0.01	0.24	!
PAPI_L2_TCM	0.09	0.00	!
PAPI_L3_TCM	0.20	0.00	!
PAPI_L2_DCR	0.01	0.02	
PAPI_L3_DCR	0.01	0.00	
PAPI_L3_DCW	0.00	0.00	

As the results in Table 5.6 show, a new set of performance events is deemed relevant: `PAPI_MEM_WCY`, `PAPI_RES_STL`, `PAPI_L3_LDM`, `PAPI_L2_LDM`, `PAPI_L2_TCM`, and `PAPI_L3_TCM`. As we notice, these events already occurred *relevant* in at least one of our previous experiment results from Tables 5.4 and 5.5. Still, not all performance events show this *relevance* in all our experiments, and additionally these experiments do not cover the whole range of possibilities. Therefore, it is important to carefully choose our limited custom group of PAPI performance events.

5.2.2 Selecting Relevant Performance Events

Using the *relevance* experiments and the correlation matrices of event groups, we create a new group of events, which combines performance events that seem to be *relevant* for us. Strongly correlating performance events have already been removed, such that there is only one representative of these perfectly correlating counters. By creating a new performance event group, we can make sure that it is only necessary to execute an application once to get *relevant* performance values for the later prediction. This group of events contains the PAPI performance events shown in Table 5.7.

Table 5.7: Performance group formed containing *relevant* performance events for co-scheduling prediction.

Event	Description
PAPI_TOT_CYC	Total cycles
PAPI_STL_ICY	Cycles with no instruction issue
PAPI_BR_CN	Conditional branch instructions
PAPI_L3_LDM	Level 3 load misses
PAPI_TOT_INS	Instructions completed
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_ICH	Level 2 instruction cache hits
PAPI_L3_TCM	Level 3 cache misses
PAPI_L3_DCW	Level 3 data cache writes

The code template we used for measuring these performance events is provided in Appendix 7 in Listing 1.

We summarize our observations from this chapter:

- Performance counters have to correlate with the elapsed time of kernel sections. Then, we can predict increased kernel times and thus predict the co-scheduling potential of applications.
- The `likwid` Marker API leads to inconsistent results when measuring hardware counters of kernel sections. On one hand the time measurements fluctuate highly for same executions, on the other hand performance counters and the time do not correlate with one another.
- We analyzed the correlation, significance, and relevance of PAPI performance events. Then, we defined a small group of PAPI performance events (shown in Table 5.7) that correlate highly with kernel time measurements. These counters can be measured concurrently, thus, a single execution suffices to measure the necessary performance events.

Predicting Co-Scheduling Potentials

We want to predict the co-scheduling potential of two applications. The idea is to use performance counter measurements to create a prediction model and further determine, whether applications should be co-scheduled or not. In this chapter, we present the idea of a prediction model for predicting the co-scheduling potential of two applications, and then evaluate the prediction model. The following questions will be answered:

- Can we predict the co-scheduling potential of two applications? What should be predicted to find out, whether two applications can be co-scheduled or not?
- Which data should be used for training and evaluating the model? Can we use measurements directly or are preliminary preparations necessary?
- How good does the prediction work?

6.1 Prediction Model Idea

We are interested in predicting the performance of co-scheduling two applications and whether we should co-schedule these applications or not. We assume having two applications A and B, and we want to know if we can co-schedule program A with program B without delaying the execution time of A significantly. Under the assumption that the execution of program A is long, e.g., half an hour or even more, we want to measure *relevant* performance events for a small amount of time and decide with this information whether to co-schedule A with B or run A in a dedicated mode. These mentioned *relevant* performance events are those events that help identifying the co-scheduling potential. We take the PAPI performance events listed in Table 5.7.

There are only two possible answers to our prediction question of whether to co-schedule A and B: *yes* or *no*. Since our problem represents a binary classification problem, we use a logistic regression.

We measure the performance events of A for a small amount of time in dedicated mode and in co-scheduled mode with B and use the classification

$$co_schedule_A_B = \begin{cases} 1 & \text{if } co_scheduled_time_A \leq t \times min_dedicated_time_A, \\ 0 & \text{otherwise,} \end{cases} \quad (6.1)$$

where $t > 1$ stands for a threshold value.

6.2 Building a Prediction Model

For our prediction model, we use the dedicated and co-scheduling combination from our *scatter* affinity mapping: $A@^{1/4}S1+S2$ for the dedicated execution of A and $A@^{1/4}S1+S2 \mid B@^{1/4}S1+S2$ for the co-scheduling of A and B. We choose this affinity mapping since this leads to diverging results when it comes to co-scheduling. Therefore, we should be able to see changing performance event values when the runtime behavior changes.

We continue using the Rodinia benchmark suite and use the following 14 applications: *backprop*, *bfs*, *heartwall*, *hotspot*, *hotspot3d*, *kmeans*, *lavaMD*, *leukocyte*, *lud*, *myocyte*, *nn*, *pathfinder*, *srad*, and *streamcluster*. One of these 14 benchmarks represents application A and the other benchmarks application B.

The goal is to predict the co-scheduling potential of A and B, where these two applications have a long running time. Therefore, we co-schedule A and B for a small amount of time and predict the co-scheduling potential of the long run by using the measured performance events of the short execution. This means that the prediction model has to be trained with the small execution results and validated with the long execution results.

Our initial approach was to run the long execution and stop the performance event measurement, along with this long execution run, after a few seconds. Since this is rather complicated with core mapping and additionally intrusive for the benchmark to stop its execution after some seconds, we use the *small* instances of Rodinia as the short measurement, and the *medium* instances as the long measurement. Subsequently, we train our logistic regression model with co-scheduled measurements of the *small* input size, and validate the model with co-scheduled measurements of the *medium* input size.

As mentioned previously, there are 14 Rodinia applications. If we choose one application to represent A, then there are 13 combinations for a co-scheduling of A and B. We measure the performance events of a dedicated execution of A with a *small* input size 1000 times, and also each of the 13 co-scheduling combinations with a *small* input size 1000 times.

The training set of the model only consists of the co-scheduled runs without the dedicated measurements, we only use the kernel time of the dedicated runs to assign either 0 or 1 to the decision variable *co_schedule_A_B*. This means that the training set consists of 13 000 entries. For the validation we use the *medium* input size and only execute each co-scheduling combination 500-times. Therefore, the validation set contains 6 500 entries.

6.2.1 Data Normalization

To make sure that we can train the model with the *small* input size, but predict the long execution with the *medium* instances, we need to normalize our performance values. It is important that there is no difference between taking the performance event values from the short or long execution. We achieve this requirement by calculating the event value per time ratio to make sure that $long_run_time/sec \equiv short_run_time/sec$. Additionally, it is very common to normalize data to get better prediction models [16, 41, 43], since models work better if data is scaled to a common range.

As a first step we normalize the performance values per time, i.e., we get a value per time ratio. For a performance value e , we define $x = e/sec$. We then normalize this value x with machine learning normalization methods.

There are several common ways to normalize data for machine learning, e.g., *min-max scaling*, *z-score method*, or *robust scaling*.

The *min-max* normalization normalizes a value x according to the formula $x_{norm} = (x - x_{min}) / (x_{max} - x_{min})$. The values x_{min} and x_{max} are taken from all measurements for x . The normalized value x_{norm} lies in the range $[0, 1]$. The drawback of this normalization is that the scaling is sensitive to outliers [43] since we use the minimum and maximum value for scaling.

The *z-score* method uses the mean μ and standard deviation σ of the measured values [41]. Therefore, it is also called *standardization*. The standardized x value is calculated by $x_{std} = (x - \mu) / \sigma$. This normalization method also has difficulties with outliers, since the mean and standard deviation are affected by outliers.

The *robust scaling* calculates the normalized value x_{norm} by taking the median and quartile values of the measured x values into account. As quartile values are not affected by outliers compared to the mean or min-/max-values [43], this *robust scaling* eliminates drawbacks of the *min-max* scaling and the *z-score* method. The normalized x value is calculated by $x_{norm} = (x - Q_2(x)) / (Q_3(x) - Q_1(x))$, where $Q_2(x)$ equals the median value of all x values, and $Q_1(x)$ and $Q_3(x)$ present the first and third quartile, respectively.

We experiment with all three of these scaling methods and eventually use the *robust scaling* for our prediction model since it is the most robust normalization method and works the best in our settings. Since we train the model with *small* input sized data, we also use this *small* input set for normalization. The normalization for the validation set is done independently for each y of the validation set by normalizing the data using the median, first, and third quartile values of the training set, i.e., the *small* instances.

6.2.2 How to Choose Program A?

From Definition 6.1, we can say that a good candidate for training a model has a similar amount of zeros and ones, such that both classification groups are represented *equally* in the model.

As shown in Table 4.6, `backprop`, `hotspot`, `hotspot3d`, and `myocyte` being co-scheduled with other applications using the *scatter* mapping often leads to a big relative time difference between dedicated and co-scheduled executions. Therefore, these applications seem like good candidates to train a model for predicting whether to co-schedule or not, due to different runtime behaviors. Such applications, like `backprop`, `hotspot`, `hotspot3d`, or `myocyte`, are therefore good candidates as application A because the training set can be balanced regarding the amount of zeros and ones.

6.2.3 Functionality of the Prediction Model

For our prediction model, we create a logistic regression model. The pseudo code is listed in Appendix 7.

We read in our measurements with the PAPI event values and the kernel time of the *small* and *medium* input sized runs and create an additional column representing the classification of a *good time*. For creating this classification in the method `create_good_bad_column`, we use the minimum kernel time measured for the dedicated runs of program A and then create this classification column for all co-scheduled measurements. We use the threshold value $t = 1.35$ for the formula shown in Definition 6.1 that is used to create a prediction model for a program A. If the co-scheduled time of an entry is smaller or equal to the minimum dedicated time \times the threshold t , then we classify this as a *good-time*, otherwise it is *not a good-time*. This threshold value t means that co-scheduled times with an increase of less than 35% compared to the dedicated kernel time are considered as a *good time*. Still, it is possible to specify this threshold value explicitly for one application. If we only create a prediction model for an application like LAMMPS, then it may be better to rethink the threshold value and specify it for this application explicitly. As a next step, the performance measurements have to be normalized, where we normalize the validation data with the measurements from the training data. This is necessary because we assume that we only have the training data set. If a new measurement is given as input to the prediction model, we have to predict the co-scheduling potential only using the provided training data.

6.3 Evaluation of the Prediction Model

For the model evaluation, we choose several Rodinia benchmarks to be program A. We analyze the distribution of the PAPI performance events normalized per time for the *small* and *medium* input size. Since we define a common threshold for all used applications as program A, we also take a look at the balance of the classification of the training data. We want to prevent a high imbalance because this would lead to a biased prediction model.

6.3.1 Test and Train Data Sets

Before we use all of the *small* input sized runs as our training set and predict the co-scheduling potential of the validation set with the *medium* input size, we have to make sure that the training set can create a sufficiently good prediction model. As already argued, there are 13 co-scheduling combinations for a program A. We do not use all of these measurements as the training set and therefore split this training set into a smaller training and test set. Out of these 13 co-scheduling combinations, we choose four combinations randomly to represent our test set, and the other nine combinations are used for model training. This means that approximately 30% of the available data represents the test set, and the other 70% are used for training. With this procedure, we show that the prediction model works within itself, i.e., within similar measurements and conditions.

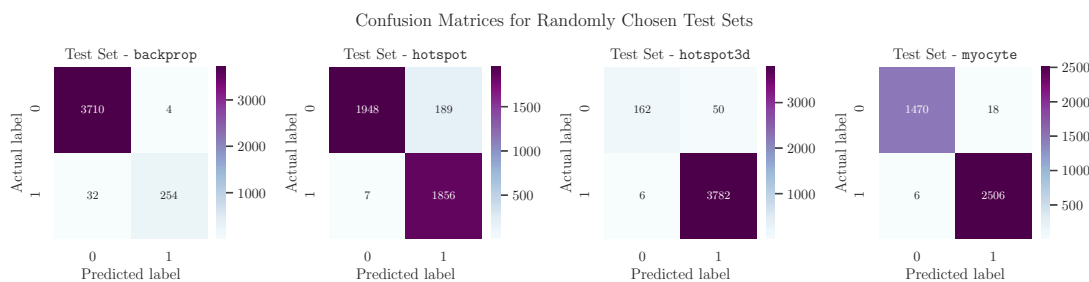


Figure 6.1: Evaluating the logistic regression model, where the test set contains randomly chosen applications from the Rodinia benchmark suite.

Figure 6.1 shows the confusion matrices of a random test set for each of the benchmarks `backprop`, `hotspot`, `hotspot3d`, and `myocyte`. We evaluate these test sets by looking at the number of true and false positives, and true and false negatives. Taking a look at the leftmost confusion matrix for `backprop`, we see that 3710 zeros and 254 ones are predicted correctly, whereas the false positives are 4, and the false negatives are 32. This is an acceptable outcome, even though the model seems biased due to the many true negatives, i.e., the correctly labeled zeros. For `hotspot`, the test set is more balanced, and there are 7 false negatives and 189 false positives. This is a small fraction compared to the correctly labeled zeros and ones. The test set for `hotspot3d` is imbalanced towards the ones, but the false negative and false positive rate is very low. Similarly, the test set for `myocyte` also shows a low false negative and false positive rate.

As a proof of concept that our randomly chosen test set is not an artifact, we created 50 models with distinct test sets, and therefore distinct training sets, for `backprop`, `hotspot`, `hotspot3d`, and `myocyte`. Table 6.1 gives an overview of minimum and maximum values, i.e., worst and best achieved precision, recall, and accuracy values.

As the best and worst evaluation metrics in Table 6.1 show, the worst precision from 200 models in total, i.e., 50 models for each one of `backprop`, `hotspot`, `hotspot3d`, and `myocyte`, is 83% for `hotspot`. The minimum precision values for `backprop`,

Table 6.1: Best and worst evaluation metrics for 50 randomly chosen test sets.

	Min. Precision	Max. Precision	Min. Recall	Max. Recall	Min. Accuracy	Max. Accuracy
backprop	0.97	1.00	0.88	0.98	0.98	1.00
hotspot	0.83	1.00	0.97	1.00	0.87	1.00
hotspot3d	0.97	1.00	0.98	1.00	0.95	1.00
myocyte	0.99	1.00	0.98	1.00	0.99	0.99

hotspot3d, and myocyte are even greater or equal to 97%, which is a good outcome. The worst recall value is 88% for *backprop* and the worst accuracy was found for *hotspot* with 87%. Overall, this is a good outcome since all minimum performance metrics are greater than 80%, and many are greater than 95%.

6.3.2 Distribution of PAPI Events between Training and Validation Sets

We are interested in the distribution differences of the PAPI performance events for our training and validation set, i.e., the *small* and *medium* sized inputs. The performance events are already normalized by time, and we compare the performance value per time ratio of the two different input sizes.

In Figures 6.2 – 6.5, we see the distribution of the PAPI performance events per time for both the dedicated and co-scheduled mode for both the training and validation set. Even though it is interesting to see what PAPI performance events per time show a different behavior between the dedicated and co-scheduled execution, we are interested in differences between the training and validation set. Our assumption and hope is that there is no difference between the training and validation set, otherwise we cannot assume that the short execution behavior with our *small* input size corresponds to the behavior of the long execution with a *medium* input size.

For the boxplots in Figures 6.2 – 6.5, we used the measurements of the 1000 dedicated and 13 000 co-scheduled executions from the *small* input size as the training set, and 500 dedicated and 6500 co-scheduled executions from the *medium* input size as the validation set.

Taking a closer look at the PAPI performance events for *backprop* in Figure 6.2, we notice no significant distribution difference between the training and validation set. Looking at the distributions of `PAPI_L3_LDM/s`, `PAPI_TOT_INS/s`, `PAPI_L3_TCM/s`, and `PAPI_L3_DCW/s`, we see a difference between the median values of the dedicated and co-scheduled executions. Still, the important observation for us is to see a similar behavior between the training and validation data.

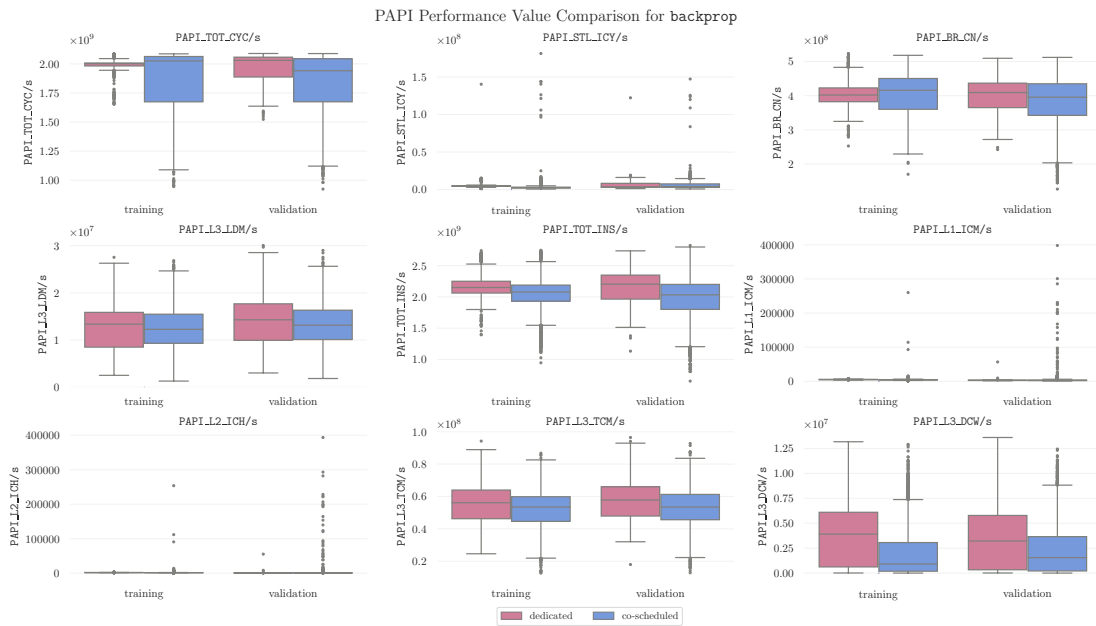


Figure 6.2: Performance event distribution comparison between the training and validation set of backprop.

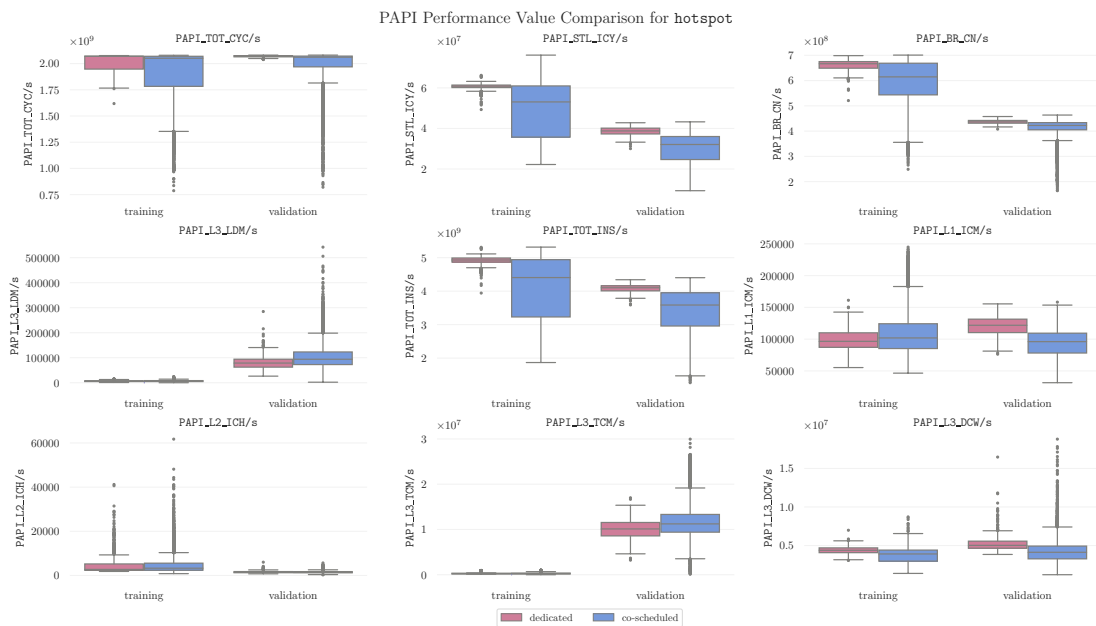


Figure 6.3: Performance event distribution comparison between the training and validation set of hotspot.

6. PREDICTING CO-SCHEDULING POTENTIALS

Figure 6.3 shows the performance event distribution of `hotspot`. We clearly see that our assumption of same distribution behaviors between the training and validation set does not hold for several performance events, e.g., `PAPI_STL_ICY/s`, `PAPI_BR_CN/s`, `PAPI_L3_LDM/s`, `PAPI_TOT_INS/s`, and `PAPI_L3_TCM/s`. The median values of both the dedicated and co-scheduled measurements differ significantly between the training and validation set.

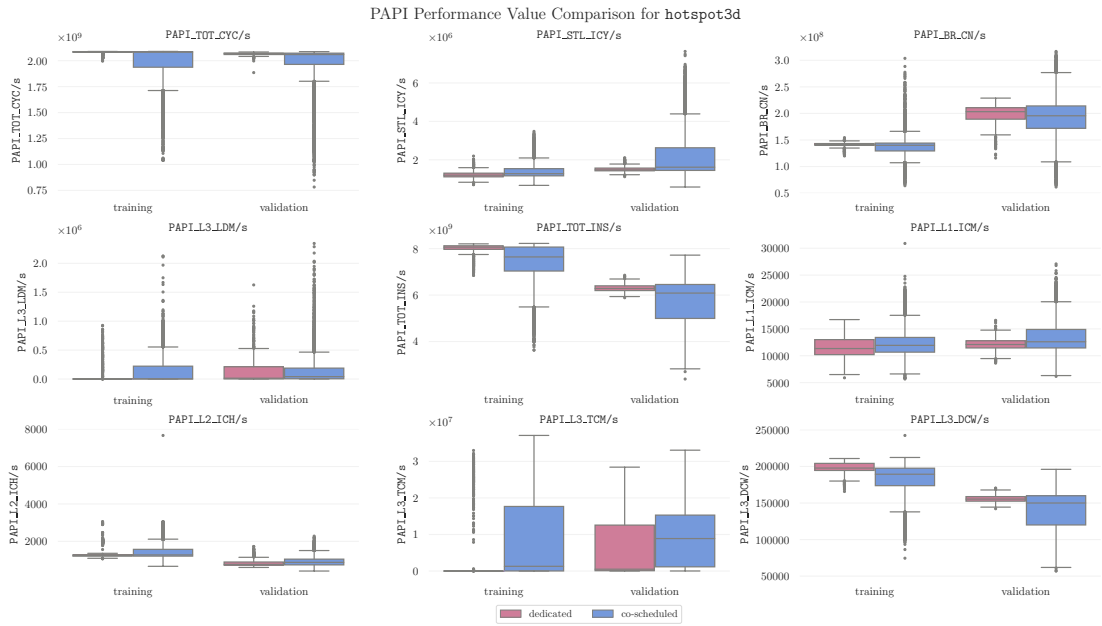


Figure 6.4: Performance event distribution comparison between the training and validation set of `hotspot3d`.

The distribution of the performance counter values of `hotspot3d` is shown in Figure 6.4. There are three performance events, `PAPI_BR_CN/s`, `PAPI_TOT_INS/s`, and `PAPI_L3_DCW/s`, which are not similarly distributed for the training and validation set executions. Comparing these non-correlating performance events to the non-correlating events from `hotspot`, we notice that these three events are a subset of the non-uniform distributions of `hotspot`.

Our fourth evaluated benchmark as program A is `myocyte`. In Figure 6.5, we see the event value distributions for `myocyte` and find similar observations as for `backprop`, i.e., similar median values between the training and validation set. Still, there are small distribution differences, as for `PAPI_TOT_CYC/s` or `PAPI_TOT_INS/s`. Comparing such differences to `hotspot` in Figure 6.3, the differences seem insignificant compared to `hotspot`.

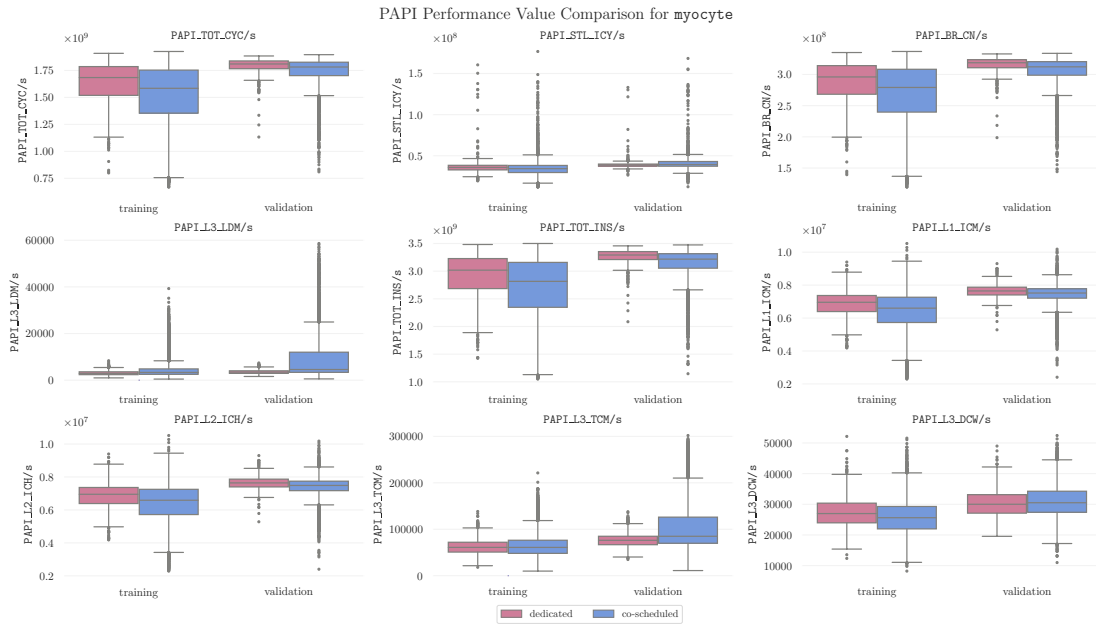


Figure 6.5: Performance event distribution comparison between the training and validation set of myocyte.

6.3.3 Evaluating the Logistic Regression Models

For each of our evaluated benchmarks `backprop`, `hotspot`, `hotspot3d`, and `myocyte`, we create a logistic regression model as shown in Appendix 7 in Listing 2. We use the *small* input sized co-scheduled runs as the training set and then validate the prediction model with the *medium* sized co-scheduled executions. We choose a threshold value of $t = 1.35$ for each application. This means that we tolerate a 35% increase in the runtime of application A.

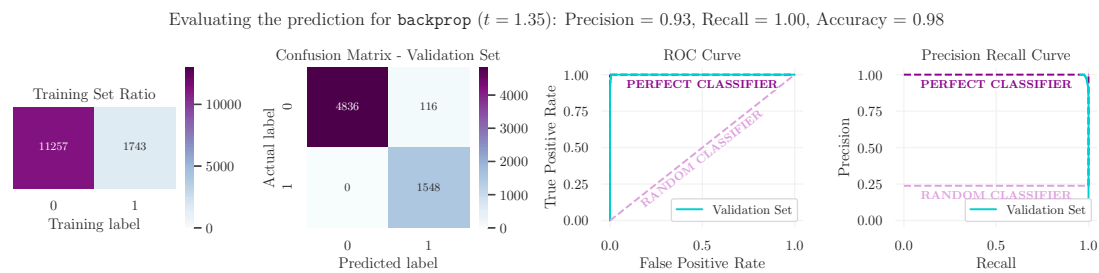


Figure 6.6: Evaluating the logistic regression model created for co-scheduling applications with `backprop`.

Figure 6.6 shows the logistic model evaluation of `backprop` being program A. Since the common threshold value is $t = 1.35$, we get an imbalanced classification ratio *zeros:ones*

of 11257 : 1743. This training data is used to train the logistic regression. Then, we validate this model by using the validation set, i.e., the *medium* input sized runs. The confusion matrix of the validation set shows a good result, since not any one-value is predicted to be a zero-value, i.e., there are no false negatives. Contrarily, some of the zeros are predicted as ones, but this corresponds to a small false positive rate. The ROC- and Precision Recall-Curves show that the model is good. The precision of this model is 0.93, the recall 1.00, and the accuracy 0.98. Therefore, we can predict the co-scheduling behavior for co-scheduled runs with `backprop` well.

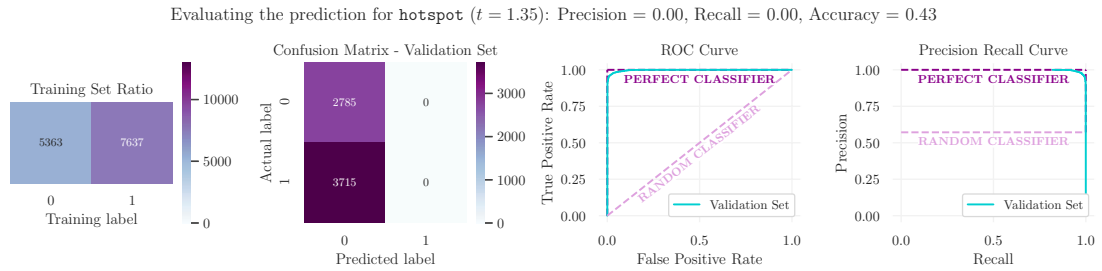


Figure 6.7: Evaluating the logistic regression model created for co-scheduling applications with `hotspot`.

For `hotspot`, we get a more balanced training set with a *zeros:ones* ratio of 5363 : 7637. As the evaluation metrics in Figure 6.7 show, the model for `hotspot` is not good: the precision and recall values are zero, the accuracy is only 43%. All entries of the validation set are predicted as zeros, i.e., that no application should be co-scheduled with `hotspot`. The ROC- and Precision Recall-Curve are distorting the bad outcome, because it seems like the model should be acceptable. This happens because the false negative rate is high, and this rate is not represented in any of these curves directly. Further, the calculated precision and recall values are both 0.00, these two values refer to the precision and recall of the model as a whole, whereas the precision recall curve consists of many precision and recall values, of whom the majority are small values near zero. Since the precision and recall of this model are zero, we see that the prediction model for `hotspot` is not good. The question arises, what may lead to this poor prediction behavior. If we look back at the distribution of the PAPI events in Figure 6.3, many event distributions do not correlate between the training and validation set. One explanation for this poor prediction behavior therefore might be the fact that the training and validation set measurements do not correlate. We train the model to behave in a certain way, but this behavior is not reflected in the validation set for several performance counters, e.g., `PAPI_STL_ICY/s`, `PAPI_BR_CN/s`, `PAPI_L3_LDM/s`, `PAPI_TOT_INS/s`, and `PAPI_L3_TCM/s`.

Therefore, one preliminary for predicting the co-scheduling behavior with a prediction model should be: the performance values used as the input for prediction have to correlate with the training data of the prediction model.

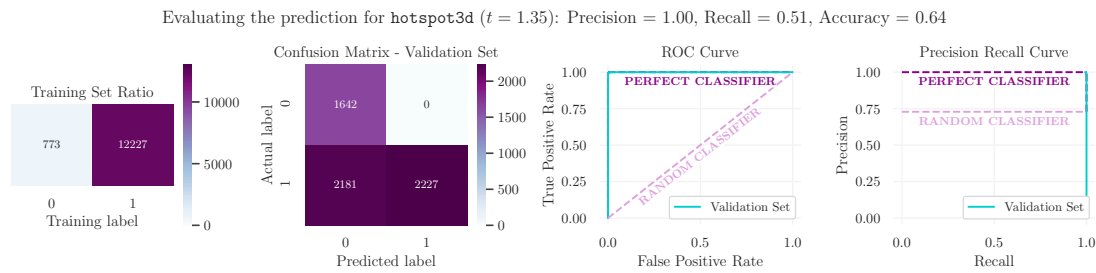


Figure 6.8: Evaluating the logistic regression model created for co-scheduling applications with `hotspot3d`.

Figure 6.8 shows the evaluation of the prediction model for `hotspot3d`. Regarding the correlation of the training and validation data set, we remember that `hotspot3d` did not correlate for all performance events similarly for the two sets. The prediction model for `hotspot3d` is highly biased towards the ones, with a ratio of 773 : 12227. Even though the precision is 100%, there are many false negatives, but no false positives. This is reflected in the recall of 51% and the accuracy of 64%. The ROC curve is again misleading, since the model seems like a good fit. But the ROC curve only considers true and false positive rates, and there are no false positives in this example. Overall, this model is not applicable for prediction. Even though the training set is biased towards one classification option, we assume that the bad outcome of the prediction still might be a result of the non-correlating performance events of the training and validation set.

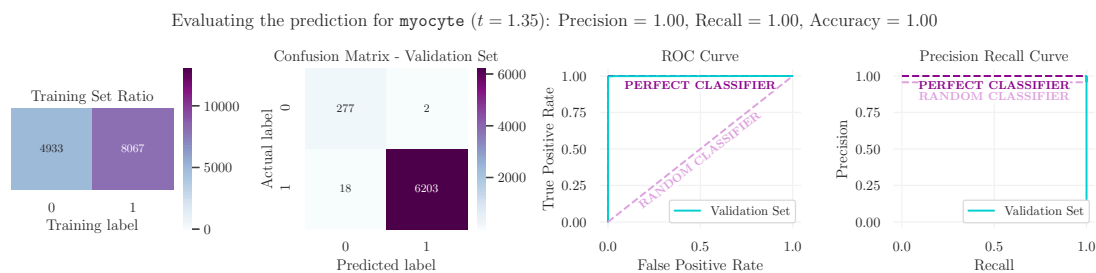


Figure 6.9: Evaluating the logistic regression model created for co-scheduling applications with `myocyte`.

In Figure 6.9, we see the model evaluation for `myocyte`, using a training set with a `zeros:ones` ratio of 4933 : 8067. Even though this training set is not perfectly balanced, the precision, recall, and accuracy values being 100% show that this model predicts the long execution runs very well with only short execution runs trained.

Using these observations, we can summarize our insights from this chapter:

- Predicting the co-scheduling potential of random applications with a specified program A is possible under certain circumstances.
- A logistic regression model suffices for predicting the co-scheduling potential.
- Normalizing performance values per time is necessary to equate the measurements $long_run_time/sec \equiv short_run_time/sec$.
- It is important to create balanced training data sets, but they do not have to be perfectly balanced. If the short execution's and long execution's performance values correlate, we can accurately predict the co-scheduling potential of two applications.
- We have to be careful with applications, where the long execution run's performance values normalized per time do not correlate with the short execution run's performance values. Such applications are not suitable for this kind of co-scheduling potential prediction since the foundation of the training data set and the actual prediction differs. Still, it is hard to discover whether the performance values of short and long executions correlate. This might only be possible if both the validation and evaluation set are large enough.

Conclusion and Future Work

The improvements in hardware design and the increasing number of cores in supercomputers demand efficient resource utilization on multicore machines. We explored scaling and runtime behaviors of various OpenMP applications from the Rodinia and SPEC OMP2012 benchmark suites. While the SPEC OMP2012 benchmark suite comes with predefined input instances, the Rodinia suite does not provide input instances. Therefore, we analyzed the Rodinia applications in detail and defined input sizes. We assessed scaling and runtime behaviors of applications on different computing resources, like sockets, and compared runtime differences of compilers. Then, we explored the optimization potential of co-scheduling, where we compared basic strategies with advanced scheduling algorithms. A basic strategy may be executing programs in parallel on all cores of a multicore machine, while integer programs represent advanced scheduling methods. For this optimization potential assessment, we used our novel execution framework, which is resource manager and scheduler for single compute nodes in one. Since there is great co-scheduling potential, we analyzed configurations of co-scheduled executions of two applications on our multicore machine *hydra*. Some configurations share close resources, which lead to increased co-scheduled times. We evaluated hardware performance counters to witness the interference of co-scheduling. Since we see this resource interference reflected in performance counter values, we use hardware counters to predict the co-scheduling potential of two applications. Assuming that there are two programs A and B with a long execution time, we predict the co-scheduling potential of A with B by sampling the co-scheduling for a short period of time. Sometimes it is possible to predict the co-scheduling potential well, sometimes not.

We learned several lessons from this thesis work:

- The majority of applications from our analysis show poor scaling behaviors, for what reason it is not necessary to execute programs in a fully parallel mode on all available cores of a system.

- Measuring kernel times of co-scheduled programs is difficult, because co-scheduled programs have different pre- and postprocessing steps. It is necessary to intervene into the co-scheduled executions by synchronizing the kernel sections. For this purpose, we implemented a lightweight synchronization library.
- A computing system provides many hardware performance counters. Choosing relevant and significant counters that can be used to predict the interference of co-scheduled applications, i.e., the co-scheduling potential of applications, is not trivial. We used several techniques to find relevant performance counters, e.g., correlation matrices and analysis of variance.
- Since we predict the co-scheduling potential of two long applications by executing and measuring performance counters for a short run, the performance measurements from both the short and long execution have to correlate with one another. Additionally, the potential slowdown of co-schedulings has to be visible in the performance counters as well. This is often hard to achieve since program parts of co-scheduled applications differ in their functionality and their execution time.

Even though we learned many lessons on application behaviors and the prediction of co-scheduling potentials, there are some shortcomings in this thesis, which point out future work:

- We explored the scalability and runtime behaviors of a limited set of applications on a limited set of hardware.
In this thesis, we use one multicore machine, but it remains to confirm our findings on other multicore machines too, to make sure that our observations are machine-independent.
Regarding the limited set of applications, we used OpenMP applications from the Rodinia and SPEC OMP2012 benchmark suite. The problem of using applications from benchmark suites is the amount of computational patterns behind the programs. For several benchmarks, the computational pattern is similar, which leads to a limited set of computational patterns. Looking at a related research topic, HPAS [9], a HPC Performance Anomaly Suite for generating anomalies on HPC systems, we can use their idea of generating anomalies to generate computational patterns. Further, we would create applications with specific computational patterns, e.g., programs with many floating point operations, or programs with many cache misses. Having such a generator for computational patterns would moreover control the model training of prediction models.
- For our prediction model, we were satisfied with a binary decision: whether to co-schedule applications or not. The logistic regression used for this binary classification problem is a sufficient first basis. To create prediction models without biased decisions, it is necessary to use more advanced machine learning methods. Additionally, we trained our models with performance counter interferences from short executions because we validated the prediction potential using long executions. But precisely these long executions should be used for training the prediction model to cover a wider range of co-scheduling behaviors.

Bibliography

- [1] Intel Developer Guide and Reference - Thread Affinity Interface (Linux* and Windows*). <https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html>, 2021, accessed on 2021-08-20.
- [2] Intel® Xeon® Scalable Processors. <https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html>, accessed on 2021-08-15.
- [3] About GROMACS. https://www.gromacs.org/About_Gromacs, accessed on 2021-08-28.
- [4] Software Modules. <https://apps.fz-juelich.de/jsc/hps/juwels/software-modules.html>, accessed on 2021-08-28.
- [5] LAMMPS Molecular Dynamics Simulator . <https://www.lammps.org/>, accessed on 2021-08-28.
- [6] LAMMPS. https://wiki.vsc.ac.at/doku.php?id=doku:molecular_dynamics#lammps, accessed on 2021-08-28.
- [7] Vienna Scientific Cluster. <https://vsc.ac.at>, accessed on 2021-08-28.
- [8] VSC-4. <https://vsc.ac.at/systems/vsc-4>, accessed on 2021-08-28.
- [9] E. Ates, Y. Zhang, B. Aksar, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019*, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. doi: 10.1145/3337821.3337907.
- [10] S. Benedict, P. Gschwandtner, and T. Fahringer. TOEP: Threshold Oriented Energy Prediction Mechanism for MPI-OpenMP Hybrid Applications*. In *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pages 1–6, 2018. doi: 10.1109/IC3.2018.8530575.

- [11] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [12] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.
- [13] J. Breitbart, J. Weidendorfer, and C. Trinitis. Case Study on Co-scheduling for HPC Applications. In *2015 44th International Conference on Parallel Processing Workshops*, pages 277–285, 2015. doi: 10.1109/ICPPW.2015.38.
- [14] J. Breitbart, S. Pickartz, S. Lankes, J. Weidendorfer, and A. Monti. Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017. doi: 10.1109/CLUSTER.2017.59.
- [15] A. D. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D. M. Tullsen, and A. E. Snaveley. The Case for Colocation of High Performance Computing Workloads. *Concurr. Comput.: Pract. Exper.*, 28(2):232–251, Feb. 2016. ISSN 1532-0626. doi: 10.1002/cpe.3187.
- [16] J. Brownlee. *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*. Machine Learning Mastery, 2020.
- [17] P. Brucker. *Scheduling algorithms*. Springer, Berlin [u.a.], 5. ed.. edition, 2007. ISBN 354069515X.
- [18] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP : portable shared memory parallel programming*. Scientific and engineering computation. MIT Press, Cambridge, Mass. [u.a.], 2008. ISBN 0262533022.
- [19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
- [20] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, 2010.
- [21] R. D. Cook and S. Weisberg. Criticism and Influence Analysis in Regression. *Sociological Methodology*, 13:313–361, 1982. ISSN 00811750, 14679531.
- [22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambridge, Mass. [u.a.], 3. ed.. edition, 2009. ISBN 0262033844.

- [23] T. Creech, A. Kotha, and R. Barua. Efficient Multiprogramming for Multicores with SCAF. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46*, page 334–345, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450326384. doi: 10.1145/2540708.2540737.
- [24] J. Davis and M. Goadrich. The Relationship between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning, ICML '06*, page 233–240, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143874. URL <https://doi.org/10.1145/1143844.1143874>.
- [25] M. Endrei, C. Jin, M. N. Dinh, D. Abramson, H. Poxon, L. DeRose, and B. R. de Supinski. Statistical and machine learning models for optimizing energy in parallel applications. *The International Journal of High Performance Computing Applications*, 33(6):1079–1097, 2019. doi: 10.1177/1094342019842915.
- [26] G. Georgakoudis, H. Vandierendonck, P. Thoman, B. R. D. Supinski, T. Fahringer, and D. S. Nikolopoulos. SCALO: Scalability-Aware Parallelism Orchestration for Multi-Threaded Workloads. 14(4), Dec. 2017. ISSN 1544-3566. doi: 10.1145/3158643.
- [27] T. Gruber. likwid perfctr. <https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr>, 2020, accessed on 2020-08-27.
- [28] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327046. doi: 10.1145/2592798.2592807. URL <https://doi.org/10.1145/2592798.2592807>.
- [29] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009. ISBN 9780387848846.
- [30] S. Hatfield. Supercomputer trend data. <https://samhatfield.co.uk/2019/04/10/supercomputer-trend-data/>, accessed on 2021-08-28.
- [31] A. H. Karp and H. P. Flatt. Measuring Parallel Processor Performance. *Commun. ACM*, 33(5):539–543, May 1990. ISSN 0001-0782. doi: 10.1145/78607.78614. URL <https://doi.org/10.1145/78607.78614>.
- [32] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. ADAPT: A Framework for Coscheduling Multithreaded Programs. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400704. URL <https://doi.org/10.1145/2400682.2400704>.
- [33] M. Mercier, D. Glesser, Y. Georgiou, and O. Richard. Big data and HPC collocation: Using HPC idle resources for Big Data analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 347–352, 2017. doi: 10.1109/BigData.2017.8257944.

- [34] M. Mirka, G. Sassatelli, and A. Gamatié. Online Learning for Dynamic Control of OpenMP Workloads. In *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCASST)*, pages 1–6, 2020. doi: 10.1109/MOCASST49295.2020.9200292.
- [35] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran. SPEC OMP2012 – an Application Benchmark Suite for Parallel Systems Using OpenMP. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP’12*, page 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.
- [36] NERSC. Process and Thread Affinity. <https://docs.nersc.gov/jobs/affinity/>. Accessed on 2021-08-20.
- [37] NERSC. Measuring Arithmetic Intensity. https://docs.nersc.gov/performance/arithmetric_intensity/, accessed on 2021-08-17.
- [38] K. O’Brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou. A Survey of Power and Energy Predictive Models in HPC Systems and Applications. *ACM Comput. Surv.*, 50(3), June 2017. ISSN 0360-0300. doi: 10.1145/3078811.
- [39] F. Pascual and K. Rzacca. Colocating tasks in data centers using a side-effects performance model. *European Journal of Operational Research*, 268, 02 2018. doi: 10.1016/j.ejor.2018.01.046.
- [40] M. Pinedo. *Scheduling : theory, algorithms, and systems*. Springer, Cham, fifth edition. edition, 2016. ISBN 3319265806.
- [41] S. Raschka and V. Mirjalili. *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow, 2nd Edition*. Packt Publishing, 2nd edition, 2017. ISBN 1787125939.
- [42] K. R. S. Hatfield. Microprocessor (and Supercomputer) Trend Data. <https://github.com/samhatfield/microprocessor-trend-data>, accessed on 2021-09-05.
- [43] D. Sarkar, R. Bali, and T. Sharma. *Practical Machine Learning with Python: A Problem-Solver’s Guide to Building Real-World Intelligent Systems*. Apress, USA, 1st edition, 2017. ISBN 1484232062.
- [44] Y. Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC Computational Science Series. Chapman and Hall/CRC, an imprint of Taylor and Francis, Boca Raton, FL, 1st edition. edition, 2015. ISBN 0429069413.
- [45] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors,

Tools for High Performance Computing 2009, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11261-4.

- [46] Z. Wang, L. Zheng, Q. Chen, and M. Guo. CAP: Co-Scheduling Based on Asymptotic Profiling in CPU+GPU Hybrid Systems. PMAM '13, page 107–114, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319089. doi: 10.1145/2442992.2443004. URL <https://doi.org/10.1145/2442992.2443004>.
- [47] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. HotSpot: a compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5): 501–513, 2006. doi: 10.1109/TVLSI.2006.876103.
- [48] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun. Tangram: Colocating HPC Applications with Oversubscription. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2018. doi: 10.1109/HPEC.2018.8547644.
- [49] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39727-4.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Appendices

Code Template for Measuring PAPI Performance Events

For measuring our limited set of PAPI performance events, we use the PAPI low level API. Listing 1 shows the synchronization and PAPI calls that need to be added around the kernel section of a program.

Listing 1: Template for measuring PAPI events.

```

1  #include "sync.h"
2  #include <papi.h>
3
4  /* everything before kernel section: preprocessing, etc. */
5
6  int EventSet=PAPI_NULL;
7  int events[] = {PAPI_TOT_CYC, PAPI_STL_ICY, PAPI_BR_CN, PAPI_L3_LDM, PAPI_TOT_INS,
8                PAPI_L1_ICM, PAPI_L2_ICH, PAPI_L3_TCM, PAPI_L3_DCW};
9  int num_events = sizeof(events) / sizeof(events[0]);
10 long_long values[num_events];
11
12 init_sync();
13
14 if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) { /* error handling */}
15 if (PAPI_create_eventset(&EventSet) != PAPI_OK) { /* error handling */}
16 if (PAPI_add_events(EventSet, events, num_events) != PAPI_OK) { /* error handling */}
17
18 do_sync();
19
20 if (PAPI_start(EventSet) != PAPI_OK) { /* error handling */}
21 /* kernel section */
22 if (PAPI_read(EventSet, values) != PAPI_OK) { /* error handling */}
23
24 do_sync();
25
26 for (int i = 0; i < num_events; i++) {
27     char EventCodeStr[PAPI_MAX_STR_LEN];
28     if (PAPI_event_code_to_name(events[i], EventCodeStr) != PAPI_OK) {
29         /* error handling */
30     }
31     printf("%s: %lld\n", EventCodeStr, values[i]);
32 }
33
34 if (PAPI_stop(EventSet, values) != PAPI_OK) { /* error handling */}
35
36 cleanup_sync();
37 /* postprocessing etc. */
    
```

Pseudo Code of the Prediction Model

Listing 2: Pseudo code of the logistic regression model.

```
1 from sklearn.linear_model import LogisticRegression
2 import pandas as pd
3
4 def create_good_bad_column(df, app):
5     # min_time is minimum dedicated time of app, t is threshold for app
6     good_bad_col = []
7     for i, row in df.iterrows():
8         if row['time'] <= t * min_time:
9             good_bad_col.append(1)
10        else:
11            good_bad_col.append(0)
12        df['good-time'] = good_bad_col
13
14 def read_in_data_create_good_bad_column(size):
15     all_dataframes = []
16     # read in all measurements from co-scheduled combinations with size
17     for a in all_apps:
18         df = pd.read_csv(filename_for_a)
19         df = create_good_bad_column(df, a)
20         all_dataframes.append(df)
21     return pd.concat(all_dataframes)
22
23 def normalize_training_values(df, cols):
24     # normalizes values per time, then uses robust scaling
25     # values for robust scaling are taken from df[x],
26     where x is a performance event in cols
27
28 def normalize_training_values(df_train, df_validation, cols):
29     # normalizes values from df_validation per time, then uses robust scaling
30     # values for robust scaling are taken from df_train[x],
31     where x is a performance event in cols
32
33 def main():
34     # feature_cols is an array of all PAPI performance events used
35
36     dataframe_training = read_in_data_create_good_bad_column('small')
37     dataframe_validation = read_in_data_create_good_bad_column('medium')
38
39     normalize_training_values(dataframe_training, feature_cols)
40     normalize_validation_values(dataframe_training, dataframe_validation,
41                               feature_cols)
42
43     feature_cols = ['s-norm/s' % x for x in feature_cols]
44
45     X = dataframe_training[feature_cols]
46     y = dataframe_training['good-time']
47
48     random_state = 5
49     log_reg = LogisticRegression(solver='liblinear', random_state=random_state,
50                                max_iter=100)
51     log_reg.fit(X, y)
52
53     X_validation = dataframe_validation[feature_cols]
54     y_validation = dataframe_validation['good-time']
55     y_pred_validation = log_reg.predict(X_validation)
```