Alcides Miguel Cachulo Aguiar Fonseca

Automatic Optimization of Granularity Control Algorithms for Parallel Programs

Doctoral thesis submitted to the Doctoral Program in Information Science and Technology,
supervised by Assistant Professor Bruno Cabral,
and presented to the Department of Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra.

September 2016

Universidade de Coimbra

Automatic Optimization of Granularity Control Algorithms for Parallel Programs

A thesis submitted to the University of Coimbra
in partial fulfillment of the requirements for the
Doctoral Program in Information Science and Technology

by

Alcides Miguel Cachulo Aguiar Fonseca
alcides.fonseca@uc.pt

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

September 2016

Financial support by
Fundação para a Ciência e a Tecnologia
Ref.: SFRH/BD/84448/2012
Automatic Optimization of Granularity Control Algorithms for Parallel Programs
©2016 Alcides Fonseca

****
****

Advisor

Prof. Bruno Cabral

Assistant Professor
Department of Informatics Engineering
Faculty of Sciences and Technology of University of Coimbra

Dedicated to my family

Abstract

In the last two decades, processors have changed from a single-core to a multi-core design, due to physical constrains in chip manufacturing. Furthermore, GPUs have become suitable targets for general purpose programming. This change in hardware design has had an impact on software development, resulting in a growing investment in parallel programming and parallelization tools.

Writing parallel programs is difficult and error prone. Two of the main problems in parallelization are the identification of which sections of the code can be safely parallelized and how to efficiently partition work. Automatic parallelization techniques can save programmers time identifying parallelism. In parallelization, each parallelizable section is denoted as a task, and a program is comprised of several tasks with dependencies among them. Work partition consists in deciding how many tasks will be created for a given parallel workload, thus defining the task granularity. Current techniques focus solely on loop and recursive parallelization, neglecting possible fine-grained task-level parallelism. However, if the granularity is too fine, penalizing scheduling overheads may be incurred. On the other hand, if the granularity is too coarse, there may not be enough parallelism in the program to occupy all processor cores. The ideal granularity of a program is influenced by its nature and the available resources. Our experiments have shown that a program that terminates within seconds with the correct granularity may execute for days with an unsuitable granularity. Finding the best granularity is not trivial, more so in the case of automatic parallelization, in which there is no knowledge of the program domain. The current approach consists in empirically evaluating several alternatives to find the optimal granularity.

This thesis proposes a more efficient model for automatic parallelization, in which parallelism is identified at the Abstract Syntax Tree (AST) node level. Static analysis is used to obtain access permissions, representations of how an AST node interacts with others in terms of memory accesses and control-flow. Parallelism at the AST node level is very fine grained and may generate more tasks than those that can be executed simultaneously, resulting in scheduling overheads. In order to reduce these overheads, tasks may be merged in coarser tasks, thus reducing parallelism. A cost-model is proposed to dynamically adjust granularity according to the complexity of tasks, resulting in programs more efficient than the best existing alternative.

Because the automatic parallelization model can generate programs that can execute either on the CPU or the GPU, it is important to automatically decide if a program should execute on the CPU with a coarse granularity, or on the GPU with a finer granularity. To perform this decision, a Machine Learning approach was built, based on static compiler-obtained and runtime features. This model performed

program classification with over 95% of accuracy and a low misclassification cost.

In order to improve the performance of automatic and manually parallelized programs, new dynamic granularity algorithms are proposed for runtime aggregation of tasks. The proposed algorithms extend the state of the art by taking into consideration the usage of the number of stack frames and machine occupation, as well as using a cost-model-based prediction of the task execution time. The existing and proposed algorithms have been evaluated in both time and energy consumed, as well as number of programs completed within reasonable time. Considering both time and energy, the proposed algorithms outperformed existing ones, but no algorithm performed better than any other in all benchmark programs. These results demonstrate the importance of using the right algorithm for an individual program.

An evolutionary algorithm was used to generate a global best granularity algorithm for a set of target programs. While improvements were not generalized to a larger set of programs, the evolutionary algorithm can be used to improve the execution time within 10 to 20 generations.

To avoid an exhaustive search for the best granularity algorithm for each program, this thesis proposes both a ruleset and the usage of machine-learning classifiers over program features. The ruleset was obtained from the empirical evaluation of different alternatives on a selected benchmark suite. Both approaches can be used by compilers or programmers to select the granularity algorithm for each program. In a real-world benchmark suite, the ruleset has shown to outperform classifiers, but on an unseen larger synthetic benchmark suite, a misclassification-weighted Random Forest was able to achieve better results than the ruleset.

Overall, this thesis proposes new approaches for automatic parallelization and granularity control that improve the performance of programs.

Keywords: automatic parallelization, compilers, concurrency, work-stealing, optimization

Resumo

Durante as últimas duas décadas, o design dos processadores mudou de um único núcleo para *multicore*, devido a limitações físicas no processo de fabrico. Para além disso, GPUs têm também sido usadas para programas generalistas e não só de aplicações gráficas. Esta mudança em design de hardware tem tido um impacto grande no desenvolvimento de software, resultando num investimento crescente em programação paralela e ferramentas para tal.

Escrever programas paralelos é difícil e sujeitável a erros. Dois dos principais problemas em paralelização são a identificação de secções de código que podem ser executas em paralelo sem causar erros, e como dividir o trabalho eficientemente. Técnicas de paralelização automática podem poupar tempo aos programadores na identificação de paralelismo. Em paralelização, cada secção que corre em paralelo é chamada de tarefa, e um programa é composto por diferentes tarefas com dependências entre elas. Partição de trabalho consiste em decidir quantas tarefas vão ser criadas para um determinado trabalho, definindo a granularidade de tarefas. As técnicas actuais focam-se em paralelização automática de ciclos e recursividade, ignorando paralelismo fino ao nível da tarefa. No entanto, com uma granularidade demasiado fina, existem custos no escalonamento de tarefas. Mas se a granularidade for demasiado grossa, pode não existir paralelismo para ocupar todos os núcleos do processador. A granularidade ideal para um programa é influenciada pela sua natureza e pelos recursos disponíveis. Nas nossas experiências, um programa que termine em segundos com a granularidade certa, pode demorar dias com uma granularidade menos própria. Encontrar a granularidade ideal não é trivial, especialmente em casos de paralelização automática, em que não existe conhecimento do domínio do programa. A abordagem actual consiste em empiricamente avaliar diferentes alternativas.

Esta tese propõe um modelo de paralelização automática mais eficiente, em que o paralelismo é identificado ao nível dos nós da Árvore de Sintaxe Abstracta (AST). Análise estática é usada para obter *access permissions*, representações de como os nós interagem com outros em termos de acessos de memória e fluxo de controlo. Paralelismo a este nível é muito fino e pode executar mais tarefas do que as que podem ser executadas eficientemente em paralelo. Para reduzir os overheads causados, tarefas podem ser agregadas em tarefas maiores, reduzindo o paralelismo. Um modelo de custo é proposto para ajustar dinamicamente a granularidade de acordo com a complexidade das tarefas, resultando em programas mais eficientes do que com as alternativas actuais.

Como este modelo pode gerar programas que podem executar na GPU ou no CPU, é importante tomar a decisão em que plataforma correr. Um programa com uma granularidade mais grossa deve executar na CPU, enquanto um programa com a

granularidade mais fina pode executar na GPU. Para fazer esta decisão, é proposta uma abordagem de *Machine Learning*, baseado em análise estática e em *features* obtidas pelo *Runtime*. Este modelo conseguiu uma precisão de 95% e um baixo custo de classificação errada.

Para melhorar a performance de programas paralelos, tanto manuais como automáticos, são propostos novos algoritmos de controlo de granularidade para agregação de tarefas pelo *Runtime*. Os algoritmos propostos estendem o estado da arte tendo em conta o uso de *stack frames* e ocupação da máquina. Os algoritmos existentes e propostos foram todos avaliados tanto a nível de tempo de execução, como de energia consumida, bem como número de programas terminados num determinado tempo. Considerado tempo e energia, os algoritmos propostos conseguiram ser melhores que os existentes, mas nenhum algoritmos conseguiu ser melhor que todos os outros em todos os programas testados. Estes resultados demostram como é importante escolher o algoritmo ideal para cada programa.

Um algoritmo evolucionário é também proposto para gerar um melhor algoritmo de granularidade para um conjunto de programas alvo. Apesar das melhorias não serem generalizáveis para um conjunto maior de programas, o algoritmo evolucionário pode ser usado para melhorar o tempo de execução entre 10 a 20 gerações.

Para evitar uma pesquisa exaustiva pelo melhor algoritmo de granularidade para cada programa, são propostos um conjunto de regras e classificadores de *Machine-Learning*. O conjunto de regras foi obtido através da análise empírica de diferentes algoritmos num conjunto de programas. Ambas as abordagens podem ser usadas por compiladores ou programadores para escolher o algoritmo de granularidade para cada programa. Num conjunto de programas reais, o conjunto de regras mostrou melhores resultados que os classificadores. Em novos programas criados sinteticamente, um classificador *Random Forest*, usando pesos baseados no custo de classificação errada, obteve resultados melhores que o ruleset.

Resumindo, esta tese propõe novas abordagens para paralelização automática e controlo de granularidade que podem melhorar a performance de programas.

Acknowledgements

First of all, I would like to thank Bruno for advising me through out this 5 year journey. Bruno has been really strict with the quality of our research output. On one hand, this meant that I had to spend countless hours rewriting the same text over and over, on the other hand, it meant that my work would be raised to higher standards. His home-cooked dinners were also quite delicious!

I am thankful to my PhD colleagues who have shared together with me the adventure of signing up for a low-income never-ending not-even-a-job adventure. I would not have gone through without the help and friendship of Nuno, Filipe, Andreia, Marco, Ivo, Pedro, Maryam and Panda.

Despite the fact that researchers think so abstractly that they do not care about practical applications, I had friends in the industry that kept me grounded to the reality. I'm talking about Rui, Miguel, Diogo and David, *o gangue dos almoços*. I am also including João Nabais, who gave me the pleasure of being his personal human debugger.

And because life is not just computer science, I would also like to thank all my other friends who had the pleasure of having to put up with me. This list includes, in no particular order, João Cotrim, the Barbeiro brothers, Carla, Rafaela, Steve, Teresa, Luís and Sílvia, Diana, Paulo, Ivano, Melissa, Ana Luísa and a few others. Finally, a very special thanks to Marta mainly because she had to put up with me the most, and everyone knows that is not an easy task.

The most important thank you note goes to my family, my parents and sister. They have provided an education more important than any academic degree. I am deeply grateful for everything that they have done for me.

Finally, the research that led to this thesis would not have been possible without the funding and support provided by the Fundação para a Ciência e Tecnologia under the scholarship SFRH/BD/84448/2012, by the CMU-Portugal program Aeminium (CMU-PT/SE/0038/2008) and the iCIS program (CENTRO-07-ST24-FEDER-002003), and by the Centre for Informatics and Systems of University of Coimbra (CISUC - R&D Unit 326/97).

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Thesis Structure . . . 4

2 State of the Art **7**
2.1 Parallelization . . . 7
2.1.1 Manual parallelization tools . . . 7
2.1.2 Automatic parallelizing compilers . . . 9
2.1.3 Parallel-by-default programming languages . . . 13
2.2 Parallel program execution and optimization . . . 15
2.2.1 Runtime task schedulers . . . 15
2.2.2 Granularity Control . . . 15
2.2.3 Evaluations of Cut-off Algorithms . . . 18
2.2.4 GPU execution of parallel programs . . . 19
2.3 Energy consumption of Parallel Programs . . . 20
2.4 Summary . . . 22

3 Automatic Parallelization with Static and Dynamic Granularity Control **23**
3.1 Introduction . . . 23
3.2 Methodology . . . 24
3.2.1 Signature Extraction . . . 25
3.2.2 Parallelization . . . 26
3.3 Implementation . . . 30
3.3.1 Automatic Parallelization . . . 31
3.3.2 Runtime Execution Support . . . 32
3.4 A Cost-Model Granularity Approach . . . 34
3.5 Evaluation . . . 36
3.5.1 Benchmark Programs . . . 37
3.5.2 Binary vs Lazy Binary Splitting . . . 37
3.5.3 Nested Loops . . . 38
3.5.4 Invocation Inside Loops . . . 38
3.5.5 Recursive Calls . . . 40
3.6 Conclusions . . . 45

4 Automatic Selection of GPU-CPU Granularity 47
4.1 Introduction . . . 47
4.2 ÆminiumGPU Architecture . . . 49
4.2.1 ÆminiumGPU Compiler . . . 49
4.2.2 ÆminiumGPU Runtime . . . 50
4.3 A Machine Learning Approach for GPU-CPU Decision . . . 50
4.4 Evaluation and Classifier Selection . . . 52
4.4.1 Dataset . . . 52
4.4.2 Experimental Setup . . . 54
4.4.3 Feature analysis . . . 54
4.4.4 Classifier Comparison . . . 54
4.5 Related Work . . . 57
4.6 Conclusions . . . 58

5 The No Free Lunch Theorem for Granularity Algorithms 59
5.1 Introduction . . . 59
5.2 Cut-off Algorithms . . . 60
5.3 Benchmark Suite . . . 61
5.4 Experimental Setup . . . 62
5.5 Comparing Cut-off Algorithms . . . 63
5.6 Verifying the No Free Lunch theorem . . . 69
5.7 Conclusions . . . 71

6 Energy Efficiency of Granularity Control Algorithms 73
6.1 Introduction . . . 73
6.2 Methodology . . . 74
6.2.1 Synthetic Benchmark . . . 74
6.2.2 Real-world Benchmark . . . 75
6.2.3 Experimental Setup . . . 75
6.3 Results . . . 76
6.3.1 Time versus Energy . . . 76
6.3.2 Balanced programs . . . 77
6.3.3 Unbalanced programs . . . 81
6.3.4 Real-world programs . . . 82
6.4 Conclusions . . . 84

7 Using Evolutionary Algorithms to Optimize Granularity Algorithms 87
7.1 Introduction . . . 87
7.2 Parallel Program Configuration . . . 88
7.3 An Evolutionary Algorithm for Parallel Program Optimization . . . 89
7.3.1 Genotype . . . 89
7.3.2 Operators and general configurations . . . 90
7.3.3 Fitness Evaluation . . . 91
7.3.4 Selection operator . . . 92
7.4 Evaluation . . . 92
7.4.1 Experimental Settings . . . 92
7.4.2 Training dataset . . . 93
7.4.3 Testing dataset . . . 95
7.4.4 Evolving single programs . . . 101

7.5 Conclusions 102

8 Selection of Granularity Control Algorithms 103

8.1 Introduction 103
8.2 Approaches for Automatic Granularity Algorithm Selection 104
8.2.1 A Static Approach 104
8.2.2 A Feature-based Ruleset 106
8.2.3 A Machine Learning Approach 108
8.3 Evaluation 111
8.3.1 Methodology 111
8.3.2 Results 112
8.4 Conclusions 115

9 Conclusions and future work 117

9.1 Overview 117
9.2 Future Work 119

List of Figures

3.1 Information Flow in the Æminium Framework 31
3.2 JPar Compiler Phases 31
3.3 Example of a task graph. 32
3.4 Runtime Areas for the different phases of the Task Lifecycle, for a quad-core machine. Tasks go through the states from left to right. . . 33
3.5 Average speedup of Binary Split and Lazy Binary Split (PPS=3 and 10) versions of the programs with loops. 38
3.6 NBody execution on the machine *server24* 39
3.7 Pi execution on the machine *server24* 39
3.8 CPU and Memory usages of Pi on the machine *server24* in JPar and OoOJava 40
3.9 BlackScholes execution on the machine *server24* 41
3.10 FFT execution on the machine *server24* 41
3.11 CPU and Memory usages of FFT on the machine *server24* in JPar and OoOJava 42
3.12 Integrate execution on the machine *server24* 43
3.13 Integrate execution on the machine *server32* 43
3.14 CPU and Memory usages of Integrate on the machine *server24* in JPar and OoOJava 44
3.15 MergeSort execution on the machine *server24* 44
3.16 CPU and Memory usages of MergeSort on the machine *server24* in JPar and OoOJava 45

4.1 Performance of the Integral program on CPU and GPU 48
4.2 Architecture of ÆminiumGPU 49
4.3 Accuracy of different classifiers for GPU-CPU decision. 56
4.4 Misclassification cost of different classifiers for GPU-CPU decision. . 57

5.1 Speedup of different cut-off mechanisms used in different programs compiled with the JPar compiler, implementing the approach presented in Chapter 3. 60
5.2 Swarm plot of different cut-off approaches for the Do-all program on the *server24* machine. 63
5.3 Swarm plot of different cut-off approaches for the Matrix Multiplication program on the *server32* machine. 64
5.4 Swarm plot of different cut-off approaches for the Fibonacci program on the *server24* machine. 64
5.5 Swarm plot of different cut-off approaches for the Integrate program on the *server24* machine. 65

5.6 Swarm plot of different cut-off approaches for the N-Queens program on the *server24* machine. . . . 65
5.7 Swarm plot of different cut-off approaches for the FFT program on the *server24* machine. . . . 66
5.8 Swarm plot of different cut-off approaches for the Raytracer program on the *server24* machine. . . . 67
5.9 Swarm plot of different cut-off approaches for the Neural Network training program on the *server32* machine. . . . 67
5.10 Swarm plot of different cut-off approaches for the KDTree training program on the *server32* machine. . . . 68
5.11 Cut-off performance in the Fibonacci program with 49 as input on *server24* . . . 70
5.12 Cut-off performance in the Fibonacci program with 51 as input on *server24* . . . 71

6.1 Example of a task tree, generated by a possible program . . . 74
6.2 Distribution between duration of the program and its energy usage. . . 76
6.3 Time and energy relative performance for different task workloads. Higher is better. . . . 77
6.4 Time and energy relative performance for different leaf tasks workloads. Higher is better. . . . 78
6.5 Time and energy relative performance for different depths with leaf workloads. Higher is better. . . . 79
6.6 Time and energy relative performance for different branching factors with leaf workloads. Higher is better. . . . 80
6.7 Time and energy relative performance for different depths with unbalanced binary branching. Higher is better. . . . 81
6.8 Time and energy relative performance in irregular real-world benchmarks. Higher is better. . . . 82
6.9 Time and energy relative performance in real-world benchmarks where the best time-efficient cut-off is not the most energy-efficient. Higher is better. . . . 83
6.10 Evaluation of different cut-off approaches over the whole real-world dataset. Blue bars represent the ratio of time spent by that cut-off that could have been save by using the best cut-off for each program. Green bars represent the same ratio, but for energy. Red bars represent the ratio of programs that could not execute within the 1 hour threshold. Lower is better. . . . 84

7.1 Fitness of the best individual of each generation of the training dataset on *server8*. . . . 93
7.2 Fitness of the best individual of each generation of the training dataset on *server24*. . . . 94
7.3 Fitness of the best individual of each generation of the training dataset on *server32*. . . . 94
7.4 Execution time per training program of the best individual of each generation of the training dataset on *server8*. . . . 95
7.5 Execution time per training program of the best individual of each generation of the training dataset on *server24*. . . . 96

7.6 Execution time per training program of the best individual of each generation of the training dataset on *server32*. 96
7.7 Fitness in the testing dataset of the best individual of each generation of the training dataset on *server8*. 97
7.8 Fitness in the testing dataset of the best individual of each generation of the training dataset on *server24*. 97
7.9 Fitness in the testing dataset of the best individual of each generation of the training dataset on *server32*. 98
7.10 Execution time per testing program of the best individual of each generation of the training dataset on *server8*. 99
7.11 Execution time per testing program of the best individual of each generation of the training dataset on *server32*. 100
7.12 Fitness of the best individual over different instances of GA, one for each individual program. 101

8.1 Misclassification cost for always selecting each algorithm on *server24* machine. 104
8.2 Misclassification cost for always selecting each algorithm on *server32* machine. 105
8.3 Number of programs completed in both machines per cut-off mechanism. 105
8.4 A distribution of each of the features per best cut-off approach, using the upper bound metric on *server32*. 107
8.5 The misclassification time of programs by the ForLoop feature, on *server32*, considering the mean time of each program. 107
8.6 The misclassification time of programs by the Unbalance feature, on *server32*, considering the mean time of each program. 108
8.7 The misclassification time of programs by the number of kernels, on *server32*, considering the mean time of each program. 109
8.8 The misclassification time of programs by the branching factor, on *server32*, considering the mean time of each program. 110
8.9 The misclassification time of programs by the number of nesting loops, on *server32*, considering the mean time of each program. . . . 111
8.10 Misclassification cost of each decision mechanism over the training dataset, on *server32*. 112
8.11 Misclassification energy cost of each decision mechanism over the training dataset, on *server32*. 113
8.12 Misclassification cost of each decision mechanism on the testing dataset, on *server32*. 114
8.13 Misclassification energy cost of each decision mechanism on the testing dataset, on *server32*. 115

List of Tables

3.1 Description of the programs used in the benchmark 37

4.1 List of features 51
4.2 Features rank using a forest of trees 55

5.1 Description of the programs used in the benchmark. 62
5.2 Details of the hardware used in the experiments. 62
5.3 Mean ranking of cut-off algorithms on the full benchmark on *server32* 69
5.4 Mean ranking of cut-off algorithms on the full benchmark on *server24* 70

7.1 Possible Cut-off Variables 90
7.2 Genotype 91
7.3 Description of the programs used in the benchmark 92
7.4 Details of the hardware used in the experiments. 93

List of Abbreviations

Abbreviation	Meaning
AST	Abstract Syntax Tree
ATC	Adaptive Tasks Cut-Off
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Language
DVFS	Dynamic Voltage and Frequency Scaling
FP	Functional Programming
GA	Genetic Algorithm
GP	Genetic Programming
GPU	Graphics Processing Unit
GPGPU	General Purpose GPU Programming
HPC	High Performance Computing
JVM	Java Virtual Machine
LTC	Lazy Task Creation
MC	Misclassification Cost
ML	Machine Learning
OS	Operating System
RAM	Random Access Memory
STM	Software Transactional Memory
SVM	Support Vector Machine
TBB	Thread Building Blocks
TLS	Thread Level Speculation

Chapter 1

Introduction

1.1 Motivation

In the last two decades, the evolution of processors has changed from single-core to multi-core. Multi-core design has been adopted in response to the lack of increase of clock speed, which has stabilized over the last decade. Manufacturers have limited clock speed because increasing it would exponentially increase power consumption. Today, the multicore design is not limited to high performance servers or workstations. Even recent smartphones feature quad and octo-core processors, showing how definite multicore design is. Computers can have multiple processors, each with several cores, and each core may have two threads of execution, which is the case with Intel hyperthreads. Graphics Processing Units (GPUs), initially designed to improve graphics performance, have been used for other types of applications, leading to the field of General Purpose GPU Programming (GPGPU). GPUs feature a higher number of parallel threads than multicore processors, allowing for massive parallelism. However, the different architecture of GPUs does not improve the performance of all programs.

In order to take advantage of these new architectures, programmers must write their programs to be parallel (Sutter, 2005). From Operating System (OS) APIs to high level parallel programming languages, there have been different models to express parallelism in software programs.

The most low-level parallel programming model is the usage of the native threads of the Operating System (OS), such as PThreads on Linux (Nichols et al., 1996). Using native threads, the programmer writes code that will be executed potentially at the same time, on different OS threads. The OS scheduler uses multiplexing to support the execution of more threads than the number of hardware threads. Multiplexing is also used on top of native threads to reduce the overhead of system calls. For instance, the Java Virtual Machine (JVM) supports green threads (Software, 1997) that appear to the programmer as being a OS-level thread. However, they follow a M to N model, in which M green threads execute on N native threads.

At a higher programming level, there are language extensions and compiler directives used to automate the creation of threads. OpenMP (Dagum and Menon, 1998), Cilk (Blumofe et al., 1996) and Thread Building Blocks (TBB) (Reinders, 2007) are examples of extensions to Fortran, C and C++ that allow the expression of parallel computations without the explicit usage of threads. Parts of the code that are parallelized are considered tasks. Since tasks have a finer granularity than

threads, one OS-level thread can execute several tasks. Tasks are also used as the smallest unit of work used to balance the workload of programs. In the case of irregular or asymmetric programs, one thread may have more work scheduled than other. The runtime systems allow for threads that finish work early to steal work from another thread (work-stealing). This can reduce the execution time of programs, by moving tasks from the critical path to available threads. More recently, new languages have incorporated parallelism as part of the language, such as parallel loops and parallel *places* or datagroups to group parallel tasks that have to be synchronized. X10 (Charles et al., 2005), Chapel (Chamberlain et al., 2007), Fortress (Allen et al., 2005) and Æminium (Stork et al., 2014) are examples of these languages, which also use work-stealing runtimes to automatically adjust the program to the available hardware.

Despite these efforts, writing parallel programs is a difficult task. McKenney (2011) identified four main problems in parallel programming, in no specific order: parallel access control, interacting with hardware, resource partitioning and work partitioning. This thesis focus on the latter. Work partitioning consists on identifying which parts of the code should execute in parallel. We can refer to each of those parts as tasks. A tasks is a representation of a set of instructions that can execute in parallel with others. A parallel program is made of several tasks. While a thread can be considered a task, this concept is more commonly used at a higher level of abstraction. Typically, several tasks are executed on a single thread, with a relatively small penalty for task scheduling. On the other hand, the existence of several tasks per thread allows work-stealing runtimes to automatically balance workload, resulting in a better performance in highly irregular programs.

The work partition problem is two-fold, as its goal is to optimize the performance while maintaining the semantics of the correspondent sequential program. Thus, the problem can be subdivided in two: identifying parallel tasks and selecting the ideal task granularity.

The identification of tasks is traditionally performed by the programmer. Either by creating threads or by annotating the source code with compiler directives, programmers decide which parts of the code can execute in parallel. This process of manual parallelization requires domain knowledge and expertise in parallel programming, and even then it is a lengthy and error-prone process. Automatic parallelization of existing sequential programs is desirable but it is even harder because domain-level information is not available. Instead, automatic parallelizing compilers perform static analysis on the code to identify spots where parallelism could be extracted. Cetus (Dave et al., 2009) and Par4All (Amini et al., 2012) are examples of those compilers, which are focused on parallelizing loops.

Selecting the right granularity of tasks is also a very complex problem. If there are not enough tasks, there is an underuse of hardware resources that could have been used to improve the execution time. If there are too many tasks, the task scheduling overhead cost will increase the execution time. If tasks are not well balanced, some of the threads will be idle while they could have been performing useful work. A program that could execute within seconds with the right granularity may take days otherwise, which makes this issue an important one. Finding the ideal granularity is hard and typically requires trial-and-error since there is no ideal granularity for all programs, as the behavior and structure of parallelism can be unique.

Compilers, either automatic or programmer-assisted, are conservative with re-

gards to task granularity, as just-enough tasks are created to use the hardware available. In the case of irregular programs, the task granularity is dynamic and will only be known during execution. Right before a point where a new task can be spawn, the runtime has to decide whether to spawn that task in parallel, or to execute the task workload sequentially to avoid scheduling overheads. This decision follows a granularity or cut-off algorithm. The simpler approach is to limit the total amount of tasks created to a multiple of the available hardware threads, or to limit the creation of sub-tasks to a certain depth. Some more complex algorithms have been developed, but no single algorithm has been shown to outperform the others in all sets of problems. Thus, it is important to understand when to use one algorithm over the others. This issue has been raised before in Duran et al. (2008b), but a solution was not presented.

The evaluation of granularity control algorithms has focused on the execution time of the programs. However, there has been a recent concern with the energetic impact of programs, specially in the case of parallel programs (Steigerwald et al., 2008). Multithreaded programs have a high impact on energy consumption because the workload controls whether cores are idle or performing, and recent processors can control the voltage and clock-speed dynamically to save energy when cores are not being used. The grain of parallel tasks directly impacts the parallelism in the program and has impact on the energy spending, which is not yet well understood. This energy impact is very important for all classes of devices, from supercomputers that spend months on workloads, to small smartphone and tablet devices that need to have a lasting battery life. Thus, it is important to understand the impact of granularity algorithms on both performance and energy consumption.

The granularity of programs needs to adapt to the underlying hardware. The ideal granularity for GPUs is finer than that for CPUs. GPUs have an higher number of parallel threads, and provide an higher performance-energy ratio. But despite these advantages, GPUs cannot be used in all types of program due to the high latency in memory copies and its internal processor architecture. As such, programs can execute on the CPU with a coarse granularity, or on the GPU with a fine granularity. Such decision is not always easy, as it depends on the input data and parallelism in the program, and the wrong decision may slow down the program several times. Thus, there is a need to automatically perform this decision with high accuracy.

1.2 Contributions

This thesis addresses the issues raised in the previous section regarding work partitioning: the identification of parallelism in sequential programs and the selection of the best task granularity for a given program. The main contribution of this thesis is a model for automatic parallelization of sequential programs supported by dynamic granularity control mechanisms for an efficient execution. This contribution can be detailed in:

- A new automatic parallelization model capable of extracting more parallelism from sequential programs than existing techniques. By using static analysis to infer data and control flow, it is possible to extract parallelism at the AST node level. Task and data-parallelism are both identified, and the resulting code can

target CPUs or GPUs. In order to generate efficient programs, the granularity of tasks is coarsened through the usage of a cost-model hybrid granularity algorithm. The model was applied to the Java language and it was evaluated against manual parallelization and a state of the art parallelizing compiler (Rafael et al., 2014; Fonseca et al., 2016; Fonseca and Cabral, 2016a).

- A machine-learning approach for automatically selecting whether to execute a program with coarse granularity on the CPU or fine granularity on the GPU. This approach defines a new metric (misclassification cost), the feature extraction process, the classifier and the cost-sensitive training required to balance the mismatch of the misclassification cost on both platforms (Fonseca and Cabral, 2013).

- New dynamic granularity control algorithms based on runtime data, such as the number of tasks in the queue and number of stacks in the program, which can be used in manually parallelized programs, or to complement the cost-model hybrid granularity algorithm. These algorithms have been evaluated against existing algorithms over a large benchmark suite, comparing execution time and energy consumption. Results have shown that there is no algorithm better than the others in all benchmark programs. Also, energy-wise, the fastest program is not always the most energy efficient (Fonseca, 2013; Fonseca and Cabral, 2016b).

- A Genetic Algorithm (GA) capable of evolving a custom synthetic dynamic granularity control algorithms, that can be used instead of the previous ones, and work-stealing configurations. The GA is able to improve the performance of individual parallel programs (Fonseca et al., 2017).

- Two new approaches for selecting the best granularity algorithm for a program given its features. The first approach uses a ruleset that was obtained from a per-feature misclassification cost analysis. The second approach is the usage of machine-learning classifiers with the same features. These approaches are able to select a granularity control mechanism that has a low misclassification cost.

Finally, it is worth mentioning that the source code for the benchmarks used in evaluations is available online (Fonseca and Cabral, 2016b), and the source-code for the compilers and runtime systems has also been open-sourced and instructions to obtain it are available at `http://alcidesfonseca.com/research/`.

1.3 Thesis Structure

Before delving into new material, **Chapter 2** addresses the state of the art and lays out the definitions and concepts on top of which new material will be presented.

Chapter 3 addresses the automatic parallelization of sequential programs and the compiler-time granularity control. This includes the rules for parallelization, the platform used and the evaluation performed against manual parallelization and another state-of-the-art compiler.

In **Chapter 4**, the selection of GPU or CPU for executing data-parallel programs is addressed. The proposed Machine-Learning methodology is introduced, describing

feature extraction and classifiers. Furthermore, the methodology is evaluated in terms of accuracy and misclassification cost.

In **Chapter 5**, new granularity control algorithms are introduced to improve on existing ones on specific types of programs. Existing and new mechanisms are evaluated on an heterogenous benchmark to identify when to use each one. This chapter verifies that the No Free Lunch Theorem is applicable to granularity control algorithms.

Chapter 6 concerns the energy efficiency of granularity control mechanisms. It starts by verifying that not all mechanisms have the same energy efficiency. Synthetic benchmarks are used to understand how program characteristics influence the energy and time performance of programs. Finally, a real benchmark is used to test the conclusions drawn.

Chapter 7 proposes an evolutionary approach to address the problem of finding the best cut-off algorithm, among other configurations. A genetic algorithm is applied to a benchmark suite, and tested on a larger one. The same genetic algorithm can be applied to successfully evolve a granularity algorithm for a specific program.

In **Chapter 8**, two new approaches are described to select the best granularity algorithm for a specific program. The first approach is a Ruleset based on per-feature analysis of empirical results. The second approach is the usage of Machine-Learning tecniques.

Chapter 9 closes this thesis with overall conclusions and discusses possible directions for future work.

Chapter 2

State of the Art

In this chapter we will cover the most recent advances in parallel programming, with a focus on automatic parallelization, runtime systems that support the execution of parallel programs, and granularity control algorithms. This chapter will also cover the context of this work, and it will emphasize the problems identified.

2.1 Parallelization

Over the years, parallel programming has moved from manual low-level native thread creation and management, to high-level parallel languages, libraries and compilers. We will cover language extensions and compiler directives that automate the manual parallelization process, automatic parallelization tools that convert sequential source code into parallel programs with no or almost no human intervention, and parallel-by-default languages.

2.1.1 Manual parallelization tools

Writing parallel programs is hard, and over the years several tools have been developed to assist developers in that task. The first step in the parallelization process is to identify which parts of the code can be executed in parallel. Different tools have different approaches to do this.

Cilk (Frigo et al., 1998) is an example of a language extension to C and C++ which allows function calls to be executed asynchronously, until a synchronization point is reached in the parent function. This style of parallelism is called divide-and-conquer or fork-join parallelism, in which a program is subdivided in smaller tasks, perhaps recursively, which are scheduled for parallel execution. Listing 2.1 shows the Fibonacci program written in Cilk. `cilk_spawn` executes the function in parallel, and `cilk_sync` awaits for all spawned functions to finish execution. Divide-and-conquer parallelism has become a popular option for expressing parallelism, and it has been reimplemented in other languages. Intel Thread Building Blocks (TBB) (Reinders, 2007) provides a functionality similar to Cilk, but presented as a C++ template library instead of a compiler extension. Being a template library allows it to avoid the function call overheads of being a library, and does not require special compilers and tools like Cilk. Java ForkJoin (Lea, 2000) is a Java library providing the same functionality. This library was considered so useful that it was included in

the standard libraries of Java and Scala, and is the engine underneath the parallel collections of Java and the actor model scheduler in Scala.

```
int fib(int n) {
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Listing 2.1: Example of the Fibonacci program written in Cilk

Fork-join parallelism allows programmers to avoid thread creation and management. Instead, programmers need only to identify which functions can be executed asynchronously. This model encourages programmers to limit the memory access through function arguments, discouraging access from the spawned function to the whole program memory. Access to shared objects still needs to be synchronized by the programmer. This approach is also independent of the underlying hardware. There is no need for the programmer to know how many threads will be created, as the creation of threads will be done during execution, when the number of hardware threads is available. The details of the runtime execution of Fork-join parallelism will be detailed in Section 2.2.

Fork-join parallelism focus on expressing task-parallelism, allowing the program to perform two different tasks at the same time. Although it is possible to express data-parallelism in Cilk, through the recursive division of the data structure, data-parallelism was the focus of OpenMP creation. OpenMP (Dagum and Menon, 1998) is the most common approach in scientific programming for shared-memory parallel programming, as it makes the process of parallelizing intensive for-loops easier. OpenMP is source-compatible with either Fortran or C, allowing for the same program to be executed in parallel or sequentially, depending on the compiler flags used. OpenMP is based on directives that programmers write inside comments. The directives give the compiler information regarding whether loops can be parallelized, which variables should be synchronized, which variables should be shared or local to the thread, among many other options. Listing 2.2 shows an example of the approximation of the integral of the function $f(x) = 50/(\pi * (2500 * x * x + 1))$ using OpenMP. The first comment defines each variable as being shared or private to each thread. The comment defines the sum over the `total` variable to be a parallel reduction. This example illustrates the type of information the programmer needs to encode, so the parallel code generated by the compiler has the same semantics as the original program.

```
# pragma omp parallel shared ( a, b, n ) private ( i, x )
# pragma omp for reduction ( + : total )
for ( i = 0; i < n; i++ ) {
    x = ( ( double ) ( n - i - 1 ) * a + ( double ) ( i ) * b ) / ( double ) (
        n - 1 );
    total = total + f ( x );
}
```

Listing 2.2: Example of the Integral program written in OpenMP

In version 3.0, OpenMP added support for task-parallelism (Ayguadé et al., 2007) in the same style as Cilk. Listing 2.2 shows the same Fibonacci program written using OpenMP task style. The OpenMP version is more verbose as it requires the access to each variable to be explicit, allowing for more fine-tuned optimization by the programmer. The features of OpenMP have been developed mostly in the OMPSS (Duran et al., 2011) language, which was design to be the playground where OpenMP ideas are tested. Examples of these features are new backends, such as GPUs, and semantically rich annotations.

```
int fib(int n) {
    if ( n == 0 || n == 1 ) return(n);
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

Listing 2.3: Example of the Fibonacci program written in OpenMP

Wool (Faxen, 2010) is a recent template-based C library with the same semantics as Cilk. Just like TBB, the execution overhead is minimal as it is a template library with a very light-weighted runtime system. Haskell also provides parallel programming constructs to create *sparks*, lightweight tasks similar to short-lived green threads (Marlow et al., 2010). The higher-order programming style of Haskell allows data-parallelism to be easily expressed, and the immutability aspect of the language does not require synchronization on variables.

Besides the mentioned tools, there are other language extensions and libraries that add parallelism to these and other programming languages. Task Parallel Library (Leijen et al., 2009), Unified Parallel C (El-Ghazawi and Smith, 2006) and Co-array Fortran (Numrich and Reid, 1998) are just a few examples. However, other tools use the same approaches here described.

2.1.2 Automatic parallelizing compilers

Even with language extensions and libraries, programmers still need to spend time and effort on parallelizing programs. In larger and complex programs, this is especially a problem as there might be complex dependencies in data and control flow in the program. Automatic parallelizing compilers avoid this problem by tak-

ing a sequential program and generating a parallel version that preserves the same semantics.

In the Functional Programming (FP) paradigm, data structures are immutable, hence avoiding conflicts between two concurrent threads. Thus, automatic parallelization of functional languages is straighforward. Typically higher order functions, such as map or reduce, parallelize the operation spawning tasks for subsets of the collection being iterated. This has been done in LISP (Hogen et al., 1992) and Haskell (Marlow et al., 2009), among other languages.

In imperative languages, like the case of C or Java, the process of automatic parallelization is more difficult. Functions can access and modify global variables. When parallelized, the modification of global or shared variables must follow the same order to maintain the semantics. In complex programs, there can be so many interactions between possible tasks and shared variables that identifying parallelism that preserves the correctness of the program is a problem for both humans and compilers.

The main focus of research in automatic parallelization has been the parallelization of for-loops. Loops can fall in three different categories: *DO-ALL*, *DO-ACROSS* or *DO-PIPE*. *DO-ALL* parallelism does not contain any interference between loop iterations, meaning that they only read shared variables and write only to local variables. In the case of arrays, there are no writes to common indices of arrays. Listing 2.4 shows an example of a loop that follows this pattern and all iterations can execute independently. The parallelization of *DO-ALL* loops consists in grouping sets of iterations into a task and schedule tasks to different native threads. The program will only advance past the loop when all iterations of the loop are completed, to ensure a consistent state.

```
for (int i=0; i< n; i++) {
    a[i] = b[i] + c[i];
}
```

Listing 2.4: DO-ALL example

DO-ACROSS loops can have a region of the iteration that may interfere with other loops when writing to shared variables. The parallelization of these loops is useful when the interfering region is small compared to the remainder of the loop. Listing 2.5 shows an example of a DO-ACROSS loop. For parallelizing DO-ACROSS loops, there are two approaches. The first is to use atomic operations, either by using processor atomic instructions (like the Compare and Swap atomic instruction), or by using mutexes to ensure only one iteration performs that operation at a given time. The second option is to use a local variable shadowing the shared variable in which temporary results are stored. In the end of all iterations, the local shadow copies are all merged into the original variable.

```
for (int i=0; i< n; i++) {
    total += b[i] + c[i];
}
```

Listing 2.5: DO-ACROSS example

Finally, *DO-PIPE* loops have dependencies between different steps inside the same iteration. Listing 2.6 shows an example of a DO-PIPE loop. The parallelization can be done by having different threads executing different steps, and iterations are moved from one thread to the other, according to which step they are in.

```
for (int i=0; i< n; i++) {
    c[i] = i + 1;
    b[i] = c[i] + 2;
    a[i] = b[i] + 3;
}
```

Listing 2.6: DO-PIPE example

In order to verify if the loops can be parallelized or not, the Polyhedral Model is frequently used (Bondhugula et al., 2008a). This model supports nested loops by considering each iteration within the most deep loop as lattice points inside a polytope object. A polytope object can have `n` dimensions and any number of flat sides. Each side corresponds to the multiple values of the variables used to identify each iteration. This model supports multiple transformations on the polyhedral model to optimize the code to the target platforms. Polyhedral skewing is an example of a rewrite rule that optimizes the data accesses to the result of previous iterations.

However, parallelization of nested loops is not always efficient. If the cost of executing one iteration in the deepest loop is too low, compared with the task creation overhead, parallelizing it will decrease performance. Most commonly, only the first or two outermost loops are parallelized, in order to create more coarse grained tasks.

Most automatic parallelizing compilers are based on this model for loop parallelization, such as Polaris (Pottenger and Eigenmann, 1995), Cetus (Dave et al., 2009), Pluto (Bondhugula et al., 2008b), Polly (Grosser et al., 2011) or Par4All (Amini et al., 2012).

Zhao et al. (2005) presented an approach in which the parallelization occurs during execution, on Java bytecode. This approach was limited to DO-ALL loops, which are the most simple to identify and the ones that yield the highest speedup. When considering the parallelization and thread creation cost, this approach is unable to provide speedup on the original programs.

Haghighat and Polychronopoulos (1993) presented an alternative to the polyhedral model through the usage of static analysis. This work focused on understanding data dependencies, much like the polyhedral model, and modeling the domain of induction variables, the variables used to identify the iteration. Through the usage of abstract interpretation, it is possible to model the execution time of loops and perform work-partitioning accordingly, in order to balance the work across threads. This approach also supports replacing a set of arithmetic operations by faster equivalent alternatives. This work proposes the usage of abstract interpretation, symbolic algebra, calculus of finite differences, number theory, convex analysis and theorem proving methods for automatic parallelization and optimization of programs. However, this work only presents small incomplete proof of concepts that cannot be systematically applied to any sequential program.

While loop parallelization has been the focus of automatic parallelization research, there is also potential parallelism at the instruction level. This type of

parallelism is present in divide-and-conquer algorithms, which are a very popular pattern and the focus of Cilk, ForkJoin and OpenMP task model. zJava (Chan and Abdelrahman, 2004) used symbolic access paths, which denote how objects are written and read at the local level. This information is then summarized at the method level, according to whether it has global effects or not. Parallelization occurs during runtime, when this information is used to translate access paths into regions. Each region represents an area of the memory of the program that can be accessed concurrently by multiple threads. Each region has a writer/reader lock, which is used to guarantee the order of the operations by concurrent threads. Threads can be created for each method invoked, but there is a need to manually select the best granularity in order to prevent slowdowns. The evaluation of this model is limited to DO-ALL loops that do not occur in an heavy synchronization overhead.

Girkar and Polychronopoulos (1992) suggested the usage of a hierarchical task graph to represent parts of the programs and their data dependencies. The graph can then be used to identify tasks that can execute in parallel. This has been the basis for most task-level parallelism approaches. Symbolic analysis is used to verify if two different recursive calls access the same elements of an array. If not, the program is parallelized speculatively. Threads are created with a shadow copy of the global memory, and only if there are no incompatible data dependencies is the local memory copied back to the global memory. If there is an incompatibility, local data is discarded. This approach was able to achieve up to 3 times of speedup on a four core machine.

Gupta et al. (2000) proposed an approach for automatic parallelization of recursive methods. This approach represents accesses to variables as *may have writes*, *definitely has writes* or *may expose the variable for future writes*. These annotations are propagated through the program up to the inter-procedural level. Symbolic analysis is used using this inter-procedural array dependencies for identifying tasks that access independent sub-ranges of arrays to parallelize. This analysis focus on data-parallelism, even in recursive calls. The authors later applied this approach to Java Artigas et al. (2000) using a runtime-supported alias analysis. This approach also used native threads instead of Java threads for performance benefits.

While not considering the identification of parallelism, Bik and Gannon (1997) suggest code transformations to apply to loops and recursive methods. The sequential code is translated into worker threads that execute strides of loops, or subsets of the recursive invocation. Speedups are obtained on a four core machine. This approach has the limitation of only supporting balanced workloads.

Abdelrahman and Huynh (1996) suggest a compiler for automatically parallelizing tasks identified in sequential source code. The compiler identifies the regions in memory which each function uses. Before executing a certain task, the scheduler has to obtain exclusive access to those regions. Region lists are auxiliary structures to guarantee order in the region accesses, in order to guarantee the semantics of the sequential program. Speedups of 23 on a 28-core processor were obtained.

OoOJava (Jenista et al., 2011) also uses static analysis to parallelize Java. The model is inspired by out-of-order processors, in which one processor selects one instruction in a stream and, as soon as all dependencies are met, it is scheduled to a functional processor. In OoOJava, when a thread reaches the beginning of a new task, it marks that task as scheduled and resumes to where the task would end. The OoOJava runtime will await for the task dependencies to be met, and

schedule that task to another thread. Programs are represented as variable reads, variable writes and variable copy statements. Additionally, entering and exiting a task are also represented. Technically, OoOJava does not perform strictly automatic parallelization, as the programmer annotates the task in the source code. However, the data dependencies of the source code are inferred automatically, leaving only the granularity of tasks to the programmer. Since OoOJava is not able to know all the data dependencies during compilation, the tracking of that information is done at runtime, especially in the case of variables stored in the head. OoOJava then translates the graph of conflicts between tasks into queues for code generation in the C language. One of the limitations of OoOJava is that it cannot handle IO or exceptions. OoOJava was able to achieve speedups in all benchmarks up to 20 times on a 24 core machine.

MP-Tomasulo (Wang et al., 2013a) also uses Out-of-Order instructions for automatically parallelizing code for FPGAs. Given the nature of FPGAs, it is possible to eliminate write-after-read and read-after-write dependencies by renaming parameters, while the control flow of each thread is independent. This approach cannot be translated to multicore processors.

Jrpm (Chen and Olukotun, 2003) is another approach for parallelizing Java programs, but operates on Java bytecode instead of the source code. This model targets chip multiprocessors with thread-level speculation (TLS) abilities. TLS allows the program to schedule threads optimistically without violating the correct behavior of the program. This work focuses on loops by identifying the regions of the program where TLS can be exploited. One limitation of this is that only one loop nesting can be parallelized at a given time. The identification of these regions is done by tracing the execution of the sequential program and collecting timing data on dependencies. This approach was able to obtain speedups between 2 and 4 on a 4 core processor.

Software Transactional Memory (STM) has been used to execute TLS programs on regular hardware. This approach detects conflicts when they occur, in contrast to preventing them during compilation. Using STM will lead to more potential parallelism because more independent tasks will be considered, allowing for a chance of conflict among them. However, whenever a conflict occurs, there is a heavy performance penalty for executing the rollback to the previous state. STM also introduces memory costs for storing the last safe memory state. Mehrara et al. (2009) presented an STM approach that was unable to achieve 4x of speedup on a 8-core machine, showing how expensive the mechanism is. Additionally, STMs are shown to only be advantageous in parallel blocks with a low chance of conflicts. HydraVM (Saad et al., 2012) is a STM automatic parallelizing compiler for legacy code. Parallelism extraction is done at the loop level by considering *superblocks* of code, that do not have any IO and represent independent paths of execution. The notion of independent paths allows for memory conflicts inside branched instructions, ideally with a low probability of executing. Hydra has shown a speedup up to 5x on a 8-core machine.

2.1.3 Parallel-by-default programming languages

Both OpenMP, Cilk and automatic parallelizing compilers assume there is an original source code written in a sequential style. Over the last decade, new languages have been developed that do not follow a sequential execution model. These languages

have been designed from scratch with parallelism in mind, making it easier for programmers to express parallelism. X10 (Charles et al., 2005), Fortress (Allen et al., 2005) and Chapel (Chamberlain et al., 2007) are examples of such languages, targeting High Performance Computing.

Fortress is a programming language designed by Sun for scientific use, a more up-to-date version of Fortran. Fortress is designed to support multiple programming paradigms. One of these paradigms is the usage of the functional style in operations over arrays. These operations cannot have side-effects, which makes it automatically a candidate for parallel code generation. The whole language executes in parallel by default. Tuples, functions, arguments, generators and explicit parallel blocks all can execute in parallel without any programmer annotation. Loops in Fortress are not guaranteed to be sequential. During execution, the runtime will execute the loop in parallel or sequentially. Determining the criteria for this decision is not trivial.

The X10 language is based on the concept of *places*, memory regions that cannot be accessed concurrently. In X10, functions and methods are annotated with the *places* used. This information can be used by the compiler to detect disjoint parts of the program that do not access any common place. These disjoint parts can be executed in different native threads without any need for synchronization. It is possible to increase parallelism, by executing functions that access the same place in parallel, as long as the access to that place is synchronized through the usage of a mutex. X10 supports the `async` keywork, which precedes a method, indicating that the invocation can occur in parallel with the remainder of the current block, similarly to `cilk_spawn`. An asynchronous invocation does not necessarily spawn a new task. It can be inlined, executing as if it was sequential. This decision is left to the runtime system. Atomic blocks are another construct used to prevent the effects of global writes to be ignored by concurrent writes through the usage of locks.

In Chapel, a *locale* represents a unit of computation that has uniform access to memory, and the program has access to a list of *locales* available to perform computations. Chapel uses the concept of domains, a set of indices which support parallel iteration. Domains as a language construct are an evolution over the concept of *places*. Domains have standard library support for iteration, reduction, scanning and other operations that can occur in parallel, mapped to *locales*. Task parallelism can be obtained in Chapel through the usage of the `cobegin` construct, which allows all the operations inside that block to start asynchronously in parallel. Producer/consumer synchronization variables are available to force a specific order, and atomic blocks are also part of the language.

Æminium (Stork et al., 2014) is another language that is designed to be executed in parallel, but it focuses on concurrency support. Æminium uses a declarative approach to concurrency, in which programs are expressed without any synchronization primitive. However, variables and method signatures are required to be annotated with access permissions, representation of how that variable can be accesses by concurrent threads. For instance, a variable may be *unique* and only one thread may have access to it, or it might be *shared*, in which several threads can concurrently access it. The programmer does not create threads or annotates tasks in the source code. Any method invocation can occur in parallel, depending on the access permissions declared. Æminium uses a dataflow approach, in which the program is converted into a Directed Acyclic Graph (DAG) of tasks. This graph is compiled into Java source code for execution on a runtime system.

All of these languages have powerful compilers that use static analysis to identify parallelism, but they also require a runtime system for efficiently executing the generated code. Several parallelization decisions are delayed until execution because only then will information such as data size and load on the system be available.

2.2 Parallel program execution and optimization

2.2.1 Runtime task schedulers

Parallel libraries or languages rely on runtime systems to manage the execution of tasks. The two main approaches for task scheduling are work-sharing or work-stealing. In work-sharing, tasks are divided among different threads for parallel execution. This distribution can be done at compile-time or during execution onto a threadpool. This approach assumes all tasks are equally balanced, meaning that they will take roughly the same amount of time to execute. When the duration of times is asymmetrical, then the performance of work-sharing runtimes will be equal to the duration of the largest group of tasks.

To overcome this deficiency in work-sharing, one can use work-stealing runtimes. The most popular work-stealing runtime is the THE algorithm (Frigo et al., 1998), implemented in Cilk, Fork-Join, X10 and Æminium. In work-stealing, a fixed number of threads are idle waiting for work. Each thread has its own queue to which tasks created in that thread are scheduled. Thus, threads execute tasks that spawn new tasks that are added to the queue of that thread. When a task tries to pop tasks but the queue is empty, it attempts to steal a task from another queue. The choice of whichspecific queue to steal from can follow different policies.

Which task to steal may be chosen according to its position on the queue, the influence of the task on the continuation of the program, the parent-child relation of tasks, or even at random. The best policy is also dependent of the nature and structure of the programs. In machines with NUMA memories, locality-aware stealing can improve the performance (Acar et al., 2000; Guo et al., 2010). Furthermore, having two queues per thread, one for sharing and another for consumption, can make the work-stealing more scalable (Dinan et al., 2009). The traditional Cilk policy is *work-first*, that tries to perform work before spawning new tasks. Guo et al. (2009) introduced *help-first*, in which creation of more work is preferred, creating more parallelism in the beginning of the program for other threads to steal early.

Wool introduced *leapfrogging* (Wagner and Calder, 1993) in task-based work-stealing runtimes, allowing one worker to perform work when waiting for synchronization on another task that is being executed. This improves the performance on scenarios where stealing would not be enough to load-balance the system. LACE (van Dijk and van de Pol, 2014) improves Wool by using two deques per worker, one private and another public, reducing the frequent overhead of synchronizing when removing tasks from the owned queue, an issue identified by Acar et al. (2013).

2.2.2 Granularity Control

Work-stealing runtimes improve the performance of a program by load-balancing tasks to reduce the overall execution time of the program. However, in order to achieve balance, tasks need to be small enough for stealing to be possible. But if

there are too many small tasks, the overhead of scheduling tasks becomes larger and it might slow down the program. On the other hand, coarse-grained tasks might not use all the parallelism in the machine. Thus, there is a trade-off in this decision, and that trade-off depends on the specific nature of the parallel program (Duran et al., 2008b).

Controlling the granularity of tasks can be done at runtime, using Lazy Task Creation (Mohr et al., 1991) (LTC). In LTC, the creation of new tasks is not always true, it is conditional. If a certain condition is true, the task is created and scheduled to run. If the condition is false, the work is immediately executed inside the current task. The advantage of this technique is that it prevents the runtime from creating unnecessary tasks, thus avoiding the overhead from object creation, memory allocation and queue management. Listing 2.7 shows an example of LTC being used in the implementation of the naïve Fibonacci algorithm. The decision criteria ($\mathsf{n} < \mathsf{16}$) is a custom solution for this algorithm and cannot be used for other programs. This thesis will focus on general purpose cut-off techniques that can be used on different programs.

```
int fib(int n) {
    if ( n == 0 || n == 1 ) return n;
    if (n < 16) {
        return fib(n-1) + fib(n-2);
    } else {
        Future f1 = new Future(() => fib(n-1));
        Future f2 = new Future(() => fib(n-2));
        return f1.get() + f2.get();
    }
}
```

Listing 2.7: Example of LTC on the Fibonacci Example

The decision whether to create tasks or to inline a sequential version is the major decision in this approach. Most of times, the cut-off decisions custom-made by developers are the ones that provide the best results. Developers are able to use any information at their disposal, including domain knowledge, to create the right cut-off strategy for each program. Unfortunately, automated systems do not have access to the same information, or are not able to understand it the way developers do, making it very difficult for these systems to obtain the same gains as their human-made counterparts. Nonetheless, several algorithms have been proposed to identify the best cut-off solutions using only knowledge about the program's executing code and run-time data.

OpenMP initially implemented two cut-off approaches (Duran et al., 2008b) for their task model: MaxTasks and MaxLevel.

In **MaxTasks**, tasks are created until the total number of active tasks in all worker queues reaches a certain threshold t. After that point, all new computations are inlined instead of spawning another thread. When the number of active tasks lowers, new tasks can be created until the threshold is reached again. The threshold in this approach is typically defined as the number of processor threads on the machine, adapting to different machines, but being oblivious to other factors such as memory and processor speed. In order to decrease the overhead of computing the size of queues, the size of other queues is estimated from the size of the current

queue after applying a factor of (number of idle threads / active threads), because idle threads are known to have 0 tasks in their queue. This estimation assumes a regular distribution among threads, which may not always happen.

MaxLevel makes use of the tree-shaped structure created by divide-and-conquer algorithms. In order to avoid the creation of too many tasks, the cut-off limit may be defined by the depth of the recursion l, which can be calculated by the number of ancestors of the running task.

Later, Duran et al. (2008a) introduced the **LoadBased** algorithm, in which tasks are inlined when all threads are executing work, and not idle. In the same work, **Adaptive Tasks Cut-Off**, or *ATC(t,l)*, was introduced. Tasks are only created if two conditions are met. The first is that there are fewer tasks than the number of threads on a given recursion level. This condition forces the threads to expand in depth, creating work for all threads and being within a certain bound limit. The second condition is that the depth-level is less than a certain threshold. Thus, ATC is the combination of *max-level* and *max-tasks*.

ATC adds a profiler that saves information regarding how much time a subtree takes to execute, and predicts further subtrees (if the prediction is larger than 1ms, the task will be created). This is, however, based on the assumption that all tasks inside a level have a similar behavior, which does not happen in unbalanced parallelism.

ForkJoin also uses another metric, **Surplus Queued Task Count**, or *Surplus(t)*. This approach relies on the size of work-stealing queues. Before creating a new task, the number of queued tasks in the current thread that exceeds the average number of tasks in other queues is compared to a threshold limit t (usually 3 in existing ForkJoin benchmarks). In other words, it inlines a task if the current queue size is t tasks higher than the average size of other queues. The goal of this cut-off approach is to allow a single task to create several tasks for others to steal early in the program, but prevent the queue from being too large when there is already work to be stolen.

Acar et al. (2011) introduced the **Oracle** granularity control mechanism. This approach has only been applied to implicit parallel languages, such as the case of ML. The programmer manually annotated each function with the asymptotic complexity. Each recursive call then uses the annotation and the task depth to estimate the cost of future calls. Then, that information is used to decide whether it is beneficial or not to create a new parallel task. In order to adjust to the program execution, this information is retrieved using a moving average of several runs.

Cong et al. (2008) introduced **Batching** in the X10 runtime. Batching pushes lists of tasks instead of single tasks into the work-stealing queues. The main advantage is that the lists of tasks promote the use of caching, because of memory locality. It additionally has implementation benefits using list swapping instead of pushing and popping, during the steals.

However Batching also requires a threshold for the size of task lists. It has been defined as the $\mathsf{min}(2^{\mathsf{Q}}, \mathsf{S})$ where Q is the queue size and S is a user-defined threshold dependent on the algorithm in cause.

Chen et al. (2007) focused on programs with cache-locality, and used profiling information to improve the performance of work-stealing. In this approach, the program is previously executed several times to generate memory traces. Then, a one-pass approach is used to group tasks together in a way that benefits the L2

cache usage.

Outside the scope of shared-memory parallel programs, there have been several approaches (Gerasoulis and Yang, 1993; Sobral and Proença, 1999; Nascimento et al., 2007) to granularity control in grid computing. The main difference is that the overhead in communication in distributed memory systems is large enough that it has to be taken into account. Most common approaches optimize the granularity by minimizing the data transfers between machines, clustering tasks that access the same data together.

2.2.3 Evaluations of Cut-off Algorithms

The evaluation of cut-off algorithms has been a recurrent concern over the years. For instance, Duran et al. (2008b) have found that, in OpenMP, *Max-Tasks(t)* and *Max-Level(l)* performed differently on different programs, with no algorithm being considered better than the other. The study also concluded that choosing the wrong cut-off could have a negative impact in performance, making the program run longer than when not using any cut-off technique. The difference in performance between tied and untied tasks was also studied. A tied task can only execute on one thread, while untied tasks can be split and distributed across different threads. Cut-off techniques are more effective in the presence of untied tasks. The authors also suggest that depth-first schedulers should be the default, since they are specially well suited to handle cut-off techniques to avoid a high level of recursion whenever there is a significant allocation of memory. Finally, the authors were not able to conclude which cut-off algorithm is the best for each class of application.

Olivier and Prins (2009) studied unbalanced workloads, concluding that different parameters of *Max-Tasks(t)* and *Max-Level(l)* yield different results. Unfortunately, they also observed that there was no unique parameter capable of achieving the best results for all scheduling policies. Also, the empirical evaluation data was collected from experiments executed on machines with just 2 cores, which is not representative of the machines available today.

Duran et al. (2008a) also studied the behavior of ATC, confirming that the best cut-off algorithm was program dependent. The Adaptive approach was introduced in an attempt to limit the problems with both Max-Tasks and Max-Level algorithms. Max-Level did not create tasks in depth, not making use of the potential parallelism in the program. Max-Tasks, on the other hand, would create too many tasks, introducing an undesirable overhead. The main objective of combining the two was not to improve performance, but rather to minimize the penalty of both approaches.

Podobas et al. (2010) compare task-based runtimes including OpenMP and Wool. The study focus the importance of the task-depth cut-off, referring that OpenMP approaches perform poorly when tasks are very fine-grained. The Wool runtime performed better by reducing the overhead in task-scheduling.

In Taura et al. (2012), several possible improvements on task-based work-stealing runtimes are presented. Cut-off approaches are identified as the optimization that can yield the higher increase in performance. Their approach is to use the depth of the task when selecting candidates for work stealing. Their approach was able to achieve 18.2% of speedup, compared with the 40% possible by selecting the best cut-off manually.

In ForkJoin, the best cut-off also changes depending on the program and the ma-

chine (Cong et al., 2008). Thus, the choice of the cut-off algorithm for the framework was left open for future work by the authors. Dig (2011) propose a profiling-based automatic refactoring engine. The engine generates several refactoring changes and evaluates the resulting performance. One of the possible refactoring change is the introduction of a cut-off limit. Achieving a good performant parallel program can be done automatically, but it consumes an unrealistic amount of time until the perfect value is found.

2.2.4 GPU execution of parallel programs

Over the last decade, GPUs have been used to improve the performance and energy efficiency of parallel programs. Despite the advantages of massive parallelism and lower energy consumption, GPUs have a different architecture that does not allow all programs to be compiled to or efficiently execute on them.

First of all, the granularity of the parallelism for GPUs is different than for multicore CPUs. GPUs can execute a large number of threads simultaneously, thus requiring fined grained programs. Multicore processors support a lower number of threads, leading to a coarser granularity in tasks. However, while the CPU has a direct access to the host memory, the GPU has to copy data from the host memory to its dedicated memory. This implies that despite the high throughput of GPU operations, each operation has a large latency. Generally, only data-parallel operations with a fine granularity, over a large amount of data, can be accelerated on the GPU.

It is important to understand how GPUs can be used in parallel programs, as well as when to use GPUs or not for a given program.

The two most common languages for GPGPU are NVidia CUDA (Nvidia, 2007) and OpenCL (Munshi, 2009). The CUDA toolkit includes a compiler and a library. The compiler supports a superset of C and generates CPU programs with calls to the CUDA library, as well as GPU-native code for GPU functions, called kernels. The CUDA library supports the dynamic execution and synchronization of kernels, as well as data transfers between the host and GPU memories. OpenCL is an alternative to CUDA, supported by the majority of GPU vendors. Both languages were designed to enable general-purpose computations on the GPU and are based on the C programming language.

These toolkits have been wrapped in bindings for higher-level programming languages, such as JCuda (Yan et al., 2009) or JavaCL (Chafik, 2011a). On top of these bindings, higher-level domain-specific languages (DSLs) have been developed for GPU programming, such as ÆminiumGPU (Fonseca, 2011) and ScalaCL (Chafik, 2011b). Both approaches feature a compiler that translates higher order functions to OpenCL. The generated OpenCL code is invoked by a runtime library that handles memory copies between platforms. Aparapi (Frost, 2011) is another alternative for Java GPU Programming that, instead of using a compiler, dynamically generates OpenCL code from bytecode just before method invocation.

Copperhead (Catanzaro et al., 2011) is another framework for GPGPU, targeting the Python language. Since Python is dynamically typed, kernels are only generated at the call-site when the types of input data are known. Cunningham et al. (2011) introduced a CUDA back-end for the X10 programming language, using *places* to represent CPU or GPU computations. However, GPU programming

in X10 is explicit and requires special annotations.

Mars (He et al., 2008) and MapCG (Hong et al., 2010) are two C libraries for GPU execution of MapReduce algorithms. These libraries use a key-value approach, instead of directly using the map and reduce operations over lists, as it is preferred by other approaches. Despite being focused on the GPU architecture, both approaches support CPU execution of algorithms.

Accelerate (Chakravarty et al., 2011) is a Haskell library for GPGPU programming that uses the laziness of Haskell to build an operation representation that is converted into CUDA or OpenCL for GPU execution. Alternatively, operations can also execute on the GPU. This approach wraps GPU-ready operations with the `Acc` monad, which limits its integration with existing code.

The approaches presented so far for GPGPU programming require the programmer to decide between the CPU or the GPU. Qilin (Luk et al., 2009) is a GPGPU framework for C++ that features adaptive mapping. Adaptive mapping consists in recording the execution time of a program with different data inputs to create a cost-model. Future executions of the program will be able to use the cost-model to decide between GPU or CPU for execution. This approach is useful for programs that are executed several times with different inputs.

Joselli et al. (2008) presents a CPU-GPU decision mechanism for real-time applications that schedule operations inside a long loop. A similar approach to Qilin is taken, in which different loads are scheduled to the CPU and GPU to test their execution times, and the best one is used afterwards.

2.3 Energy consumption of Parallel Programs

Nowadays evaluating the performance of different program granularities is not enough. It is also important to understand the impact of different parallelization approaches on energy consumption

There have been two approaches for energy efficient scheduling on multicore processors: inter- and intra-program management. An inter-program scheduling will be handled by the OS scheduler, while intra-program scheduling is handled by each application. Given the focus of this thesis in granularity management, the focus will be on intra-application scheduling.

There have been three main approaches for managing the energy efficiency in parallel programs. The first approach is through the usage of asymmetrical multicore processors, of which *big.Little* is an example. In this architecture, a task can be scheduled for a faster processor or to a slower processor, depending on its urgency or role in the critical path of the program. The second approach is to reduce the parallelism of the program, so only a subset of the available processors are used, saving energy but increasing the execution time. Finally, the last approach is to use Dynamic Voltage and Frequency Scaling (DVFS), a mechanism some processors have to reduce and increase the voltage and frequency of processors. These approaches have been applied to parallel programs, specifically those with real-time requirements or deadlines for completion, in which some slow-downs may be acceptable.

DVFS has been used for static scheduling of parallel programs for HPC. Parallel programs can be represented as DAGs, with dependencies between tasks. Whenever a program is unbalanced, a task may depend on two or more tasks that have different execution times. In that case, the lightest task could execute at a slower speed

without slowing the whole program. Kimura et al. (2006) applied this idea to clusters powered by AMD Turion and Transmeta Crusoe processors. The approach focused on identifying tasks that were not in the critical path, and reducing the frequency of those processors. Results showed it is possible to obtain a 25% energy gain with a 1% loss in performance. However, this was only possible on a distributed cluster architecture, not on a single-machine program.

On a single machine, it is not possible to reduce the clock speed of just one single thread. However, in Multiple-Clock-Domain processors, DVFS can be done earlier and independently for each processor. Cai et al. (2008) and Rangasamy et al. (2008) both presented a compiler-based approach to identify which threads are critical, and which ones can be slowed down without any large performance impact. Non-critical threads would be scheduled for processors that would be slowed down, while critical tasks would execute on the faster processor.

Huang et al. (2009) proposes an earliest-deadline-first static scheduling approach, that makes usage of DVFS to improve energy efficiency in embedded real-time applications. This scheduling approach reduced energy consumption between 3 and 9%.

Furthermore, dynamic techniques have also been applied in Multiple-Clock-Domain processors. Wu et al. (2005) proposed a dynamic frequency adjustment based on the length of its task queue, similarly to what is done in granularity control for work-stealing runtimes. This approach has resulted in a 16% energy savings improvement over a 5% performance degradation.

HERMES (Ribic and Liu, 2014) extends the Cilk work-stealing runtime with DVFS capabilities. When a worker fails to steal a task, its frequency is reduced. On the other hand, when it has work from the current queue, it increases the frequency. This approach leads to a dynamic adjustment of frequencies over workers. This approach requires a static assignment of workers to CPU cores.

On a single-machine with Single-Clock-Domain processors, a different approach is required. ParallelismDial (Sridharan et al., 2013) builds on work-stealing techniques to handle dynamic programs, identifying when queues are empty. When that occurs, that worker is killed, reducing the number of cores being used, thus saving energy. Unlike DVFS techniques, there is no possibility of reducing the speed of each processor individually, but there is the possibility of not using it. By continuously monitoring the system, it is possible to dynamically adjust the number of workers to the workload. It is important to notice that this approach is only valid when a program has the right granularity or coarser. If the granularity is too fine, workers will always have work, just not productive work.

Seo et al. (2008) proposed a dynamic partitioning model for scheduling real-time tasks on Single-Clock-Domain processors. This model is based on two concepts: migration of tasks from busier to under-occupied cores, and dynamic core scaling. The migration will balance workload among processor cores, with the goal of adjusting workload to cores that execute at the same frequency. Dynamic core scaling will disable cores in order to save energy when they are not being used. In this case, work may be migrated from one core to the other. Dynamic task partition resulted in 25% energy consumption reduction, and dynamic core scaling resulted in a reduction of 40%. Despite these improvements, this model requires all the information to be available before a task is executed, and does not concern with task dependencies or other program-specific characteristics.

2.4 Summary

In this chapter we addressed parallel programming with a focus on granularity management, both from the compilation and runtime point of views. Automatic Parallelization tools are mostly limited to exploring parallel loops, ignoring other possible parallelism in function calls and recursive calls, a common pattern for parallelism. Approaches that target this type of parallelism are limited in the optimization of tasks in regards to granularity.

Regarding granularity control in work-stealing runtimes, LTC is the most efficient way of dynamically controlling the size of tasks, but it depends on a certain cut-off criteria. There are several choices that can be used, but none is better for all types of problems. Additionally, there is no heuristic to choose the right cut-off for a given program.

These issues will be addressed in the remaining chapters.

Chapter 3

Automatic Parallelization with Static and Dynamic Granularity Control

Automatically parallelizing sequential code, to promote an efficient use of the available parallelism, has been a research goal for some time now, given its importance in developer productivity and improvement of the performance of legacy programs. This chapter presents a new automatic parallelization approach based on static analysis and static granularity management. Finally, we evaluate that approach on a small set of heterogeneous benchmark programs.

3.1 Introduction

Nowadays, in order to achieve the best performance on multicore machines, programmers have to write parallel programs. This is typically done using threads, either directly or indirectly through high-level constructs of the language. Traditionally, parallelization is done manually by defining threads and synchronizing them. However, this process is often cumbersome and error-prone, often leading to the occurrence of problems such as deadlocks and race conditions. Furthermore, as the code base increases it becomes increasingly harder to detect interferences between executing threads. Writing, debugging and tuning multi-threaded code is very time-consuming, as there are multiple combinations of executions that make the performance and visibility of errors non-deterministic. Furthermore, there are billions of lines of source code inside existent software that are not able to benefit from today's multicore architectures. Parallelizing these programs is a daunting and extremely costly task, one that hardly anyone is eager to initiate.

The automatic parallelization of existing software has been a long running objective and prominent research subject (Banerjee et al., 1993). Existing research has focused mainly on the analysis and transformation of loops, since these have always been perceived as the main source of potential parallelism in sequential programs (Feautrier, 1996). Nonetheless, other models have also been studied, such as the parallelization of recursive methods (Bik and Gannon, 1997) and of sub-expressions in functional languages. Focusing only on the parallelization of loops is not enough

in most cases and other approaches have not revealed significant performance improvements.

In this chapter we introduce a new approach for performing a fine-grained automatic parallelization of programs. This approach is distinct from others, since it parallelizes all the instructions that can, effectively, be executed in parallel. To identify which instructions can be parallelized, we infer instruction signatures from the source code of the program. These signatures include dependency and control flow information, which allows us to organize instructions into a task-oriented structure. The result is a program that exhibits the maximum possible parallelism at the finest granularity level (e.g. one task can equal one instruction). However, in order to achieve good performance and decrease the overhead in run-time task management, the granularity of tasks is coarsened during compilation and also during run-time. At run-time, the system load influences granularity control. Furthermore, a work-stealing scheduler is used to efficiently manage execution and control dependencies.

The proposed approach can parallelize irregular recursive programs with a low runtime overhead, resulting on up to 20x of speedup, on a 24 thread machine and an average of 5x of speedup. Because of dependency tracking and transformations during compilation, we are able to avoid harsh runtime overheads from which existing solutions suffered. This chapter contributes with an hybrid methodology for analyzing procedural source code and translating it to a parallel version with a broad level of parallelism and granularity, that is fine-tuned during execution by runtime granularity control mechanisms. The parallelization approach was tested with popular benchmark tests for task-based parallelism, and compared with the state of the art in Automatic Parallelization and a manual approach.

This approach was applied to the Java language, one of the most popular programming languages, since it has a large code base of legacy sequential software. Java compiles to a bytecode format that is interpreted by a virtual machine, which may compile to native code parts of code frequently used. Despite Java being an interpreted language with garbage collection, our approach can be applied to any procedural or object-oriented language. Typically, object-oriented languages are not the focus of automatic parallelization, as alias between variables to the same object may cause conflict. This approach takes a conservative approach to aliasing, avoiding conflicting code that would not be safe.

In this chapter we will discuss in detail the methodology of the solution in Section 3.2. The implementation details and runtime support will be explained in Section 3.3. Section 3.4 presents a cost-model granularity control mechanism. In Section 3.5, the platform will be evaluated in different programs. Finally, Section 3.6 lays the conclusions.

3.2 Methodology

The proposed automatic parallelization model is based on performing static analysis of the code to identify which AST nodes can be executed in parallel without introducing race conditions. This model uses tasks to represent sections of the code that can be executed in parallel with others. As such, the parallelization process consists in converting sequential code into tasks. The efficient execution of tasks is left to a work-stealing runtime.

The automatic parallelization is performed in two steps: signature extraction and parallelization.

3.2.1 Signature Extraction

In order to automatically parallelize the program, it is necessary to analyze the memory accesses to understand dependencies between parts of the program. If two program parts read and write to the same variable, then they cannot be parallelized without guaranteeing determinism. Thus, the first step of the compiler is to understand what each AST node reads and modifies. Datagroups (Leino et al., 2002) are used to represent different memory sections and if two method calls share no datagroup, it means that they can be executed in parallel. After this phase, each AST node will have a signature, composed of one or more datagroups permissions. An example of signatures in code can be seen in Listing 3.1. Datagroup permissions can be one of the following:

- **read(dg)** - the AST subtree reads the variables represented by datagroup dg;
- **write(dg)**- the subtree writes to the objects in datagroup dg;
- **control(dg)** - the control flow of other operations in datagroup dg may be altered. This is the case with return statements, breaks, continues, ifs and whiles.

In order to account for aliasing (the usage of two variables to refer to the same object), static analysis follows the flow of the program and keeps track of the element and its permission. In case of branching, it considers the conservative union of both branches permissions.

Method invocations are treated differently because, in order to account for side-effects, it is important to match the access permissions of the method declaration with the arguments. Invocations have the access permission of the method declaration where the arguments permissions are transferred to the arguments. In the special case of recursive calls, either directly or indirectly, a two-pass analysis is required to account for recursion. The first pass analyzes the method invocation ignoring the recursive call. The second pass considers the recursive call with the access permissions obtained from the first pass.

```
int f(int n) {
    if (n < 2) { // read(n), control(f)
        return n; // write(return), control(f)
    }
    int a = f(n - 1); // call(f), read(n), write(a)
    int b = f(n - 2); // call(f), read(n), write(b)
    return a + b; // read(a), read(b), control(f), write(return)
}
```

Listing 3.1: Examples of Signatures in Fibonacci Program

Conservatively considering the union of branching instructions can limit the potential parallelism. Thread-Level Speculation (TLS) could be used to increase the

parallelism at the cost of introducing a runtime penalty for keep track of transactional memory, as well as for rolling back when conflicts occur. Given the discouraging results of STM solutions, a conservative and strict parallelization was applied, but this methodology can also be applied to generate TLS parallel programs.

Another advantage of not using STM is that IO is supported by this model. All the IO in the program is considered to be in one single global datagroup. This can be a bottleneck in the cases of several tasks writing or reading to different files, for instance. This bottleneck can be removed by annotating the IO methods with signatures. One example would be the writing to a file, that would required a *write(f)* permission for that file only, allowing it to execute in parallel with a `System.out.println()` method invocation.

3.2.2 Parallelization

Although this model supports parallelization at every AST node, the focus of most parallelism in procedimental and OOP languages is in method invocations, and loops. The parallelization of loops is different according to the interdependencies among iterations, thus resulting in different parallelization methods for DO-ALL and DO-ACROSS loops. While loops are parallelizing in a specific manner, the parallelization process of invocations can be applied to any other AST node. This model also considers nested parallelism among loops, invocations and any mix of the two.

Parallelization of Invocations

The parallelization of method invocations consists in starting the invocation asynchronously as soon as possible, blocking when the result is needed and the invocation has not finished yet. Futures (Swaine et al., 2010) are a representation of this concept. The invocation is started as soon as all dependencies are available, wrapped in a future object, and the invocation is replaced by a *get* call to the future object.

Listing 3.2 shows the parallelization of the code in Listing 3.1, without performing the optimization for recursive calls. A sequential version is always available, in order to use Lazy Task Creation (Mohr et al., 1991) for granularity management. The decision between executing the sequential version or the parallel version is left to the runtime, where information such as the load of the system is available.

```java
int f(int n) {
    if (RuntimeManager.shouldSeq())
        return jpar_sequential_version_of_f(n);

    if (n < 2) {
        return n;
    }
    Future<Integer> b_tmp = new Future<Integer>(task -> f(n-2));
    Future<Integer> a_tmp = new Future<Integer>(task -> f(n-1));
    int a = a_tmp.get();
    int b = b_tmp.get();
    return a + b;
}
```

Listing 3.2: The Fibonacci program translated with futures, without considering the special case of recursion.

The first invocation of `f` is wrapped in a future call and replaced by a `get()` call to the future object. The future is created as soon as possible. In this case, the **if** statement has a control dependency on the current method, which prevents the future from being instantiated before. The second `f` call is also converted into a future that is created as early in the method as possible: just after the **if** statement for the same reasons.

When the future object is instantiated, the task is marked for execution and an available thread may start to execute it. When the `get()` method is called, the current task awaits for the execution of the task and reads its result.

The main decision is where to introduce the future creation, maximizing parallelism while keeping the same semantics of the original program. The proposed requirements for starting the execution of a future task are:

- Must not be declared before the declaration of all accessed variables;
- Must not be declared before an expression which has a **control** access permission over the current method (return statements);
- Must not be declared before an expression which has a **control** access permission over the block (break, continue);
- Must not be executed before an expression that has a **write** permission over any variable accessed inside the lambda;
- Must not be executed before an expression that has a **read** permission over any variable that is written inside the lambda;
- Must be before its original position and, consequently, the future `get()` call.

These requirements can result in either **soft** or **hard dependencies**. An hard dependency represents an AST node that must precede the future creation. A soft dependency represents a task that must have completed before the future is executed.

The first three and the last requirements are strict, and they are considered hard dependencies. An hard dependency of a task is an AST node that must precede the

future creation. One example of this is the declaration of a variable that is used in the future body, which would resulting in invalid source code if preceded by the future declaration.

The other requirements can result in either hard or soft dependencies. If the dependent AST node was already parallelized into a future, than that dependency is considered a soft dependency, and does not need to precede the future creation. Instead the future creation of the dependent task must precede the future creation of the task at hand, and there must be a scheduling dependency between the two tasks. Listing 3.3 exemplifies a soft-dependency between a task that reads a variable written by another task. In the case of the parallelization of `f(i)`, `w.get()` is a soft-dependency, and the creation of `w` is an hard dependency, because that task is required to be listed as a dependency for the future creation of `r`. The requirement that the future `r` will only execute after `w` is completed will be handled by the runtime system.

```
int raw(int n) {
    int i = 0;
    Future<Void> w = new Future<Void>(task -> { i++; });
    Future<Integer> r = new Future<Integer>(task -> f(i), w);
    w.get();
    return r.get();
}
```

Listing 3.3: An example of a soft dependency between task r and w

The distinction between hard and soft dependencies is used to schedule tasks to the runtime even before they can be executed. This early schedule allows the runtime system to have more information about its future parallelism to perform dynamic optimizations, such as granularity control decisions.

Algorithm 1 describes how to find the ideal position to create the future task, determining hard and soft dependencies of task, based on the access permissions. Let us consider θ as the function that, for an AST node, returns its access permission set, **meth** the method in which the invocation is found, **node** the invocation being processed, **stmt** the statement being analyzed and **block** is the current block being analyzed. *block* starts as the most outer scope (the method body) and moves deeper until it is the scope block in which the invocation is in. This order attempts to schedule the task for execution as soon as possible, even outside the current block if possible.

Finally, as seen in the first lines of Listing 3.2, each method containing parallelized tasks has a dynamic check for granularity control. This check will allow the runtime to execute a sequencial version instead of the parallel version of the method. The criteria to choose between each approach will be discussed in the following sections.

Parallelizing DO-ALL loops

Besides invocations, for and for-each loops are also targets for parallelization. DO-ALL and DO-ACROSS loops are handled differently. Iterations of DO-ALL loops are independent and have dependencies only with code outside the for-loop.

In order to verify if a loop is DO-ALL or DO-ACROSS, array and arraylist

Algorithm 1 Algorithm to find hard and soft dependencies for a task

```
harddep ← None
softdeps ← ∅
for stmt ∈ block do
    if control(meth) ∈ θ(stmt) ∨ control(block) ∈ θ(stmt) then
        harddep ← stmt
        continue
    end if
    if ∃a, [read(a) ∈ θ(stmt) ∧ write(a) ∈ θ(node)] ∨ [write(a) ∈ θ(stmt) ∧ read(a) ∈
θ(node)] ∨ [write(a) ∈ θ(stmt) ∧ write(a) ∈ θ(node)] then
        if isTask(stmt) then
            softdeps ← softdeps ∪ stmt
        else
            harddep ← stmt
        end if
    end if
    if stmt ⊃ node then
        break
    end if
end for
```

accesses are annotated with indexed datagroups. For example, `array[i] = 1` will have a permission $\mathsf{write}(\mathsf{array}[\mathsf{i}])$ that is treated as a $\mathsf{write}(\mathsf{array})$ for all code outside loops. Inside loops, the indexed permission is used to verify if reads and writes are independent. The verification performed is rather naïve, as it only considers for-loops in which the iteration variable is monotonic. Nevertheless, the polyhedral model can be applied, obtaining a better degree of parallelism in more complex loops.

```
// for (long i=0;i<n; i++)
// array[i] = sin(i);

Future<Void> loop = ForHelper.forLong(0L, n, (Long i) -> {
    array[i] = sin(i);
});
```

Listing 3.4: An example of a DO-ALL loop parallelized.

The iteration of the loop is converted into a lambda expression, which is passed as an argument to a runtime helper that will handle the work-partitioning. This helper will return a future that can be used as a soft dependency. Listing 3.4 shows an example of a simple DO-ALL loop parallelized using the runtime helper method.

Parallelizing DO-ACROSS loops

In order to parallelize DO-ACROSS loops, the loop must contain the same conditions as for DO-ALL, but write permissions can be allowed inside the loop, namely operations that are commutative and associative. By default, the compiler considers for these tasks the operators `+,-,*` and the methods `Math.min(), Math.max()`. How-

ever, any other operation can be annotated as such, and will be parallelized using the same mechanism.

The compiler generates a Map-Reduce operation for the DO-ACROSS loop. The loop body is converted into a lambda function, saving memory writes inside the lambda, instead of on the shared variable. The reduce operation will aggregate the results from the lambda execution. This Map-Reduce operation also returns a Future for later use as a soft dependency. Listing 3.5 shows an example of the approximation of the π value, using the Map-Reduce helper to perform the work-sharing and aggregation of results.

```
// for (long n=0;n<dartsc; n++)
// scored += inside(randomPosition(), randomPosition()) ? 1 : 0;

int pi(long dartsc) {
    if (RuntimeManager.shouldSeq())
        return jpar_sequential_version_of_pi(n);
    long score = 0
    Future<Long> calc = ForHelper.forLongReduce(0L, dartsc, (Long n) -> {
        return inside(randomPosition(), randomPosition()) ? 1 : 0;
    }, ForHelper.longSum);
    score += calc.get();
    return (4.0 * ((double)(score))) / ((double)(dartsc));
}
```

Listing 3.5: The Pi program translated with futures.

3.3 Implementation

The proposed automatic parallelization model has been implemented in the JPar compiler (Fonseca et al., 2016). This compiler was a complete rewrite of an initial version (Rafael et al., 2014), improving the performance, making generated code more readable and using a simpler representation of dependencies.

The JPar compiler translates sequential Java into parallel Java code that targets the Æminium Runtime (Stork et al., 2014). Since the JPar compiler delays several decisions to the runtime, a shim library was introduced to support the usage of JPar futures on top of the Æminium Runtime. Futures are wrappers of code that begins executing asynchronously on possibly another thread, and the result of which can be obtained later. If the result is not yet available, the result request will block until the future is complete. JPar futures differ from regular futures as they also hold soft dependencies to other futures. A future will only begin executing after all soft-dependencies are finished.

The JPar compiler is built on top of the Spoon compiler framework (Pawlak et al., 2015), which handles the parsing and code generation from and to Java. The Æminium Runtime is a task-based runtime library for Java that executes tasks on top of a work-stealing scheduler. The architecture of the implementation is depicted in Figure 3.1, with the JPar compiler translating Java to parallel Java, the ÆminiumGPU compiler generating GPU-compatible versions of data-parallel operations. The compiled bytecode is executed on the JVM using the Æminium and ÆminiumGPU runtimes for multicore and GPU execution. The ÆminiumGPU

modules are opcional and make usage of the automatic parallelization of the JPar compiler.

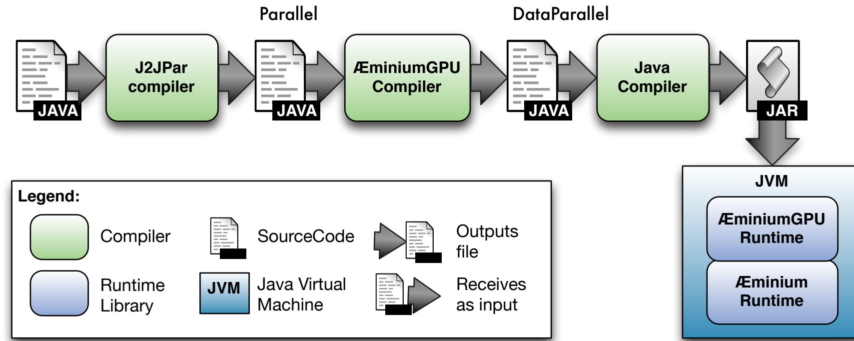

Figure 3.1: Information Flow in the Æminium Framework

3.3.1 Automatic Parallelization

An overview of the compilation phases of JPar can be seen in Figure 3.2. Parsing and code generation are handled by the Spoon compiler. The other three phases support the automatic parallelization model.

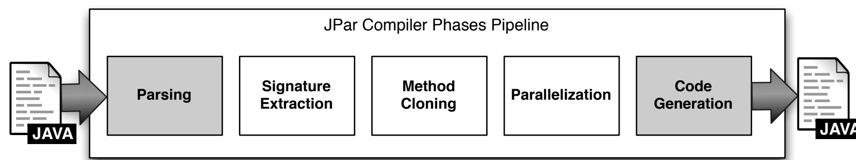

Figure 3.2: JPar Compiler Phases

Signature extraction consists on keeping track of **read**, **write** and **control** access permissions. This is done in a bottom up transversal of the AST. Variable accesses generate **read** or **write** permissions over that variable. Return, continue and break statements generate a **control** permission over the corresponding block. Try-catch statements also generate **control** permissions, resulting in a conservative approach to exception handling. Fonseca and Cabral (2012) explain in detail the issue of exception handling in automatic parallel languages and compilers.

The method cloning phase creates a shadow copy of each method, in order to keep a sequential version of each method. This copy allows the Runtime system to switch to the sequential mode, reducing the task granularity. The advantage of doing it is reducing the scheduling overhead when there is already enough parallelism in the runtime. In order to perform this choice, the parallel version of each method starts with a runtime check condition that will switch to the sequential version. Listing 3.5 has an example of this check.

The parallelization step transverses the AST identifying which nodes can be parallelized. The current implementation focuses on loops and invocations, but it can be applied to any other node. The decision whether to parallelize is based on the access permissions of that node. Additionally, there is a simple static granularity

control in place. In order to prevent the creation of tasks for extremely light weighted methods, methods will be parallelized if they have at least a given number of instructions or method calls. The limit of number of instructions is a simple heuristic to prevent creating tasks for methods that will execute faster than the creation of a future. This value has a default value of 10, but can be increased according to the specific machine. This granularity control is referred as Naïve Granularity for evaluation purposes.

Additionally, it is possible to reduce the overhead in recursive methods. The usage of lambdas or anonymous inner classes has a performance penalty. The JPar compiler extracts recursive methods into static classes that can be passed as future arguments. Static classes have a lower performance penalty that lambdas, which results in a lower scheduling penalty in frequent recursive calls.

3.3.2 Runtime Execution Support

Tasks and Dependencies

The Æminium Runtime is a Java library that exposes APIs for expressing asynchronous execution of code. The Runtime is composed of modules that allow for an efficient execution of the source code, by leveraging the hardware threads available.

The core concept of the Æminium Runtime is the task as a representation of code that can execute asynchronously. Tasks have a body, which can be represented as a lambda, an anonymous inner class or as a regular class (useful when doing recursive calls). Tasks are also defined by a set of dependencies on other tasks. If A depends on B, it means that A cannot execute before B is completed. Tasks can also have a parent task to represent subcomputations. If A is the parent of B, then A is only completed when both the body of A and task B are completed. An example task graph can bee seen in Figure 3.3, which represents 6 tasks with dependencies among them, as well as parent-child relationships.

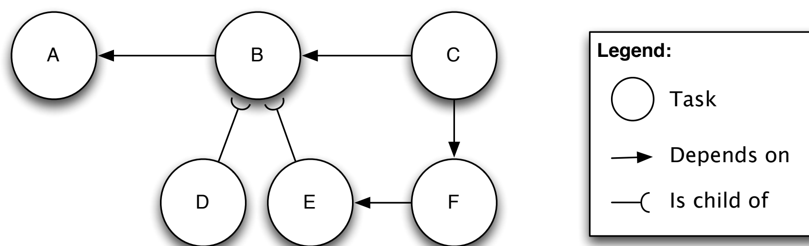

Figure 3.3: Example of a task graph.

Each task can be of one of three types:

- **Non-Blocking Tasks** are all operations that are purely computational.
- **Blocking Tasks** are tasks that have at least one operation that performs input or output, such as disk reads/writes, communication over sockets or other interactions with the Operating System.
- **Atomic Tasks** are tasks that cannot execute at the same time as other Atomic Tasks that share the same Data Group. The Data Group acts as the lock that

each atomic task must acquire before executing and release after executing. However, two Atomic Tasks with different Data Groups can execute concurrently.

Figure 3.4 shows the lifecycle of tasks inside the runtime. When a task is submitted to the runtime, along with its dependencies, the runtime analyzes if the dependencies are already met. If so, the task is sent to a queue for execution. If not, no action is performed at this point.

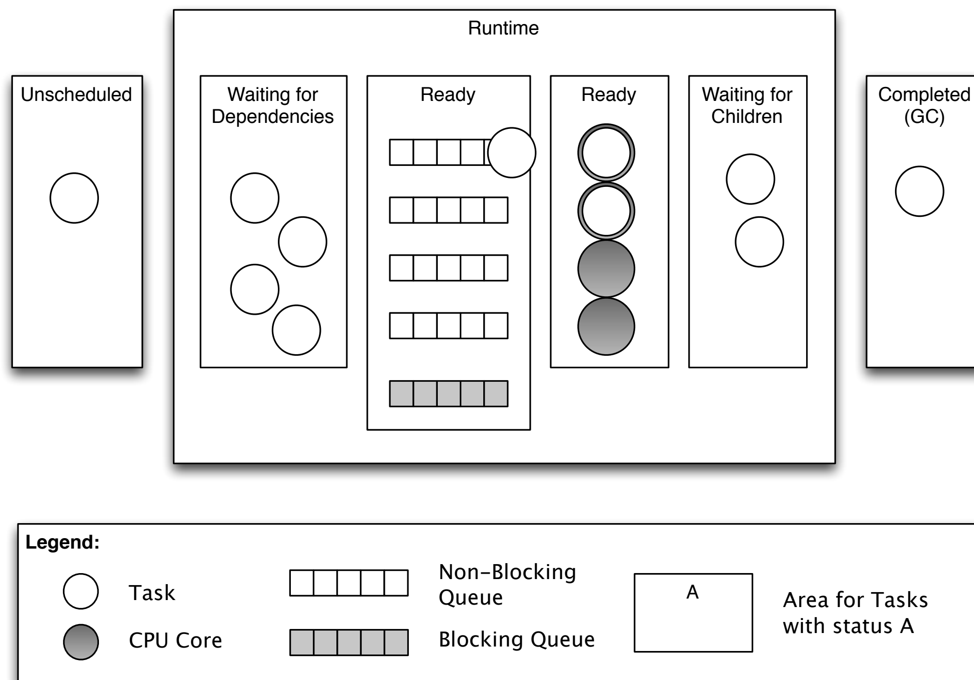

Figure 3.4: Runtime Areas for the different phases of the Task Lifecycle, for a quad-core machine. Tasks go through the states from left to right.

The Æminium Runtime does not create a thread for each task, as the overhead would be very noticeable. Instead, there are always n work-stealing threads running, one for each processor core available in the machine (this number can be configurable per program execution). These threads are responsible for executing tasks that are ready. In order to reduce the contention on the queues, each thread has its own queue. When a thread finds its queue empty, it will *steal* a task from the queue of other thread. The Æminium Runtime has different stealing algorithms: tasks can steal from a random queue, steal from the largest queue or from tasks with more dependant tasks. While the stealing algorithm can be configured per program, the default algorithm is stealing from the largest queue as it balances irregular programs.

Non-Blocking and Atomic tasks are stored on those regular queues. Since Blocking Tasks can take a long time to execute, given the dependencies on the kernel, such as reading from sockets or files, these tasks are added to a special queue, which is processed by an independent thread-pool. This avoids blocking Work-Stealing workers with Blocking tasks.

When a task completes, it will check if there are any child tasks that were scheduled during execution and belong to the logical concern of the current task.

When all child tasks have finished, the task is marked as completed, and it will notify both the dependent tasks and parent task that they do not need to wait for it anymore.

Executing DO-ALL and DO-ACROSS loops

Since each iteration may take a different time to execute, DO-ALL and DO-ACROSS loops cannot simply be divided in equal parts and executed in slices. In order to balance loads across cores, a more dynamic approach is required. Two different approaches are available: Binary Splitting and Lazy Binary Splitting (Tzannes et al., 2010). Binary Splitting divides the current range in two if the Decider module considers that it is still useful to create new tasks. If not, it executes the current range iterations immediately. With Lazy Binary Splitting, there is a parameter PPS which represents how frequently should the range be checked for division in half. A PPS of 3 means that every 3 iterations the runtime checks if the remaining range should be split in two.

For DO-ACROSS loops, the Map-Reduce approach is better than creating lock-protected atomic blocks, since it avoids locking contention when all threads want to access data. However, only associative and commutative operations can be converted into Map-Reduce. This is not a large problem, as most data-intensive computations are based on those operations, such as `+,*, -`.

3.4 A Cost-Model Granularity Approach

One of the main concerns about automatic parallelization is managing task granularity, specially when the workload depends on runtime data. For those cases, runtime dynamic granularity control mechanisms, such MaxTasks and MaxLevel, can be used. Chapter 8 will focus on analyzing different dynamic cut-off mechanisms to understand which one to apply to each program.

However, an hybrid approach can be taken: the compiler can build a cost model reasoning about the code inside a task, and that information can be used during runtime to estimate the costs of executing a method sequentially or dynamically.

```java
public class Example {
    public double csc(double a) {
        return 1/Math.sin(a);
    }

    public double loop(int N) {
        double a = 0;
        for (int i=0; i<N+1; i++) {
            a += csc(i);
        }
        return a;
    }
}
```

Listing 3.6: Example Java code for illustration of the Cost Model used.

In order to model a Java program, the cost of a node cannot be a single final

value. The reason for this is that the cost depends on the machine in which it will execute and it may depend on runtime information about the program. Instead, the cost of a node is a weighted sum of its components, in which the weights can be represented as AST nodes, in this case Java expressions. Considering the example in Listing 4.1, it is clear that it is not possible to know the cost of the `for` loop during compilation because `N` is not defined in this code. The cost for the function `csc` also depends on how fast the machine can execute the `sin` function. Given this information, we can model the cost of the `csc` function as Equation 3.1. The function costs one operation (the division) and one call to the standard library `Math.sin(double)`. The cost of the for-loop is modeled as Equation 3.2. It allocates one variable (`i`), and $N+1$ times, it repeats 3 accesses to the variable `i`, one access to the variable `N` and one access to the `a` variable. Additionally, it also repeats the `+=` operation, the `<` and `>` operations in the condition and the `++` operation in each loop iteration, as well as the cost model for the `csc` function.

$$C(csc) = (1 * C(op)) + (1 * C(Math.sin(double))) \tag{3.1}$$

$$\begin{aligned} C(for) = 1 * C(alloc) + (N+1) * (5 * C(access) + \\ 5 * C(op) + (1 * C(Math.sin(double))) \end{aligned} \tag{3.2}$$

This model simplifies the actual cost. The `+` and `*` operators have different costs but the difference is insignificant for the purpose of evaluating if the operation is expensive enough compared to the parallelization overhead. The scheduling of a task performs work that is several times longer than any of the operators We are also ignoring the difference between heap and stack allocation, which we have found to be irrelevant for our goal.

$$C(node) = \begin{cases} C(type(node)), & \text{if node is leaf} \\ repetitionNode * \sum_{child} C(child), & \text{if node is loop} \\ \sum_{child} C(child), & \text{otherwise} \end{cases} \tag{3.3}$$

Equation 4.1 represents the cost model for a node. C is the function that returns an expression for a given AST node. If a node is a leaf, it will add a variable representing the type of that node. Possible node types are:

1. alloc, for memory allocation;
2. access, for variable reads and writes;
3. arrayaccess, for reads and writes to an element of an array;
4. op, for binary and unary operators;
5. if, for if and conditional instructions;
6. class#method(args), for each method in java.lang.Math, java.util.Random, java.util.concurrent.ThreadLocalRandom, java.util.List, java.io.PrintStream.

If a node is not a loop, then the cost is the sum of all child nodes. If a node is a `for` loop, the weight for that expression will be the estimated number of iterations of the loop. If it is possible to obtain the number of iterations statically (in cases of `for(int i=0; i<100;i++)`, then the value (100) is used. If it is not possible, the AST node is used in the comparison of the stopping condition, subtracting the initial value.

The cost expression may depend on variables, such as `N` in the Listing 4.1. For the cases in which variables are not modified after the initial value is set, that value is used. For other cases, the dependency is represented as a dependency on the AST node. The estimation is delayed until runtime, when the AST node will be evaluated, resulting in one value, completing the estimation.

Before making the decision to parallelize or not, it is necessary to apply the variable values in the expression. Values for *alloc*, *access* and other node types are previously recorded using a simple benchmark program that executes each operation 1000 times and calculates the average. These values are machine dependent, which requires the compilation to be performed in the machine, or using a configuration file retrieved from the machine.

3.5 Evaluation

In this section, we evaluate the proposed model using the JPar implementation, using the following alternatives:

- **JPar (No Granularity Control)** - The JPar compiler without any granularity control , maximizing the parallelism expressed;
- **JPar (Naïve Granularity)** -The JPar compiler with a naïve parallelization decision available in the Æminium Runtime;
- **Manual** - A manual parallelization performed on top of the Æminium Runtime. The division of loops is done using the Lazy Binary Splitting algorithm. This version is used as a baseline for how much parallelism can be extracted from the program.
- **OoOJava (sequential) and (parallel)** - The OoOJava compiler, the state of the art in Java automatic parallelization, which compiles Java to C code.

Experiments were executed on two machines running Ubuntu 14.04 server 64bits and a Java Hotspot 64-bit Server JVM. One machine, *server24*, has 12 cores and 24 hyperthreads, and 24GB of RAM in two NUMA regions auto-balanced. The other machine, *server32*, has 16 cores and 32 hyperthreads, and 32GB of RAM also in two NUMA regions auto-balanced.

The Æminium Runtime was configured with a number of threads equal to the number of hyperthreads available on the machine.

Since results on both machines were very similar, we will present the results from *server24* and will refer to *server32* when they differ. Given the non-deterministic behavior of memory allocation, garbage collection and work-stealing, experiments were executed 7 times and the median value was used to reduce the interference of external aspects. Executing more than 7 times would not change the median value

beyond reasonable error. Programs had a time limit of 10 minutes for execution, a value much larger than the sequential version. This limit was imposed after some programs with no granularity control would take days to execute.

3.5.1 Benchmark Programs

In order to evaluate the performance of the compiler, we used the sequential version of 6 programs from the Æminium Benchmark Suite (Fonseca, 2013). The configuration for each program is described in Table 3.1, as well as the parallelization performed for each program.

DO-ALL programs spend most of the execution time in loops that execute independently. DO-ACROSS programs also have loops, but they have writes to the same value, which has to be synchronized. Recursive programs are defined by recursive methods. This benchmark includes programs that are hard for compilers to automatically parallelize. For instance, N-Body is a DO-ALL program, which can easily be split in several chunks. However, N-Body is a skewed program, in which early iterations will have a heavier workload than laters ones. Recursive programs are also difficult to parallelize because getting an estimation of its cost depends on the input parameters.

Since each program represents different parallelization scenarios, the performance of each program will be analyzed.

Program	**Parallelism**	**Input size**
BlackScholes	DO-ACROSS	100000
FFT	Recursive	8388608
Integrate	Recursive	s=-2101, e=1700, error=10^{-14}
MergeSort	Recursive	n=251658240
N-Body	DO-ALL	it=10, bodies=25000
Pi	DO-ACROSS	n=1500000000

Table 3.1: Description of the programs used in the benchmark

3.5.2 Binary vs Lazy Binary Splitting

Programs that have for-loops parallelized can generate tasks in two different ways: Binary Splitting or Lazy Binary Splitting. For Lazy Binary Splitting, we used the recommended value of 3 for the PPS parameter, and a higher value of 10 for less frequent decisions. Figure 3.5 shows the speed-up over sequential programs of the three approaches in program with loops. The Lazy Binary Split version achieved best results in the BlackScholes program, running in less than half of the time as its Binary Split counterpart, but could not complete the other programs within the predefined time-out, resulting in no speed-up. The BlackScholes program is made of DO-ALL loops with very lightweight tasks, which allows for the executing between splits to be efficient. The conclusion is that Binary Split is a conservative approach that can be used for any program, while Lazy Binary Split can be used to achieve best results on programs with lightweight iterations.

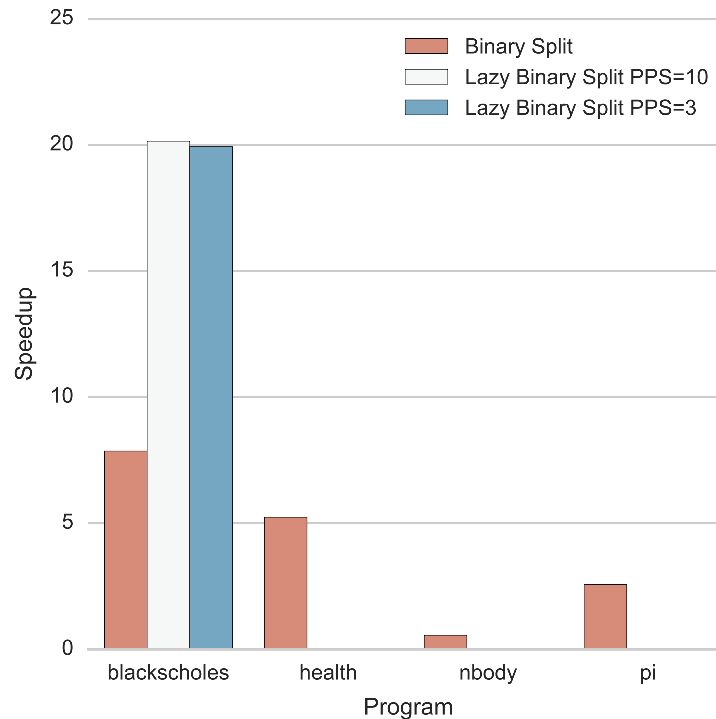

Figure 3.5: Average speedup of Binary Split and Lazy Binary Split (PPS=3 and 10) versions of the programs with loops.

3.5.3 Nested Loops

NBody performs a simulation predicting the individual motions of 50000 Jovian planets after three iterations. Two nested for-loops comprise almost all of the work done by the program. Granularity of nested loops is difficult to control, especially since the size of each loop is only known during execution (the number of bodies is received as an argument to the program). In this case, the cost-model generates an expression to predict the cost of each loop, based on a runtime variable. During execution that variable is replaced by the actual value and the decision how to divide the loop is made.

The speedup achieved by each version can be seen in Figure 3.6. The OoOJava compiler could not successfully compile this program and the naïve approach exceeded the time limit. The version without any granularity control was not able to achieve speedup because it generated too many tasks that would cause the system to spend more time scheduling tasks than executing them. The manual version with two parallel loops also relied in the runtime decision for the loop partition (the problem dimension was unknown to the programmer) so it did not perform as well as the Cost-Model version.

3.5.4 Invocation Inside Loops

The Pi benchmark estimates the value of pi using a Monte-Carlo simulation of dart throwing inside a square. In this program, each dart can be thrown in parallel, and each target coordinate of the dart can also be obtained in parallel.

Figure 3.7 shows the results of the different versions of Pi. The versions of JPar

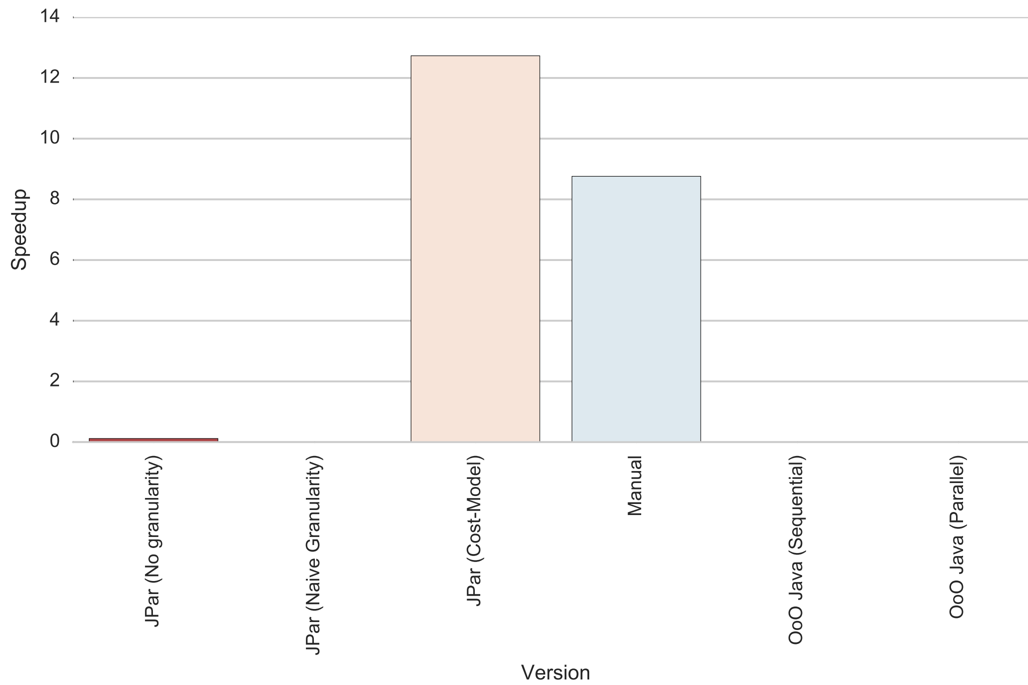

Figure 3.6: NBody execution on the machine *server24*

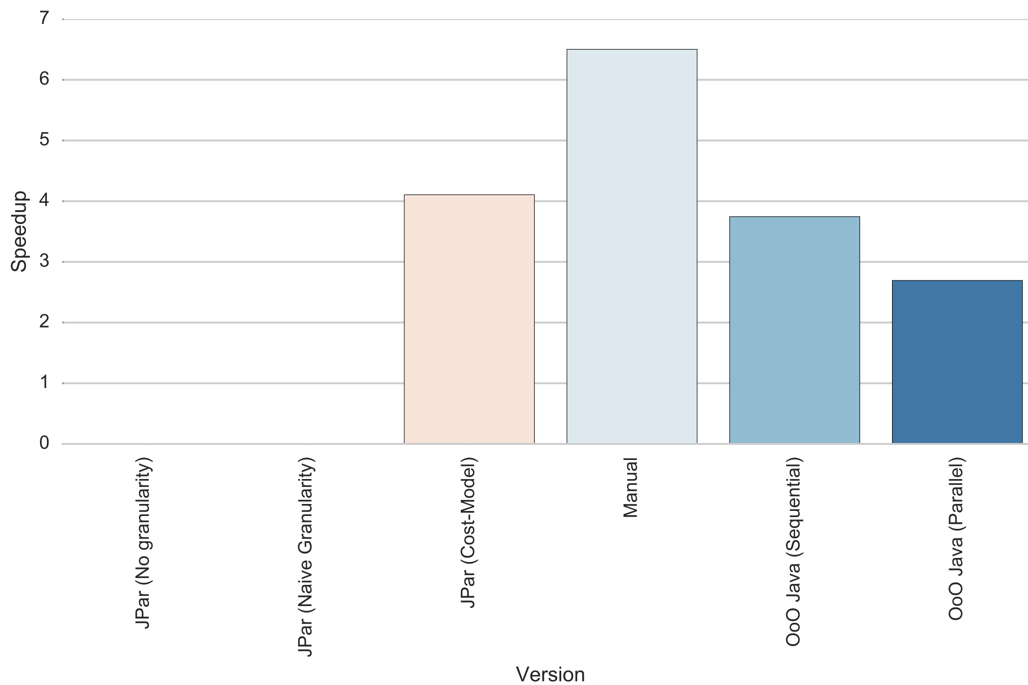

Figure 3.7: Pi execution on the machine *server24*

without the Cost-Model are not capable of understanding that the generation of each coordinate in parallel has a very large overhead. The version with the Cost-Model outperforms the OoOJava versions, but it is not as good as the manual parallelization approach. Figure 3.8 shows the CPU and memory usages of the Pi program on both JPar and OoO Java versions. The memory usage of the JPar version shows that it is creating more parallel tasks than the OoO version, which

is also verified in the CPU usage. The final result is that the JPar executes faster with a better CPU occupation.

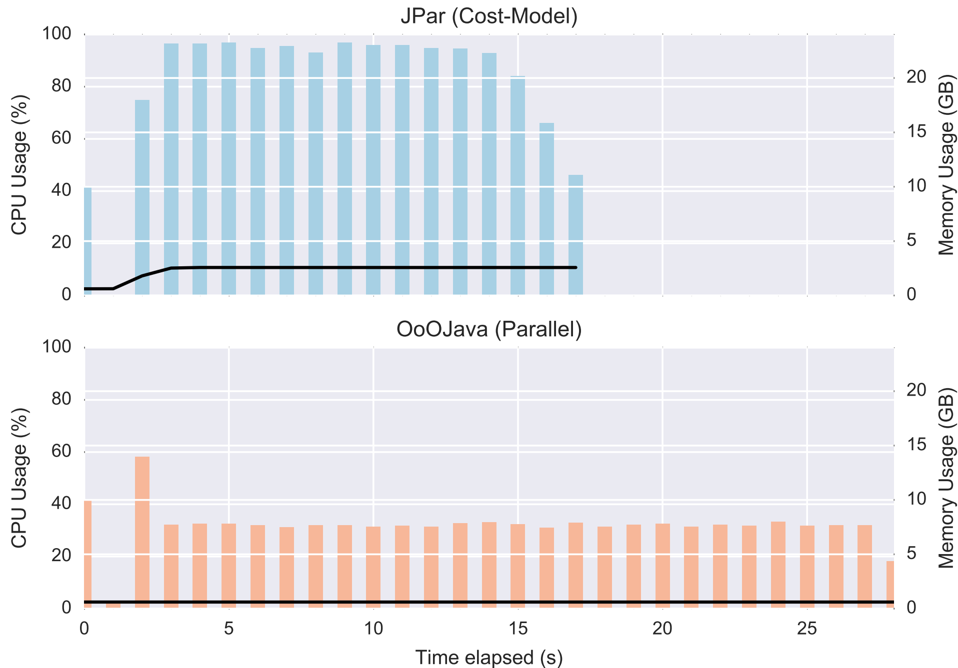

Figure 3.8: CPU and Memory usages of Pi on the machine *server24* in JPar and OoOJava

The limitation of parallelizing only one level in OoOJava, which was not beneficial in the NBody benchmark, is now the right choice. However, the parallelization performed is still not enough to produce a speedup compared to the sequential version of OoOJava.

BlackScholes also performs different Monte-Carlo simulations of the BlackScholes formula. Parallelization can be achieved by parallelizing the loop, but also from parallelizing some functions inside the for-loop. Similarly to the Pi benchmark, the innermost functions are not worthwhile to parallelize.

Figure 3.9 shows the speedup of different versions of the BlackScholes benchmark. Results from OoOJava are missing, because OoOJava was unable to compile the same program. All version of the JPar compiler were able to achieve a speedup, with the Naïve version having a lower speedup because of unnecessary overheads in runtime decision. The cost-model outperformed the other approaches, including the manual approach, since it generated a fixed number of chunks, instead of using a dynamic approach like Lazy Binary Splitting. Since there were two main loops to parallelize, one with lightweight iterations and other with another non-parallelizable loop inside, the Cost-Model generated two different chunk sizes for each loop.

3.5.5 Recursive Calls

The FFT program performs the Fast-Fourier Transform of an array of complex numbers. The parallelism can be extracted from the different for-loops, but there is also fine-grained parallelization inside complex objects, such as the parallel sum of components of two complex objects.

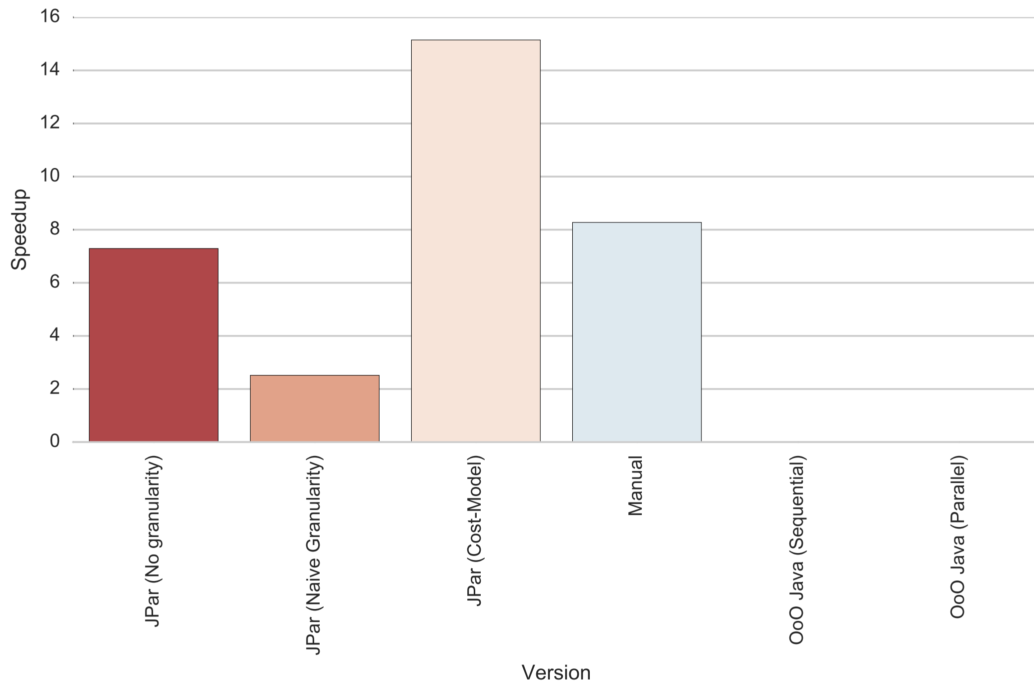

Figure 3.9: BlackScholes execution on the machine *server24*

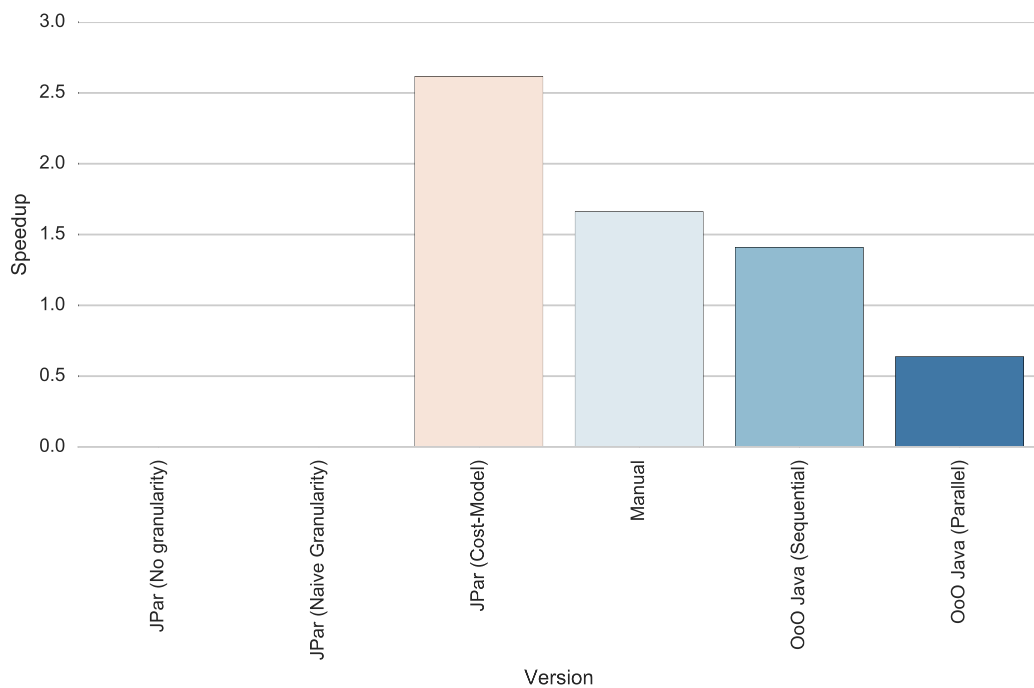

Figure 3.10: FFT execution on the machine *server24*

Figure 3.10 shows the result of the FFT benchmark in the different versions. The fine-grained parallelism inside the Complex objects prevents the JPar compiler from having speedups, and the Runtime decision mechanism is to heavy to help. The Cost-Model version achieves a good speedup, even better than the human version because it limits parallelism when the memory is reaching its limit, based on the prevision of memory allocation. OoOJava is unable to get a speedup compared

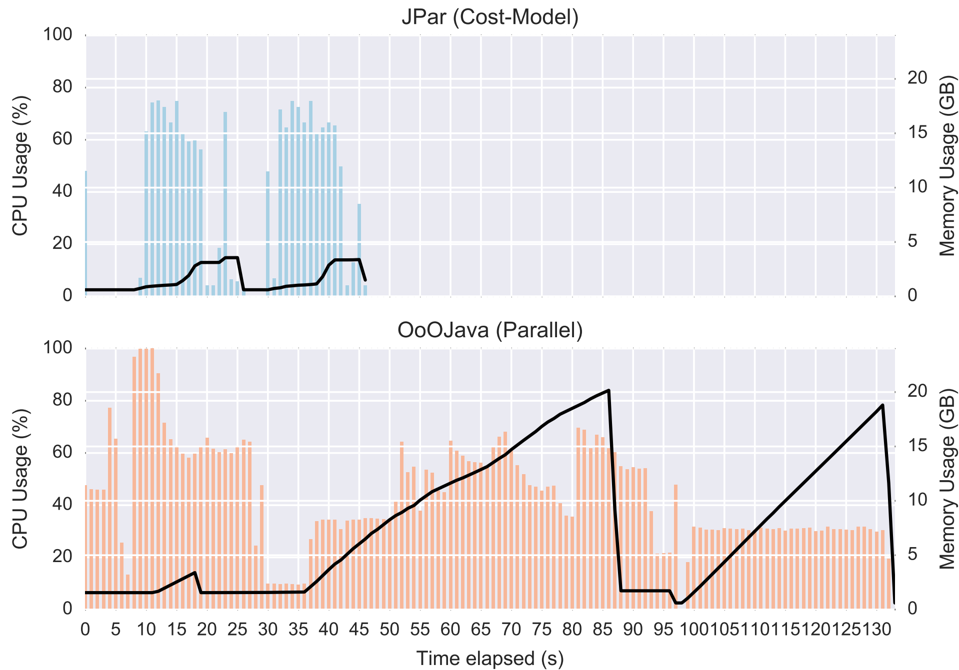

Figure 3.11: CPU and Memory usages of FFT on the machine *server24* in JPar and OoOJava

to either sequential version. Figure 3.11 shows the CPU and Memory usage of the FFT program for OoOJava and JPar with the aggregation mechanism enabled. Since the parallelization of FFT requires memory, it is possible to see that the JPar version results in more coarse grained-tasks, while the OoO version continues with the parallelization, without increasing the CPU usage, compared with the JPar version. It is also noticeable that the JPar version finishes early, showing that the grain level obtained suits this program better.

The Integrate program calculates the approximation of the integrate of $(\mathsf{x}^2 + \mathsf{1}) * \mathsf{x}$ using the trapezoidal rule. Like previous examples there is the recursion parallelism, as well as a fine-grained parallelism that will not benefit performance.

Figure 3.12 shows the performance of Integrate on the *server24* machine. All versions using the Jpar compiler have a low speedup compared to the human approach. The JPar compiler delays some of the decisions to the runtime, and that adds some overhead to the decision, that the manual version does not have. OoOJava, on the other hand, does not have any speedup. Figure 3.14 shows the CPU and memory metrics for JPar and OoO, showing relatively equal CPU utilization for the duration of the programs. Thus, the higher granularity of the OoO version does not result in higher CPU occupation or speedup. In fact, the program runs slower.

Figure 3.13 shows the performance of the same program, but on the *server32* machine. On this machine, with slower processors, the JPar compiler generates programs with lower speedups than on *server24* without any granularity control. The version with runtime granularity decision does not finish within the time limit. The Cost-Model version actually has better speedups than on *server24*. This program behaves differently according to processor speed and number of cores. This is the reason why the Cost-Model takes into account the characteristics of the machine.

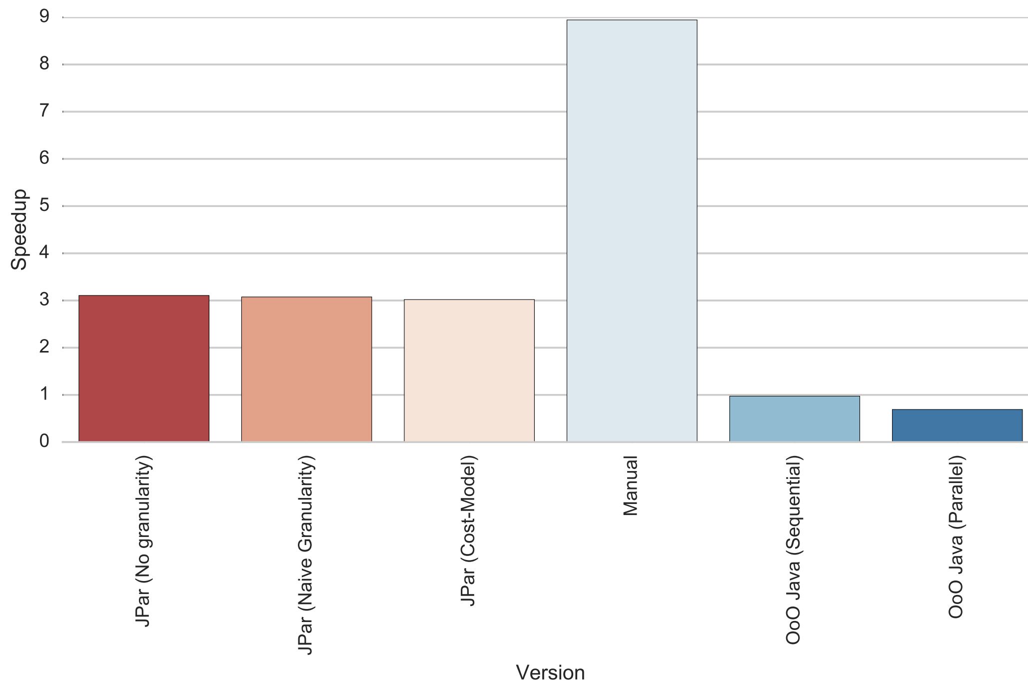

Figure 3.12: Integrate execution on the machine *server24*

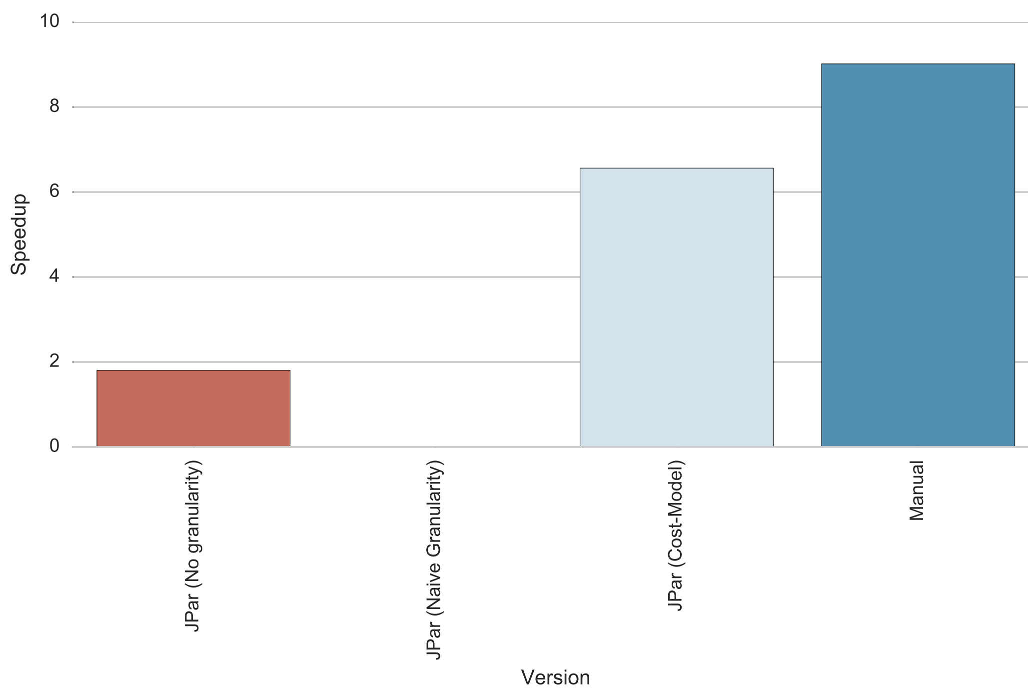

Figure 3.13: Integrate execution on the machine *server32*

MergeSort applies the merge-sort algorithm to an array of integers. It has divide-and-conquer parallelism as well as small loops with small degree of parallelism. Figure 3.15 shows the performance of MergeSort across different versions. Both baseline JPar versions are unable to execute within the time limit because of the fine-grained parallelism. The version with Cost-Model outperforms the manual approach by controlling the memory usage and not parallelizing inner loops. The OoOJava

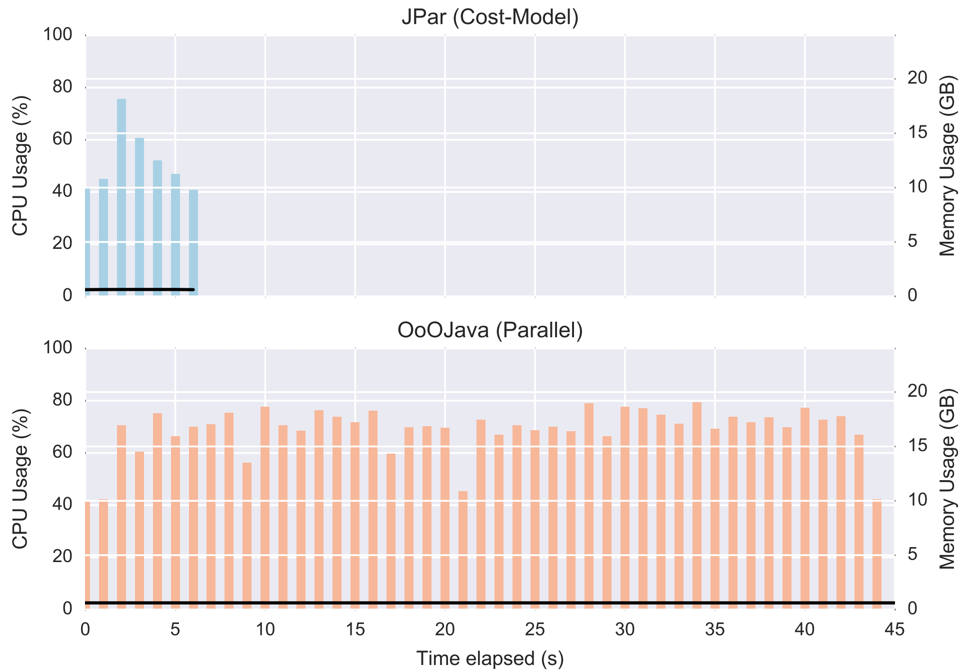

Figure 3.14: CPU and Memory usages of Integrate on the machine *server24* in JPar and OoOJava

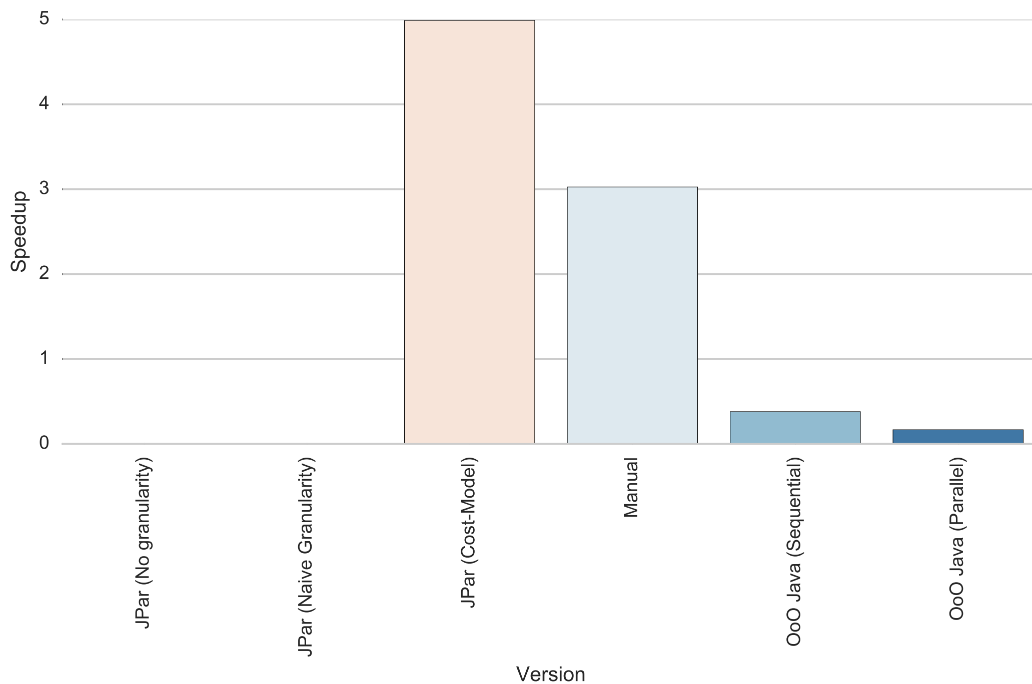

Figure 3.15: MergeSort execution on the machine *server24*

is again unable to obtain any speedup. Figure 3.16 shows the CPU and memory usage of JPar and OoO versions. The JPar version creates less tasks, and more slowly, which can be seen by the memory usage. This leads to an early result. The OoO version creates more tasks, most of them with more overhead than sorting computation, spending more time executing the extra tasks.

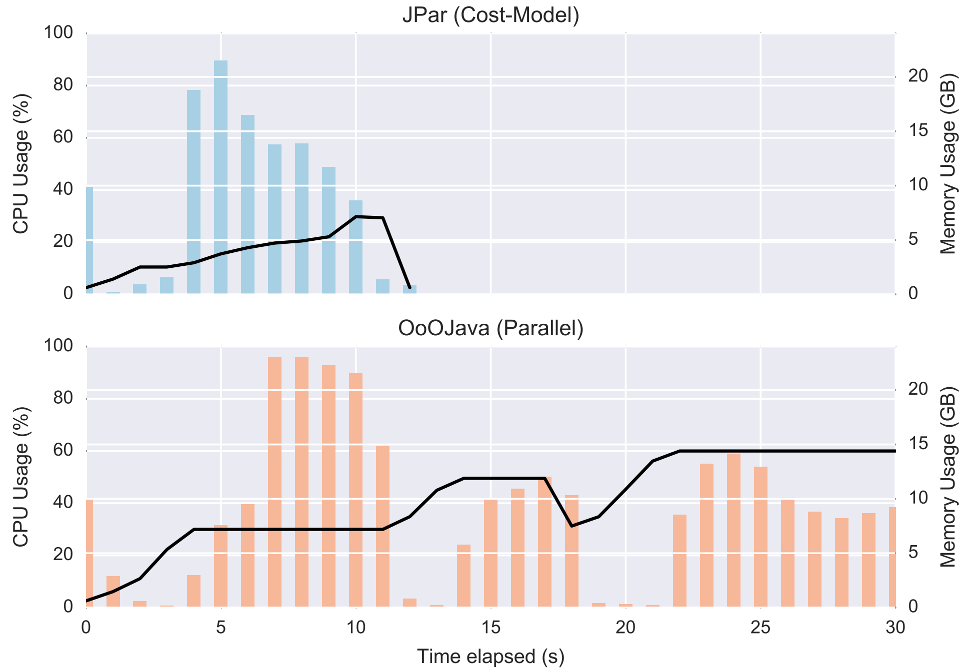

Figure 3.16: CPU and Memory usages of MergeSort on the machine *server24* in JPar and OoOJava

As in conclusion, it is possible to see that the Cost-Model of JPar provides a better granularity than the state of the art approach, OoO Java. In FFT, Integrate and MergeSort, JPar creates less tasks, reducing the overhead in scheduling. In Pi, JPar creates more tasks, but results in an early finish of the algorithm, showing that the granularity chosen is better than by OoO Java.

3.6 Conclusions

This chapter presented a novel approach for automatic parallelization. This approach relies on automatically inferring access permissions from sequential code, and when access permissions assure program semantics will not be modified, AST nodes are parallelized into tasks. In the generated source code, recursive calls and loops can be replaced by future invocations that execute on the work-stealing-based Æminium Runtime. The generated source code is readable and allows developers to understand how parallelization occurs and can have benefits in teaching.

This approach was improved with the introduction of a Cost-Model used to dynamically manage task granularity during execution, using predictions of the sequential and parallel execution times.

These approaches were implemented in JPar, a Java automatic parallelization compiler, and evaluated on several benchmarks. We can conclude that this approach always improves the performance, compared with not using any granularity control, or with delaying the decision to the runtime. It is also shown that this approach provides speedups in all benchmarks, unlike OoOJava, a state of the art in automatic fine-grained parallelization of Java. In some of the applications, the Cost-Model is unable to achieve the same results as a manual approach, but in other cases it

actually improves upon it.

There are some topics that are still left for exploration. In Automatic Parallelization, we have identified exception handling as problematic language feature to efficiently parallelize, which was described in Fonseca and Cabral (2012). Another topic left to explore is how different granularity mechanisms behave with different programs, and this will be covered in the next chapter.

Chapter 4

Automatic Selection of GPU-CPU Granularity

Chapter 3 presented an automatic parallelization model capable of generating programs that can execute on either the GPU or the CPU. However, the decision of which platform to use is left to the programmer. With the goal of automatic parallelization in mind, this Chapter presents a novel approach for automatically selecting the fastest platform for a specific program, based on the characteristics of the program, the input data and the processor. This approach can be used in the proposed model for efficient execution of automatic parallelized programs.

4.1 Introduction

Since GPUs have been user-programmable, scientists and engineers have been exploring new ways of using the processing power in GPUs to increase the performance of their programs. GPU manufacturers acknowledged this alternative fashion of using their hardware, and have since provided special drivers, tools and even models to address this small, but fast-growing niche.

GPUs have became a target of parallel programs because of the number of threads available, higher than any current multi-core processor. In spite of the higher throughput, GPUs carry a high latency due to the asynchronous access to the main memory. GPUs have an independent memory which GPU programs, called *kernels*, access. Regions on the host memory accessed by kernels must be previously copied to the GPU memory, and later copied back to the CPU. Additionally, GPUs threads inside an executing group share a program counter, which causes branching operations to reduce the parallelism in the program.

Overall, the architecture of GPUs can improve the performance of some programs, while decreasing on others. GPUs can efficiently execute programs with high granularity, infrequent branching operations with intensive computations over a large set of data. Deciding to use the GPU for a specific computation is not trivial. Developers that do not understand the programming model and the hardware architecture of a GPU will not be able to perform this decision, and even with a deep knowledge of GPGPU, some profiling may be necessary. In the case of automatic parallelism, the decision must be made without any information from previous executions.

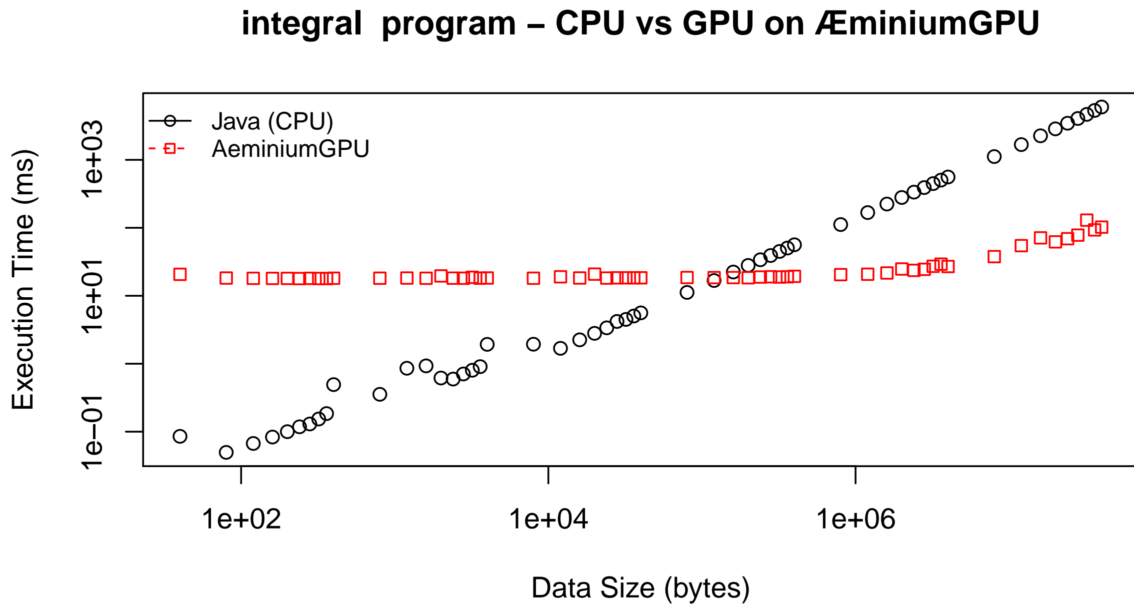

Figure 4.1: Performance of the Integral program on CPU and GPU

```
Double integral = new Range(RESOLUTION).map(new LambdaMapper<Integer, Double
    >() {
  public Double map(Integer input) {
    double n = RESOLUTION;
    double b = Math.pow(Math.E, Math.sin(input / n));
    double B = Math.pow(Math.E, Math.sin((input+1) / n));
    return ((b+B) / 2 ) * (1/n);
  }
}).reduce(new LambdaReducer<Double>(){
  public Double combine(Double input, Double other) {
    return input + other;
  }
});
```

Listing 4.1: Example of Map-Reduce to Calculate the Integral of a Function using the trapezoid method

Listing 4.1 is an example of programs that can execute on the GPU and calculates the integral of $\mathsf{f}(\mathbf{x}) = \mathsf{e}^{\sin(x)}$. This is an embarrassingly parallel problem, which is expressed using a data-parallel approach by means of map and reduce operations. Figure 4.1 shows the execution time of the program in both CPU and GPU for different data sizes. The GPU version is faster after a certain data size and it is able to achieve up to 64 times of speedup. But, note that the threshold from which the GPU performance starts to gain on the CPU is not always the same. The actual threshold value depends of the program logic and even with the hardware being used. Thus the decision whether to run a program on the GPU or CPU is not an easy one.

This chapter expands the ÆminiumGPU (Fonseca, 2011) GPGPU framework,

which is part of the Æminium ecosystem, with a decision mechanism that decides whether a certain operation occurs in the CPU with a coarser granularity or the GPU with a very fine granularity.

4.2 ÆminiumGPU Architecture

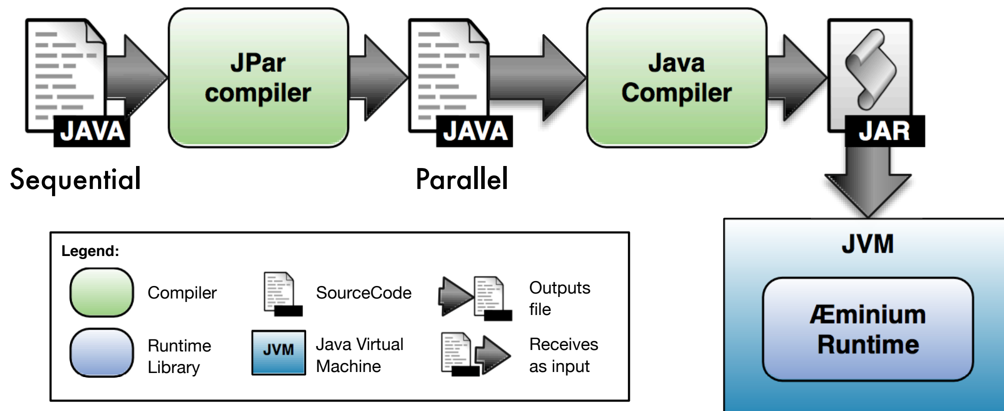

Figure 4.2: Architecture of ÆminiumGPU

The ÆminiumGPU framework was designed for supporting Æminium (Stork et al., 2014) and Java programming languages. Since Æminium compiles to Java, we will address this issue from the Java point of view. Since Java is not a language natively supported by GPUs, Java is translated to OpenCL (Stone et al., 2010) functions during compile-time. Then during execution, the ÆminiumGPU Runtime schedules those functions dynamically. The general architecture can be seen in Figure 4.2.

4.2.1 ÆminiumGPU Compiler

The ÆminiumGPU Compiler is a source-to-source compiler from Java-to-Java, in which the final Java code has some extra OpenCL code. The OpenCL code is based on lambda functions present in the source code. For each lambda in the original code, the compiler creates an OpenCL version. This version is later used to generate a kernel which will execute on the GPU.

The compiler was implemented using Spoon, a Java-to-Java compiler framework (Pawlak et al., 2015). Spoon parses and generates the AST and generates the Java code from the AST. The ÆminiumGPU compiler introduces new phases that produce the OpenCL version of existent lambdas. The compiler looks for methods with a special signature, such as map or reduce. The AST of lambdas passed as arguments are then analyzed and a visitor tries to compile Java code to OpenCL as soon as it descends the AST.

It is important to notice that not all Java code can be translated to OpenCL. The ÆminiumGPU compiler does not support all method calls, non-local variables, for-each-loops, object instantiation and exceptions. It does support a common subset between Java and C99 with some extra features like static accesses, calls to methods and references to fields of the Math object.

4.2.2 ÆminiumGPU Runtime

The ÆminiumGPU Runtime is a Java library responsible for providing Æminium and Java programs with parallel-ready lists that implement the GPU methods, such as `map` and `reduce` methods. Each list can be associated with a GPU, thus supporting several GPUs on the same machine. Whenever a GPU operation is summoned, the following phases occur:

- The compiler-generated OpenCL function is inserted into a predefined template (specific for each operation, such as map or reduce). The result is a kernel function which is compiled to the GPU;
- Input data, if any, is copied from the host memory to the GPU memory;
- The kernel is scheduled to execute on the GPU. The scheduling includes the choice of granularity-specific arguments: *workgroup* and *workitem* dimensions and sizes. These parameter define how many concurrent tasks are scheduled;
- Once execution is finished, output data is copied back to the host device.
- All GPU resources are release and memory freed.

The templates used for Map and Reduce, since we are focusing on these operations for this work, are really straightforward. The map kernel only applies a function to an element of the input array and writes it to the output array. The reduce kernel is a generic version of NVIDIA's implementation (Harris, 2010), allowing for more data-types than the four originally supported.

For these operations in particular, one optimization already implemented is the fusion of maps with maps, and maps with reduces. This optimization is done by considering the Map operation a lazy operation that is only actually performed when the results are needed. This laziness allows for merging together several operations, saving time in unnecessary memory copies and kernel calls. Because of this optimization, the final kernel is only known and compiled at runtime.

All GPU operations have a correspondent CPU version, which executes on top of the Æminium Runtime. Thus, supported operations can occur on either the GPU or the CPU.

4.3 A Machine Learning Approach for GPU-CPU Decision

The proposed approach is based on Machine Learning techniques to automatically decide if a given operation should be executed on either the GPU or CPU. The problem can be described as two-classed because each program execution can be classified as either *Best on GPU* or *Best on CPU*. Supervised learning will be used, since it is important to associate certain features of programs to the two platforms.

Since decisions are hardware dependent (CPU and GPU combination), the prediction model must be trained for each machine. Thus, when installing ÆminiumGPU on a new machine, a training benchmark should be executed to collect training data. However, machines with the same processor and GPU can reuse the same models.

Name	Size	Collected at	Description
OuterAccess	3	Compilation	Global GPU memory read.
InnerAccess	3	Compilation	Local (thread-group) memory read. This area of the memory is faster than the global one.
ConstantAccess	3	Compilation	Constant (read-only) memory read. This memory is faster on some GPU models.
OuterWrite	3	Compilation	Write in global memory.
InnerWrite	3	Compilation	Write in local memory, which is also faster than in global.
BasicOps	3	Compilation	Simplest and fastest instructions. Include arithmetic, logical and binary operators.
TrigFuns	3	Compilation	Trigonometric functions, including *sin*, *cos*, *tan*, *asin*, *acos* and *atan*.
PowFuns	3	Compilation	*pow*, *log* and *sqrt* functions
CmpFuns	3	Compilation	*max* and *min* functions
Branches	3	Compilation	Number of possible branching instructions such as *for*, *if* and *whiles*
DataTo	1	Runtime	Size of input data transferred to the GPU in bytes.
DataFrom	1	Runtime	Size of output data transferred from the GPU in bytes.
ProgType	1	Runtime	One of the following values: Map, Reduce, PartialReduce or MapReduce, which are the different types of operations supported by ÆminiumGPU.

Table 4.1: List of features

The critical aspect for having a good classification is choosing the right features to represent programs. For instance, it is not feasible to consider the full program in ASCII, since the length would be variable and the abstraction level ill-suited for classification techniques. Even a skilled programmer may not be absolutely certain how the code guarantees that performance on one platform will be better. Table 4.1 lists all the features used in the classification process.

```
a(); // Level 1
for (int i=0; i<10; i++) {
  b(); // Level 2
  while (j < 20)
    c(); // Level 3
}
```

Listing 4.2: Examples of Level categorization

Features can be extracted either during compilation or during runtime. This means that a given program will always hold the same values for the first features, while the last three features may be different, depending on the conditions of execution. Features marked with a size of 3 have three values, one for each depth of loop scopes. Listing 4.2 shows an example in which three functions are considered in 3

different loop levels. This distinction is important since operations in inner levels are executed more times than ones in the outer levels.

The choice of some selected features was inspired by other applications of Machine Learning that use source code as input, but unrelated to GPU scheduling (Cavazos and Moss, 2004; Russell et al., 2005; Wang and O'Boyle, 2009). Here, features are selected also based on their importance to the GPU programming model.

Memory accesses were considered a feature as they are one of the main reasons why GPU programs are not as fast as one would expect. As such, there are features for all three main kinds of memories in GPUs (global and slow, local and fast, global read-only and fast). Note that some GPU models may not have one of these, but this feature model is generic for any GPU.

In terms of operations, micro-benchmarks were used to assess their execution cost. For instance, 4 or 5 *plus* operator calls execute much faster than one single *sin* call. As such, OpenCL functions were grouped according to the relative cost they have on execution time.

Besides these features, each benchmark also collected the execution time in both CPU and GPU, and the class each execution belongs to. This is used for training and evaluation.

4.4 Evaluation and Classifier Selection

In this section we will describe the experiments performed for verifying and validating our approach and to select a classifier to use in the implementation.

4.4.1 Dataset

The dataset used for training and testing was comprised of two benchmark suites, one manually written and the other synthetically generated.

The first benchmark suite was written by a programmer representing common operations that could be scheduled to the GPU. The following 8 programs are used:

1. A map operation that adds 1 to each element of the input array;
2. A map operation that applies the *sin* function to each element of the input array;
3. A map operation that applies the *sin* and *cosine* functions to each element of the input array and sums the values;
4. A map operation that calculates the factorial for each element of the input array;
5. A map-reduce operation that calculates the integral from 0 to the size of the array for $f(x) = e^{\sin(x)}$;
6. A map-reduce operation that calculates the minimum value from 0 to the size of the array for $f(x) = 10x^6 + x^5 + 2x^4 + 3x^3 + \frac{2}{5}x^2 + \pi x$;
7. A map-reduce operation that calculates the sum of all natural numbers up to a given value which are divisible by 7;

8. A map-reduce operation that calculates the sum of all elements of the input array which are divisible by 7.

Each one of these programs was executed several times with varying amounts of input data. The size of input data varies from 10 to 10^7 elements, with an exponential increase of 0.1 using base 10.

The second benchmark suite was randomly generated. Each program could be a map, a reduce or a map-reduce operation. Each operation was generated from a random AST with a maximum depth of 30. The grammar used to generate the AST is described below. The grammar reflects the Java language and requires the resulting program to be correctly typed. Some implementation details were omitted for the sake of reading. *random* and *randint* assume randomly generated doubles and integers and *id* assumes a randomly generated identifier.

⟨*program*⟩ ::= ⟨*input*⟩ ⟨*map*⟩
  | ⟨*input*⟩ ⟨*reduce*⟩
  | ⟨*input*⟩ ⟨*map*⟩ ⟨*reduce*⟩

⟨*input*⟩ ::= 'new IntList()'
  | 'new LongList()'
  | 'new FloatList()'
  | 'new DoubleList()'

⟨*map*⟩ ::= 'map(input =>' ⟨*stmt*⟩ ')'

⟨*reduce*⟩ ::= 'reduce(input, other =>' ⟨*stmt*⟩ ')'

⟨*stmt*⟩ ::= ⟨*var*⟩ '=' ⟨*exp*⟩
  | 'var' ⟨*id*⟩ '= 1'
  | 'if (' ⟨*expr*⟩ ') ' ⟨*stmt*⟩ ' else ' ⟨*stmt*⟩ ''
  | ⟨*stmt*⟩ ⟨*stmt*⟩
  | 'for(int '⟨*id*⟩'=0;'⟨*id*⟩'⟨'<*randint*⟩';'⟨*id*⟩'++) ' ⟨*stmt*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩ '%' ⟨*expr*⟩ '== 0'
  | ⟨*expr*⟩ > ⟨*expr*⟩
  | ⟨*expr*⟩ + ⟨*expr*⟩
  | ⟨*expr*⟩ - ⟨*expr*⟩
  | ⟨*expr*⟩ * ⟨*expr*⟩
  | 'Math.min(' ⟨*expr*⟩ ',' ⟨*expr*⟩ ')'
  | 'Math.max(' ⟨*expr*⟩ ',' ⟨*expr*⟩ ')'
  | 'Math.round(' ⟨*expr*⟩ ')'
  | ⟨*randint*⟩
  | ⟨*random*⟩
  | 'Math.cos(' ⟨*expr*⟩ ')'
  | 'Math.sin(' ⟨*expr*⟩ ')'
  | 'Math.tan(' ⟨*expr*⟩ ')'
  | 'Math.log(' ⟨*expr*⟩ ')'
  | 'Math.pow(' ⟨*expr*⟩ ',' ⟨*expr*⟩ ')'
  | 'Math.sqrt(' ⟨*expr*⟩ ')'

This grammar was used to generate 700 independent programs, which were executed with lists with different number of elements, from 10 to 10 million elements with a 10 times increment.

Overall, the full benchmark had 16898 instances of GPU-ready operations, across 8 manually written programs and 700 randomly generated programs.

The dataset was balanced using the SMOTE oversampling technique (Chawla et al., 2002). This was necessary because the original dataset had more programs that executed faster on the CPU. Because there is not a large quantity of data points, oversampling was preferred to undersampling.

Additionally, features were scaled between 0 and 1, necessary for the MLP classifiers.

4.4.2 Experimental Setup

In this evaluation, the CPU was a i7-3520M at 2.90GHz with two cores and four hyperthreads and the host memory size was 8GB. The GPU was a GeForce GT 640M LE, with 384 CUDA cores and 2GB of device memory.

The Operating System was Ubuntu Linux 14.04 with CUDA 7.5 and Java HotSpot 1.8. The results presented here are specific to this particular hardware and software, but the same evaluation can be performed on any GPU-CPU combination.

4.4.3 Feature analysis

The importance of each feature was evaluated using a meta-estimator consisting of 10 decision trees. Results of this evaluation are shown in Table 4.2, omitting features with no relative importance.

The most important feature is the parallel operation, which can be map, reduce or map-reduce. The operation determines the parallelism and synchronization required in the program, as map is embarrassingly parallel, and reduces require synchronization at each step. The data copied to the GPU is also another very important feature, because memory transfers have a high penalty, even if in small amounts. As such, it is important to make sure the computation is long enough to compensate the memory transfer time.

Memory accesses are also important, which is consistent with the GPU memory model. Reading from and writing to the GPU main memory is slower than to register or local memory. Inner memory accesses have no importance in the model, but outer accesses do.

Additionally, basic operations are also important, specially those inside loops, because they occur more frequently.

4.4.4 Classifier Comparison

In order to achieve the best accuracy, it is important to choose an adequate classifier. For this task, several off-the-shelf classifiers from SkLearn (Pedregosa et al., 2011) were evaluated, and some custom classifiers were also developed. The following classifiers were used as baselines:

- **Random** classifier that randomly assigns either class to a particular instance

Feature	Relative importance
Op	0.208
DataTo	0.119
OuterWrite1	0.118
OuterAccess1	0.093
BasicOp2	0.090
BasicOp1	0.079
OuterAccess2	0.071
DataFrom	0.035
MinOp1	0.033
OuterWrite2	0.031
OuterAccess3	0.023
ConstantAccess1	0.017
BasicOp3	0.016
SinOp1	0.016
PowOp1	0.015
OuterWrite3	0.015
MinOp2	0.011
SinOp2	0.005
MinOp3	0.004

Table 4.2: Features rank using a forest of trees

- **Always CPU** that always classifies as *Best on CPU*
- **Always GPU** that always classifies as *Best on GPU*

Additionally, classifiers using different approaches were also considered:

- **NaiveBayes** Classifier (John and Langley, 1995)
- Support Vector Machine (**SVM**) classifier using a radial basis function kernel (Wu et al., 2004)
- Multi-Layer Perceptron (**MLP**) using a Stochastic gradient-based optimizer (Kingma and Ba, 2014)
- **DecisionTree** classifier (Loh, 2011)
- **Random Forest** classifier with 100 estimators (Breiman, 2001)
- **Balanced Random Forest** classifier with 100 estimators, using the absolute different between CPU and GPU execution as instance weights (Chen et al., 2004)

Besides these classifiers, a regression-based approach was evaluated, using additional metrics such as: *CPUTime* and *GPUTime*. The main idea was to use regression techniques to predict values of *CPUTime* and *GPUTime* for each instance and then select the smallest value. However, regressions have shown to have a poor quality with correlation coefficients between 70 and 80%. The final classifier behaved very similarly to the Random classifier. Thus, this approach was not further advanced.

Classifiers were evaluated using a group-sensitive 10-fold cross-validation. The split between training and testing sets at each fold kept instances originated from the same program on different sets, in order not to use other executions of the same program to improve results.

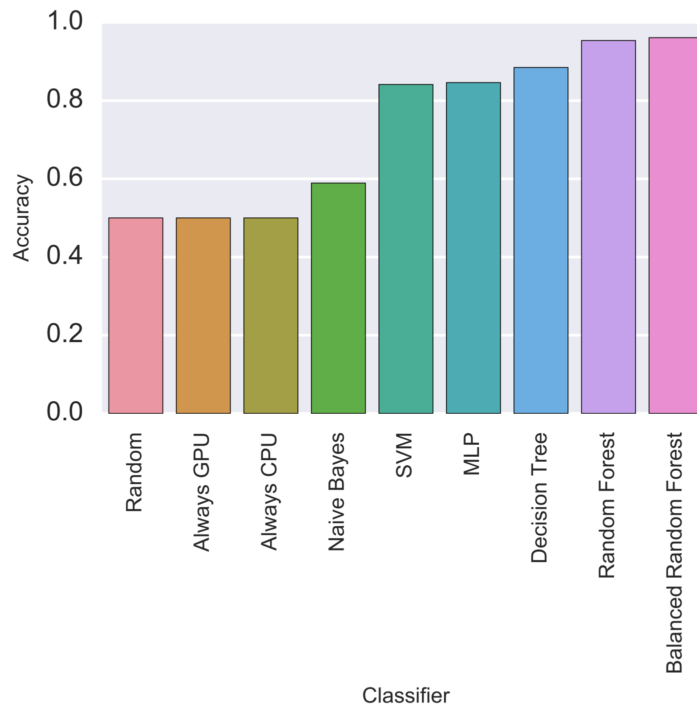

Figure 4.3: Accuracy of different classifiers for GPU-CPU decision.

Figure 4.3 shows the accuracies of the different classifiers over the full benchmark. It is expected that Random, Always GPU and Always GPU all have 0.5 of accuracy, given that the dataset is balanced. The Naïve Bayes classifier improved little over the random results, but all the other classifiers performed well, with accuracies over 80%. Both Random Forest implementations had an accuracy of over 95%, with the balanced version being slightly better.

In this particular problem the distinction between false positives (FP) and false negatives (FN) is not relevant because their impact is different depending on the data size. For programs with a small input array, choosing the GPU has a relative huge impact on the performance. For a very big program, choosing the CPU may also have a strong penalty on performance. This difference can be seen in the example of Figure 4.1.

In order to better represent the impact of taking the wrong decision, a measure of cost was introduced to replace the tradition confusion matrix. Since the CPU

and GPU execution times were recorded, the time lost because of a wrong decision can be obtained. Equation 4.1 defines the cost function applied for each instance.

$$\mathsf{cost}(\mathsf{i}) = \begin{cases} |\mathsf{cpuTime_i} - \mathsf{gpuTime_i}| & \text{if misclassified} \\ \mathsf{0} & \text{if well classified} \end{cases} \quad (4.1)$$

Figure 4.4 shows the misclassification cost of the same classifiers, with a logarithmic scale on the cost. The lowest the cost is, the better the classification is. A perfect classifier would have a cost of 0. The random classifier has an average cost of 24 seconds, which can be considered as a ceiling for this dataset.

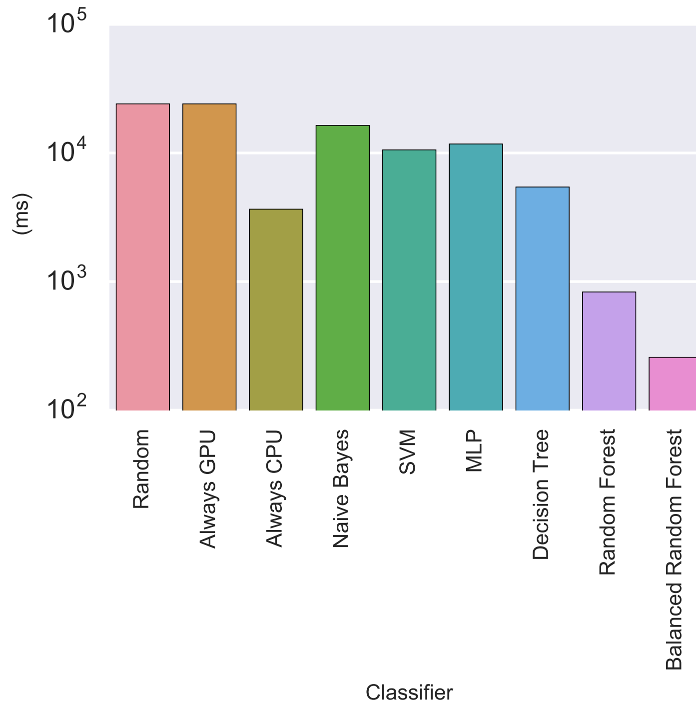

Figure 4.4: Misclassification cost of different classifiers for GPU-CPU decision.

The Always GPU classifier has a higher cost than Always CPU because the cost of misclassification when the GPU is the wrong choice is higher. This typically occurs with lightweight operations over a small amount of data. In this case, the usage of the GPU would imply a high penalty in memory copies. On the other hand, choosing the CPU for an intensive operation over a large array would be slower, but not that slower compared with the first scenario.

Using the misclassification cost it is possible to see the impact of the Balanced Random Forest, compared with the regular Random Forest. By using the possible cost as weights for node decision in the Decision Trees, the decision is skewed to minimize the misclassification cost, without decreasing accuracy (although in this problem it would be acceptable).

4.5 Related Work

Since this work has been published, other Machine-Learning approaches have been applied in related approaches. This section presents these works and compares them

to this one.

Simultaneously to the development of this work, a similar approach using Machine Learning was published by Grewe and O'Boyle (2011), performing all the feature extraction at compilation time and focusing on OpenCL. This approach has the limitation of not covering programs that use dynamic data. In the meanwhile, other works on the same area have arisen, such as Kofler et al. (2013) that compares Artificial Neural Networks with SVMs, and take the same conclusions: ANNs perform better than SVMs at the cost of a longer training time.

Wang et al. (2013b) uses profiling to improve the scheduling of operations between CPU and GPUs, recording data points, and not using static analysis. Shen et al. (2014b) extends the profiling approach to take into consideration irregular programs, like the Integral program. Shen et al. (2014a) predicts the ideal partitioning of data to distribute over GPU and CPUs, using modeling of distributions by performing partial profiling with a subset of the data. These three approaches use previous runs of the same program with different data sizes, which is not usable in automatic parallelization for newly written programs.

Baldini et al. (2014) shows how the usage of ML can be used to predict GPU performance given the CPU execution times. This model is similar to Kerr et al. (2010), in which the performance of several GPUs is predicted using polynomial regression. Our results have shown that binary classification has an higher accuracy than predicting GPU and CPU execution times independently.

These works claim that each GPU should have its own model, with separate training. This aspect is considered in the proposed approach, with micro-benchmarking being machine-specific.

4.6 Conclusions

In this chapter we have presented a Machine-Learning approach to automatically decide whether to execute a program with fine granularity on the GPU or with a coarse granularity on the CPU. This approach defined the feature extraction, which occurs during compilation and execution. These features were ranked to understand their impact on the decision. A dataset was prepared for this problem, including programs written by a programmer, and randomly generated according to a defined grammar.

Potential classifiers were evaluated using accuracy and a new metric: misclassification cost. The best classifier was a Random Forest trained with instance weights equal to the potential misclassification cost, obtaining over 95% average accuracy and the lowest misclassification penalty of all classifiers.

Chapter 5

The No Free Lunch Theorem for Granularity Algorithms

In Chapter 3, the granularity of parallel tasks has been identified as one of the most influential attributes for performance in parallel programs. Granularity control algorithms are used to dynamically adjust the granularity of a program. However, granularity control algorithms perform differently in different parallel programs (Duran et al., 2008b,a). An unsuitable granularity control algorithm can slow down the execution of a program several times, even if both were parallelized using the same model. This Chapter presents new granularity control algorithms that can have benefits in some types of programs. A wide-range evaluation across multiple granularity control algorithms and a heterogenous benchmark suite is conducted, identifying this as a case of the No Free Lunch theorem (Wolpert et al., 1995; Wolpert and Macready, 1997), in which granularity algorithms have the same performance averaged across all classes of problems.

5.1 Introduction

As seen in Chapter 1, task granularity is one of the most important aspects of tuning a parallel program. Figure 5.1 shows the speedup of different cut-off mechanisms used in the programs generated by the JPar compiler presented in the previous chapter. The cost-model approach was not considered because it requires knowledge of the program beforehand, while all other approaches do not and can be used with any parallel program, whether manually or automatically parallelized. It is possible to see that the best cut-off strategy is different for each program. For instance, the best cut-off for BlackScholes is *MaxLevel(12)*, which is unable to provide any speedup on other programs. Other algorithms perform relatively better in some cases, and worse in others.

The main issue with cut-off algorithms is that each class of problems requires a different cut-off strategy. Unfortunately, selecting the right algorithm is traditionally done by a trial-and-error or simulation-based approach, which is cumbersome and time-consuming, forcing developers to test all possible combination of algorithms and parameters for that program with a matching workload. Making the best possible choice is very important, as a wrong decision can influence the execution time of the program by orders of magnitude.

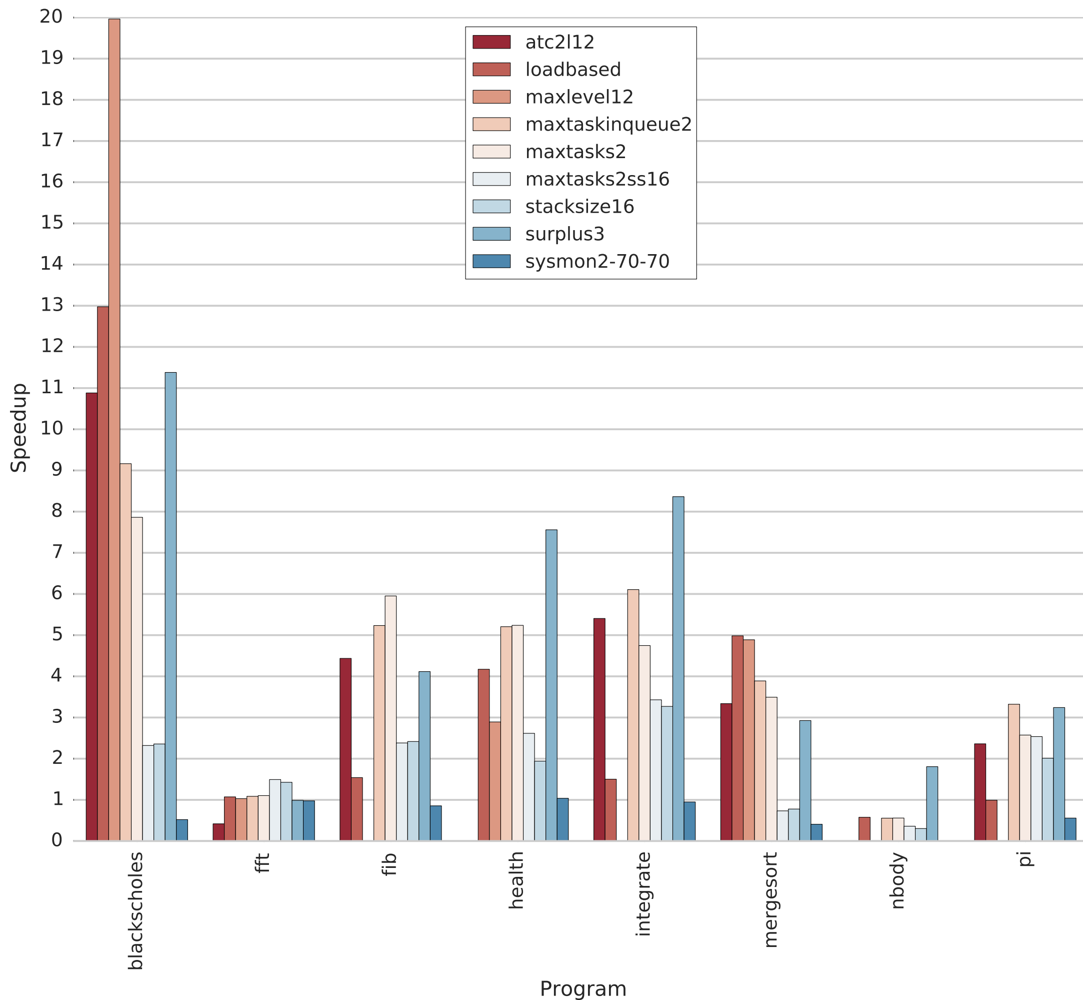

Figure 5.1: Speedup of different cut-off mechanisms used in different programs compiled with the JPar compiler, implementing the approach presented in Chapter 3.

In this chapter, novel cut-off algorithms are presented (Section 5.2). These algorithms, as well as existing algorithms, are evaluated to understand what program characteristics impact the choice of best cut-off algorithm (Section 5.5). Finally, the No Free Lunch Theorem is shown to apply to Cut-off Algorithms (Section 5.6).

5.2 Cut-off Algorithms

A cut-off mechanism is an algorithm that decides whether a task will spawn new tasks for parallel work, or will it execute tasks sequentially. This section briefly presents the cut-off mechanisms discussed in Chapter 2 and describes new proposed cut-off algorithms to improve the performance on a subset of parallel programs.

The following mechanisms have been previously introduced:

LoadBased - A new task is only created if there is at least one idle core (Duran et al., 2008a).

MaxLevel (Maximum task recursion level) - Tasks are only created until a certain depth in the recursion level (Duran et al., 2008b).

MaxTasks (Maximum number of tasks) - Tasks are only created if the overall

number of tasks is below a threshold length.

ATC (Adaptive Tasks Cut-Off) - A task is only created if there are less tasks in the system than a certain threshold and the recursion is below a certain depth (Duran et al., 2008a).

Surplus (Surplus Queued Task Count) - A task is created if the number of queued tasks in the current queue exceeds the number of other tasks by a certain threshold (Lea, 2000).

In order to improve these algorithms, three novel algorithms are proposed:

MaxTasksInQueue (Maximum Queue Size) - We introduce this new approach, which limits the number of tasks in the local queue to a certain threshold. This approach differs from *MaxTasks* in only looking at the local queue, instead of all the queues, avoiding the overhead of accessing information from other threads. If the threshold is one or two tasks higher than the threshold of *MaxTasks*, queues will have extra tasks that can be stolen by other threads.

Stack Size - In recursive divide-and-conquer programs, the recursion limit of the platform imposes heavy limitations on the parallelization of programs. Recursive calls are necessary to inline the execution of tasks inside the same worker thread. As such, the performance of programs decreases when the stack grows beyond a certain size. Having this in mind, we introduced a new approach, StackSize, which counts the number of stack frames produced at a given moment, and only allows the creation of tasks if that count is lower than a predefined threshold.

MaxTasks-SS is a hybrid algorithm between MaxTasks and StackSize. It first avoids task creation if the number of stack frames is higher than the threshold. If not, the creation of tasks is regulated by the MaxTasks mechanism.

5.3 Benchmark Suite

In order to evaluate cut-off algorithms, we use a benchmark suite comprised of different fork-join programs that represent the different types of programs being written for task-based work-stealing runtimes. Table 5.1 shows the list of the 24 programs used, their sources and the input sizes used.

The included programs are examples of divide-and-conquer, pipelined parallelism, DO-ALL loops, DO-ACROSS loops, nested parallelism and partial parallelism in a sequential algorithm. There are balanced and unbalanced programs in the benchmark suite.

Except for Do-all, all programs are real-world examples and some are used in other benchmark suites, because of their heterogeneity. Compared with other evaluations, this is the largest and most heterogeneous set of programs ever used for evaluating cut-off algorithms. The benchmark suite is freely available at `https://github.com/AEminium/AeminiumBenchmarks`.

Based on empirical experiments, some of the characteristics of programs that may influence the performance of granularity control algorithms are identified.

Table 5.1 includes some of these characteristics: Type, Balance, Number of Kernels, Branching Factor and Nesting Factor. A program is always recursive, in the sense that the program will recursively solve two halves of the problem in parallel. We discriminate For-Loop programs because they have slightly more overhead in the Binary or Lazy Binary Splitting (Tzannes et al., 2010) scheduling of the Runtime,

Program	Source	Input size	T.	Ba.	K.	Br.	N.
BFS	PBBS (Shun et al., 2012)	d=26,w=2	Rec.	R	1	2	0
BlackScholes	PARSEC (Bienia, 2011)	10000^2	Loop	R	3	2	2
Convex-Hull	PARSEC (Bienia, 2011)	10000^2	Rec.	R	1	2	0
Do-All		100 million	Loop	R	1	2	1
FFT	Cilk(Frigo et al., 1998)	8388608	Rec.	R	1	2	0
Fibonacci	ForkJoin (Lea, 2000)	n=47	Rec.	S	1	2	0
Fibonacci	ForkJoin (Lea, 2000)	n=49	Rec.	S	1	2	0
Fibonacci	ForkJoin (Lea, 2000)	n=51	Rec.	S	1	2	0
Genetic Knapsack		g=100,p=100	Loop	R	5	2	1
Health	BOTS (Duran et al., 2009)	l=7	Loop	R	1	5	1
Heat	ForkJoin (Lea, 2000)	4096x4096, it=1024	Loop	R	1	2	1
Integrate	ForkJoin (Lea, 2000)	error=10^{-9}	Rec.	S	1	2	0
KDTree	PBBS (Shun et al., 2012)	n=10000000	Rec.	R	3	2	0
LUD	ForkJoin (Lea, 2000)	4096x4096	Rec.	R	3	4	1
Matrix Mult	ForkJoin (Lea, 2000)	p=10000, q=r=1000	Loop	R	1	2	1
MergeSort	ForkJoin (Lea, 2000)	n= 100000000	Rec.	R	1	2	0
MolDyn	JGrande (Smith et al., 2001)	it=1 size=40	Loop	R	3	2	1
MolDyn	JGrande (Smith et al., 2001)	it=5 size=30	Loop	R	3	2	1
MonteCarlo	JGrande (Smith et al., 2001)	10000x60000	Rec.	R	1	2	1
N-Body	PBBS (Shun et al., 2012)	n=50000, it=3	Loop	I	1	2	2
N-U Knapsack		items=30, corr=3	Rec.	S	1	2	0
NeuralNet		it=500000	Rec.	R	1	2	0
N-Queens	Cilk (Frigo et al., 1998)	n=8..15	Loop	I	1	2	N
N-Queens	Cilk (Frigo et al., 1998)	16	Loop	I	1	2	N
Pi		n=100.000.000	Loop	R	1	2	2
Quicksort	ForkJoin (Lea, 2000)	n=10000000	Rec.	R	1	2	0
RayTracer	JGrande (Smith et al., 2001)	n=2000	Loop	R	1	2	1

Table 5.1: Description of the programs used in the benchmark.

T. stands for the type of parallelization, Recursive or Loop-based. Ba. stands for Balance, either Regular (R), Irregular (I) or Skewed(S). K stands for Number of Kernels. Br. stands for Branching factor and N for Nesting level.

but do not have merge workload. Programs can be balanced, in which the two halves of the program have approximate amount of work to perform, or irregular, in which a half has more work than the other. The Number of Kernels refers to the amount of different parallel operations that execute during the program. The Branching Factor is the amount of parallel subtasks created inside each tasks, which in most cases is 2. Recursively splitting into 2 tasks is better for unbalanced loads and also distributes workloads across different cores faster. Finally, the nesting level is the amount of nested parallel for-loops. In the case of N-Queens, this value depends on the granularity of tasks.

5.4 Experimental Setup

Name	Processor	CPU Cores	Threads	RAM
server32	Intel Xeon E5-2650 0 @ 2.00GHz	16 cores	32 threads	32GB
server24	Intel Xeon X5660 @ 2.80GHz	12 cores	24 threads	24GB

Table 5.2: Details of the hardware used in the experiments.

Programs were implemented on top of the Æminium Runtime (Stork et al., 2014).

Two machines (Table 5.2) were used in order to generalize results to more than one machine, both running Ubuntu 14.04 and Java HotSpot(TM) 64-Bit Server VM with Java 1.8.

To collect values, a practical statically rigorous methodology (Georges et al., 2007) was applied. For each combination of program and cut-off, we obtained the mean and the 95% confidence interval for the execution time distribution in steady state. The distribution was sampled from up to 30 executions, stopping when the Coefficient of Variance was below 5%. Each program had a timeout of 1 hour. All programs were executed in the same conditions, changing only the cut-off algorithm. There was no other load on the machine besides the experiment and the operating system.

5.5 Comparing Cut-off Algorithms

In sync with findings from prior works (Duran et al., 2008b,a), this section corroborates that no cut-off approach performed better than the others for all programs. Here, the differences in performance from the algorithms are addressed. Since the time distribution of the algorithms is not normal, swarmplots will be used. For parametrized cut-off approaches, we have used the parameters that achieved the best global time, in a preliminary evaluation.

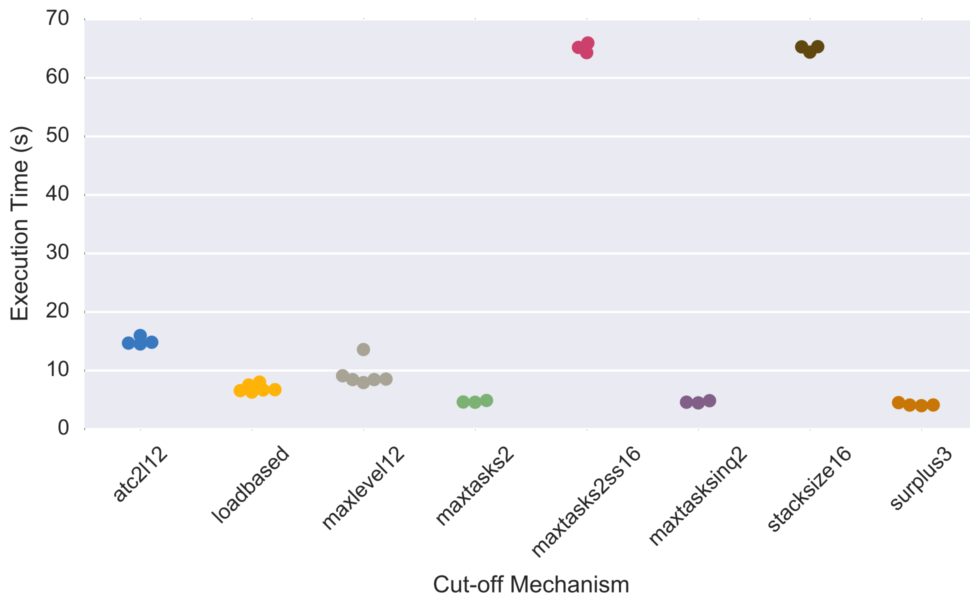

Figure 5.2: Swarm plot of different cut-off approaches for the Do-all program on the *server24* machine.

Do-all is made of parallel for-cycles with several iterations doing only one operation. Figure 5.2 shows the performance of different cut-off mechanisms in the *server24* machine. MaxTasks, MaxTasksInQueue and Surplus were the most efficient strategies and they are all based on having enough work on each queue for other to steal. LoadBased has a similar approach, but does not have extra work in queues. In this case, allowing more threads to steal work results in a faster work distribution across the CPU cores. Recursion-depth approaches like MaxLevel and

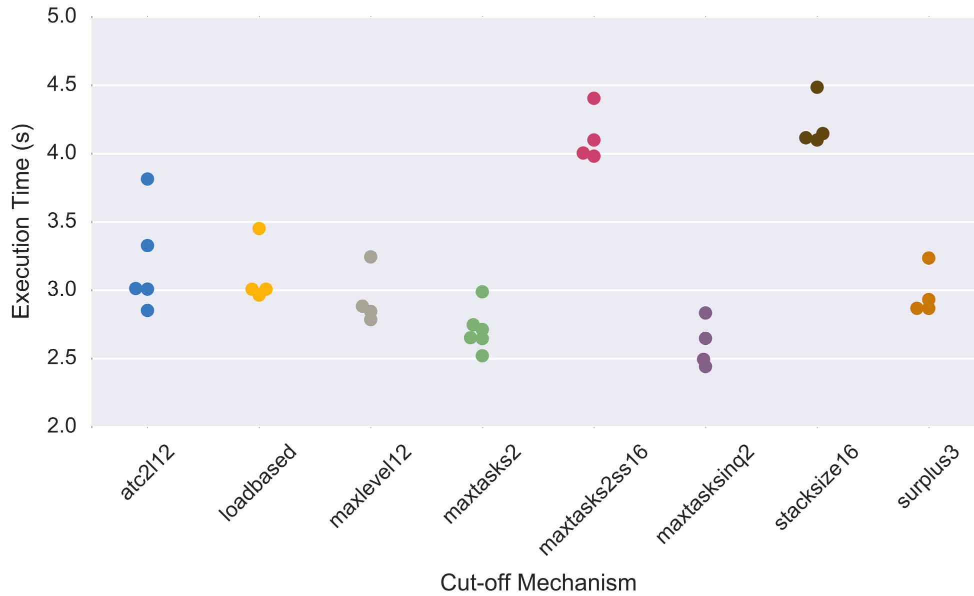

Figure 5.3: Swarm plot of different cut-off approaches for the Matrix Multiplication program on the *server32* machine.

ATC are slower because, in this case, the depth considered was too deep and it created too many tasks. In this case a smaller depth, such as 6 would result in less tasks created, and less overhead, but in other programs it would result in worse performances. Stack-size approaches create too many tasks as well in this case. Figure 5.3 shows the same behavior in the Matrix Multiplication program, which also has lightweight tasks in a 2 dimensional loop cycle.

Figure 5.4: Swarm plot of different cut-off approaches for the Fibonacci program on the *server24* machine.

Figure 5.4 shows the Fibonacci program with different cut-offs. Fibonacci is a highly irregular program that generates a skewed parallelization tree, with a ex-

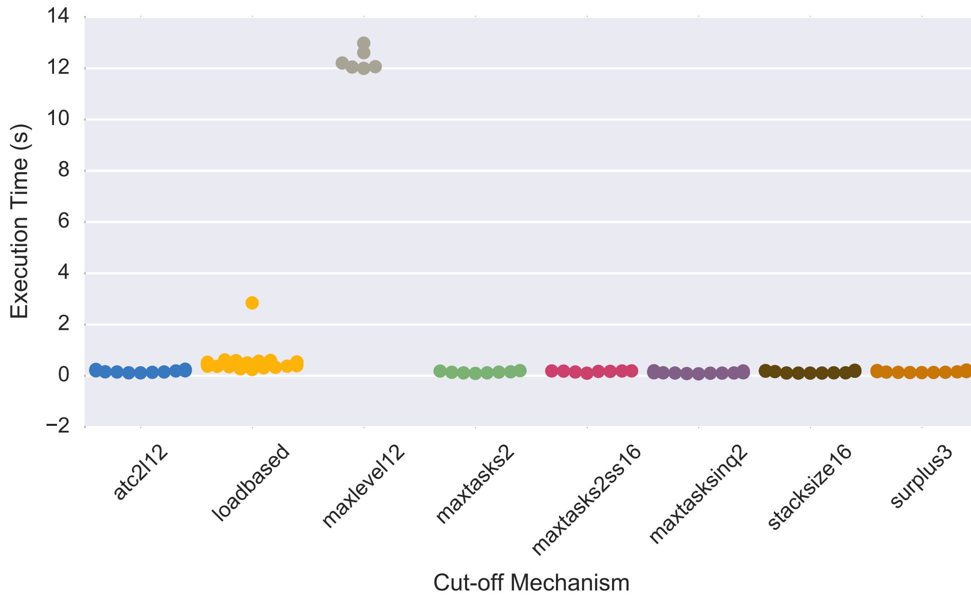

Figure 5.5: Swarm plot of different cut-off approaches for the Integrate program on the *server24* machine.

tremely lightweight computation. In this case all approaches handle the program reasonably well, but MaxLevel is not able to finish the program within the defined timeout. Figure 5.5 shows Integrate, another highly irregular program, in which cut-off programs show the same relative performance, with MaxLevel being much slower than its counterparts.

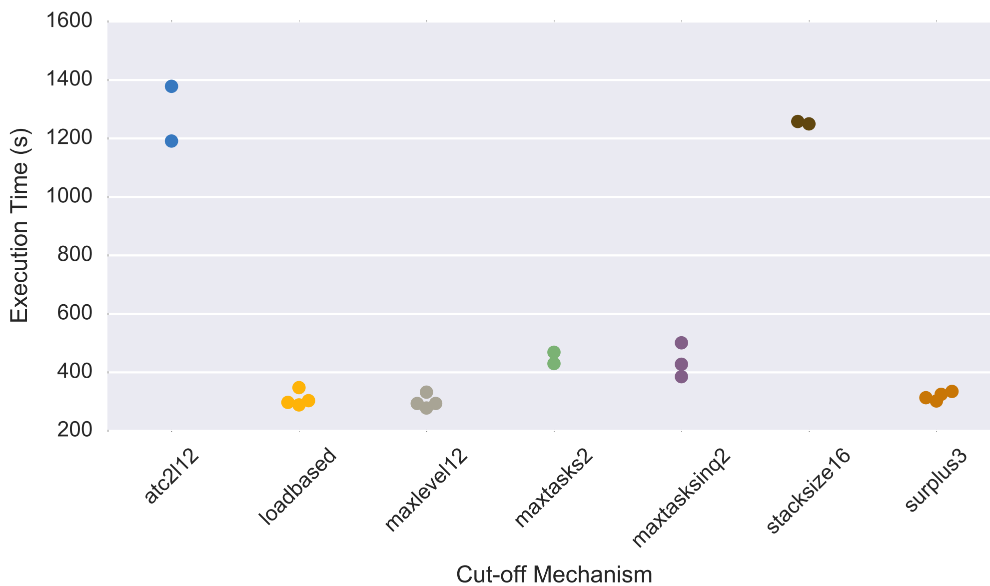

Figure 5.6: Swarm plot of different cut-off approaches for the N-Queens program on the *server24* machine.

In the N-Queens program, in Figure 5.6, Loadbased, MaxLevel and Surplus are the fastest algorithms. This program has a high branching factor and a high penalty for over-creating tasks, because it needs to allocate memory for each parallelization.

MaxLevel avoids going too deep in the recursion tree, but created enough parallelism for the program to have a speedup. LoadBased also prevents creating extra tasks and performs similarly to MaxLevel.

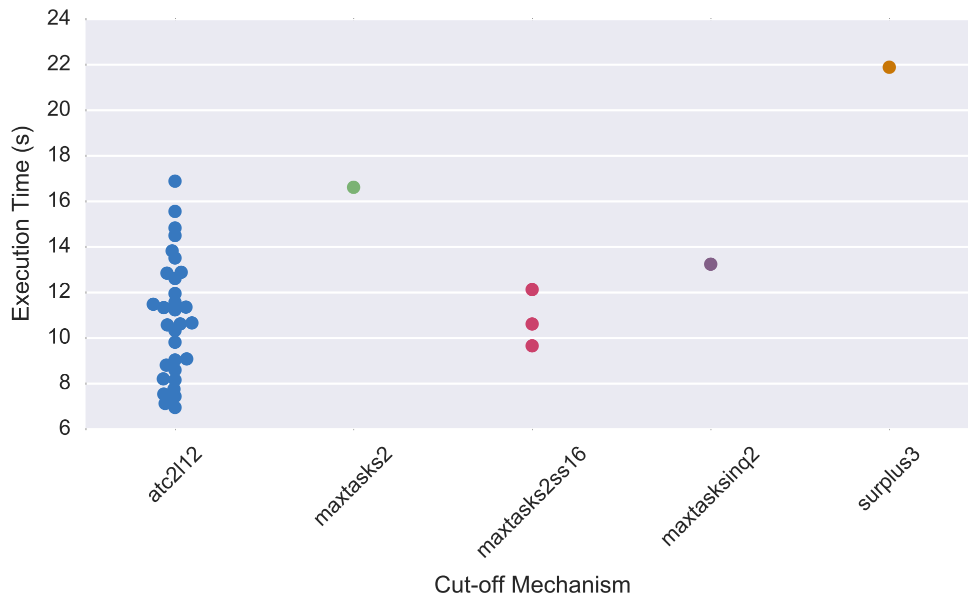

Figure 5.7: Swarm plot of different cut-off approaches for the FFT program on the *server24* machine.

Figure 5.7 shows the cut-off performance in the FFT program. Only ATC and MaxTasksWithStackSize have finished the program 3 times within the timeout. FFT is a program that allocates a large amount of memory in its divide-and-conquer process. The allocation of tasks on top of the baseline allocation of the sequential program penalizes the creation of a large number of tasks. The two best approaches have two mechanisms to limit the creation of tasks, one limiting the queue size, and another preventing it from going too deep in the recursion level. The difference between the two is that ATC limits using the programs recursion and MaxTasksWithStackSize uses the internal recursion of the work-stealing runtime.

In Figure 5.8, we can see the opposite behavior in which ATC and MaxTasksWithStackSize are the worst approaches. One reason for this is that these hybrid approaches use two mechanisms to improve their worst-case programs, but introduce overhead in cases where the individual algorithms are ideal.

Figure 5.9 shows the same plot for the Neural Network program. In this program, creating tasks has a relatively large overhead compared to the program and only StackSize approaches have been able to complete the program within the timeout, in a relatively small time. This is one example that justifies the introduction of stack-size approaches, in which the workload of tasks is very light and there merging recursive task results is expensive. This behavior is the same as in Fibonacci with a very large input. KD-Tree is another program where Stack-based approaches are also advantageous, but not by a larger difference, which can be seen in Figure 5.10.

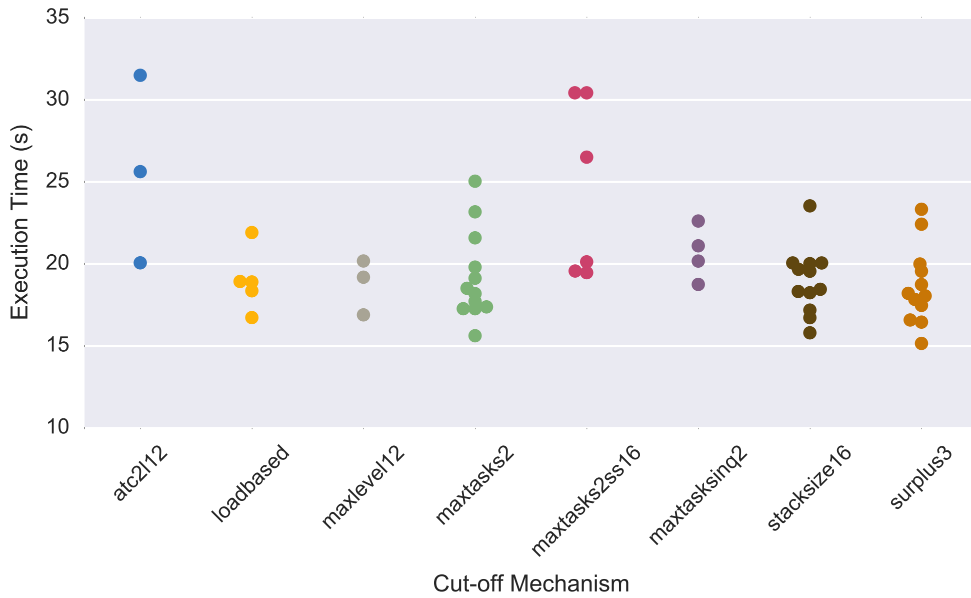

Figure 5.8: Swarm plot of different cut-off approaches for the Raytracer program on the *server24* machine.

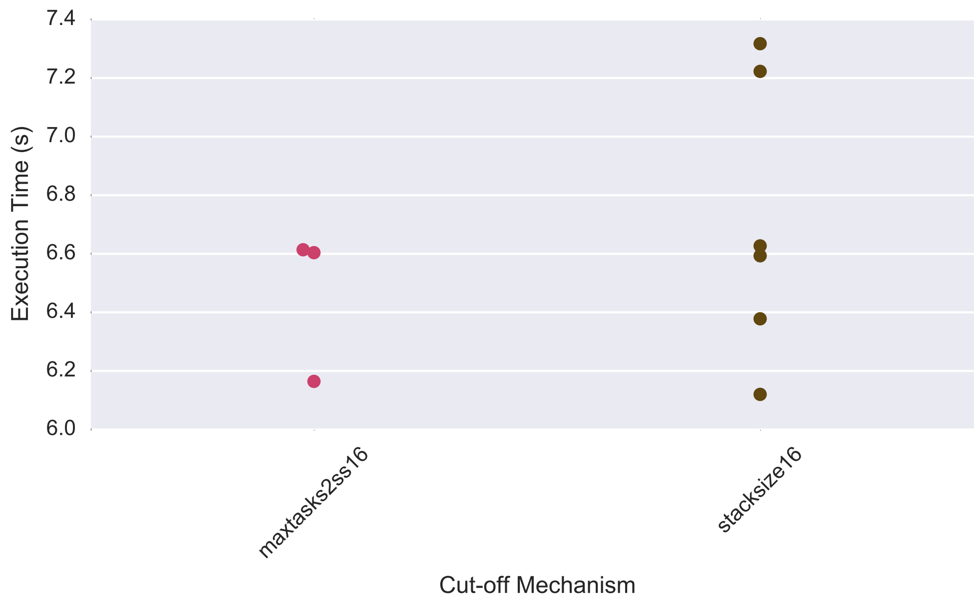

Figure 5.9: Swarm plot of different cut-off approaches for the Neural Network training program on the *server32* machine.

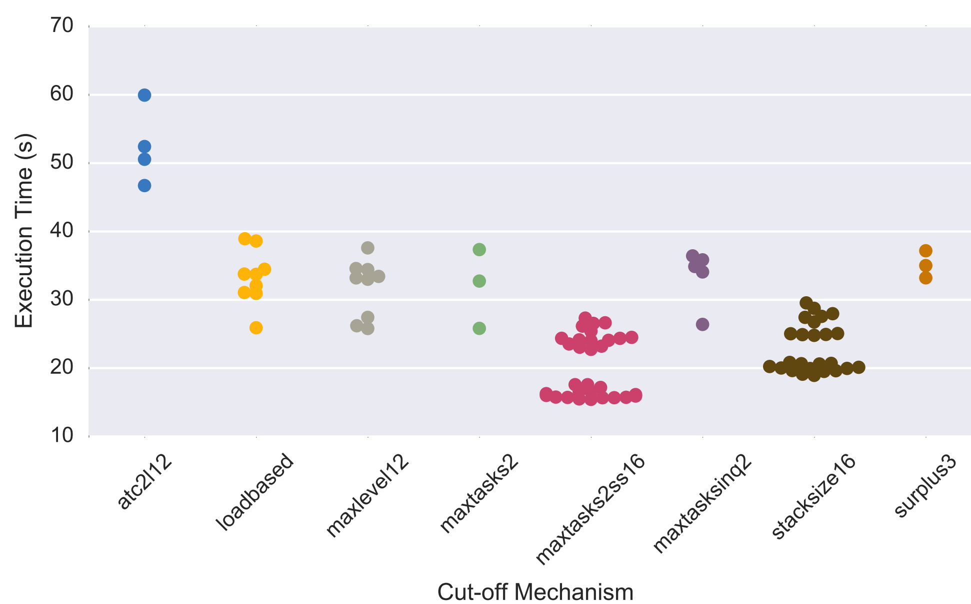

Figure 5.10: Swarm plot of different cut-off approaches for the KDTree training program on the *server32* machine.

5.6 Verifying the No Free Lunch theorem

In order to assert whether the No Free Lunch Theorem applies, we will rank each algorithm according to the relative performance attained in each program. Since we cannot assert in all cases that an approach is better than the other, our ranking reflects three metrics:

- the mean;
- the lower bound of the confidence interval;
- the upper bound of the confidence interval.

Although using the mean for comparison can be a fair assumption, it does not represent the distribution of the execution time. Work-stealing programs can have a large standard deviation in execution time, and comparing the mean represents comparing average runs. By choosing one option over the other, the developer is risking having an execution that is several times the mean. In order to account for that risk, we use the upper bound of the confidence interval for the mean. This metric is more conservative, in which we compare worst-cases and are aware of the deviation in execution times.

Tables 5.3 and 5.4 show the mean ranks, considering the three metrics, on both machines for all cut-off algorithms over the full benchmark. For each program, each cut-off was ranked from best (1) to worst (8), and all approaches that did not finish within the timeout interval were classified as 8.

Considering the ranks using the upper bound, values are limited by 4 and 6, which is the global average rank, considering that there are several programs exceeding the timeout interval. There is no program that performs outstandingly from the others. Thus, we can empirically conclude that the No Free Lunch Theorem applies to Cut-off Algorithm selection for Fork-Join programs. The same conclusion can also be taken if using the mean instead of the upper bound.

	lower bound	mean	upper bound
cut-off algorithm			
atc2l12	5.167	5.583	5.208
loadbased	4.708	5.708	5.833
maxlevel12	4.625	4.250	4.208
maxtasks2	4.417	3.833	4.083
maxtasks2ss16	4.833	4.583	4.333
maxtasksinq2	4.333	3.917	4.417
stacksize16	4.250	4.583	4.708
surplus3	4.750	4.167	4.292

Table 5.3: Mean ranking of cut-off algorithms on the full benchmark on *server32*

Another aspect to evaluate is how the cut-off mechanism scales with different workloads of the same problem. Figures 5.4, 5.11 and 5.12 show the performance of the Fibonacci program for different input parameters in the *server24* machine.

	lower bound	mean	upper bound
cut-off algorithm			
atc2l12	4.808	5.923	5.615
loadbased	4.808	4.962	5.308
maxlevel12	4.577	4.231	5.000
maxtasks2	4.038	4.154	4.077
maxtasks2ss16	4.654	4.154	4.154
maxtasksinq2	4.462	4.000	4.269
stacksize16	5.385	5.038	4.577
surplus3	4.500	4.269	4.231

Table 5.4: Mean ranking of cut-off algorithms on the full benchmark on *server24*

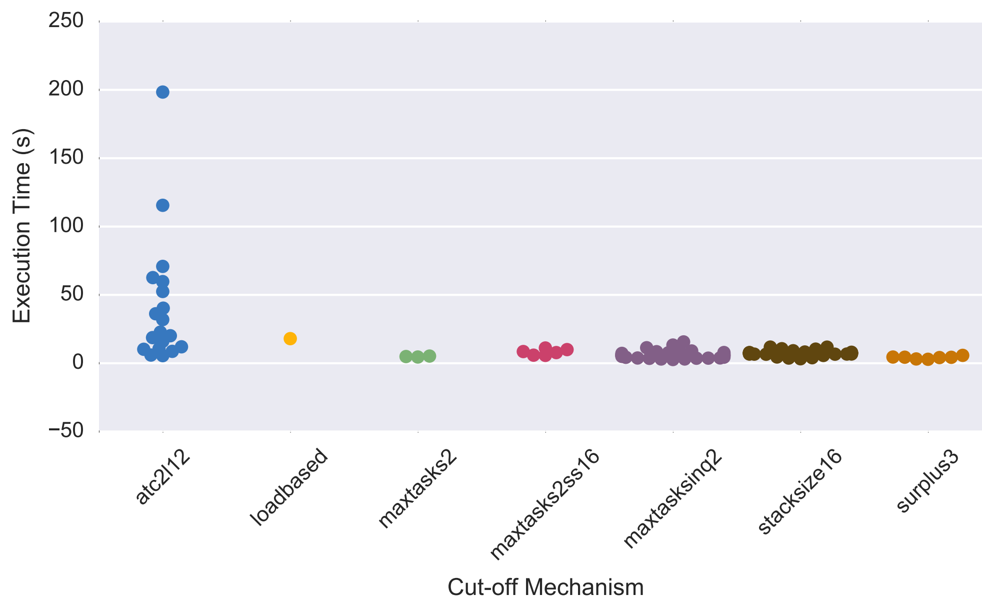

Figure 5.11: Cut-off performance in the Fibonacci program with 49 as input on *server24*

Independently of the time values, the relative performance of cut-offs is the same. The same behavior applies to N-Queens and MolDyn. From this data, it is possible to infer that for large input sizes, the cut-off performance is similar.

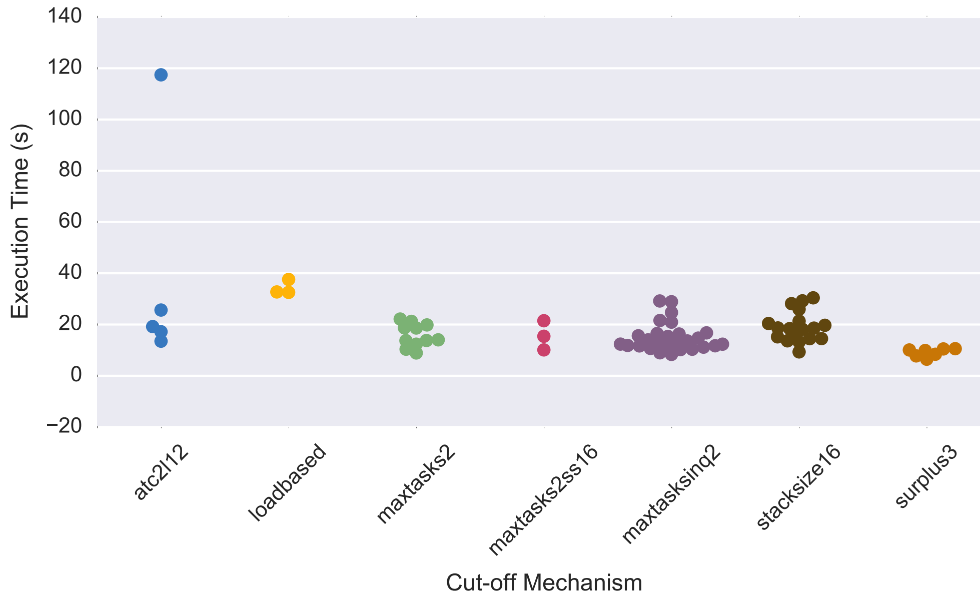

Figure 5.12: Cut-off performance in the Fibonacci program with 51 as input on *server24*

5.7 Conclusions

In this chapter, three novel cut-off algorithms were introduced: MaxTasksInQueue, Stack Size and Max Tasks with Stack Size. These three algorithms have shown to have better performance than existing algorithms on different classes of problems. An empirical evaluation of the performance of different cut-off algorithms was conducted, identifying some characteristics of programs that influence the choice of cut-off algorithms. Finally, the No Free Lunch Theorem has been shown to apply to Cut-off Algorithms, by empirically evaluating the benchmark suite and showing that the mean ranking of algorithms was very similar across programs.

Chapter 6

Energy Efficiency of Granularity Control Algorithms

Recently, there is a concern about reducing the energy consumption of data centers and clusters for economical and environmental reasons. Furthermore, energy consumption on mobile devices is also important to improve battery life. This work addresses the performance-energy trade-off on shared-memory multicore devices in parallel programs. In particular, the energy impact of granularity algorithms is accessed. The aim is to give programmers the knowledge they need to understand how to maximize performance of parallel programs while minimizing energy spending, when selecting granularity control algorithms.

6.1 Introduction

In nowadays multicore platforms, to improve the performance of computationally intensive programs, they have to be designed to execute in parallel. Currently, all types of computers, from smartphones to supercomputers, have multiple CPU cores available. In both ends of the spectrum, lowering energy consumption has become an important goal. On smartphones, tablets and other mobile devices, good performance is important to improve user experience, but battery longevity is also crucial. On clusters and datacenters, energy consumption has also been an important driver for supercomputer design, both for economical and environmental reasons.

On the software side, there are several attributes that influence both speed and energy. This work focuses on shared-memory multicore processors. Typically, the longer a program is running, more energy is being spent on that computation. However, the energy spent does not only depend on the time a program takes to execute, but also on the way CPU and memory are used.

One of the aspects of writing and optimizing parallel programs is granularity control. Most of the times, there is more parallelism in the program than hardware threads. In that case, work must be grouped together to create an ideal match between tasks and hardware threads. A task is a representation of individual work that can be executed asynchronously on any given core. If the number of tasks

is smaller than the number of available hardware threads, some cores can become idle but keep consuming energy. If the number of tasks is larger than the number of hardware threads, time and energy will be spent in non-profit scheduling operations. This work considers tasks instead of Operating System (OS) threads because using system calls would introduce an undesirable overhead. The proposed model uses a one-to-one matching between software OS-level and hardware threads, similar to how green threads work but without preemption.

This Chapter details the methodology used to understand the impact of granularity control on the energy efficiency will be presented (Section 6.2). This methodology will be applied using both synthetic and real-world benchmarks to obtain results (Section 8.3), from which conclusions will be drawn (Section 6.4).

6.2 Methodology

In order to evaluate time and energy consumption, resulting from different cut-off options, a set of relevant programs of multiple benchmark suites has been selected for execution. But, since these programs are very diverse in their nature, it is not possible to draw conclusions for each individual characteristic they exhibit. As such, a synthetic program that can mimic behaviors similar to those of recursive parallel programs has been developed. By increasing and decreasing parameter values, it is possible to see how one aspect influences the performance and power drain of programs.

6.2.1 Synthetic Benchmark

The synthetic benchmark is made of a single program that can be configured to have different behaviors. The main design goal of this program is to be representative of any parallel program, and to be able to intensify the effect of each identified characteristic.

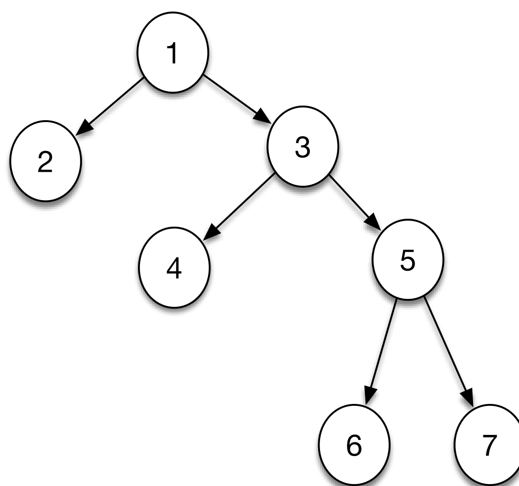

Figure 6.1: Example of a task tree, generated by a possible program

Different behaviors can be achieved by combining four characteristics: workload, memory allocation, leaf workload and branching, each multiplied by factors such

as depth or side of the tree. The computational tree will be considered to define those aspects. Figure 6.1 shows an example of a possible task tree, generated by a program that divides work in two recursively, but only continues recursion on one of the tasks. Tasks identified with 2, 4, 6 and 7 are considered leafs, as they do not spawn any more tasks.

One of the aspects that characterizes a parallel program is the amount of work it performs. The SHA1 hash function is used to simulate workloads in different points. Two possible locations for work are considered: there can be work at every task, before spawning new tasks, or just at the leaf tasks. The amount of work may be static across the program, or it may depend on the depth (the deeper the task, the more work it performs) or it may even depend on the side of the tree, as seen in the Figure 6.1. In this case, the number of previous spawns is used as a factor when generating the unbalanced workload.

Other aspect is the allocation of memory that is performed before spawning tasks. This represents the case of some programs that need to allocate extra memory to perform the work division for parallel calls, a step that would not be necessary for the sequential version.

Finally, the last aspect is branching. A task can have zero or more child tasks. The number of children is typically 2, but depending on the problem it might be higher, or be dynamic. As such, branching is modeled as either static, depth-dependent or side-dependent.

By representing aspects and their factors using variables, it is possible to generate several programs, ranging from regular and balanced to irregular and unbalanced programs. The 13 variables that define a synthetic program are: Maximum Depth, Allocation before spawning, Task load, Task load static factor, Task load side factor, Task load depth factor, Leaf load, Leaf load static factor, Leaf load side factor, Branching, Branching static factor, Branching side factor, and Branching depth factor.

6.2.2 Real-world Benchmark

A real-world benchmark is used to ensure that the conclusions drawn from using the synthetic benchmark would hold on real-world programs. This real-world benchmark is made of several programs, used in different areas. Most of these programs are present in other benchmarks used for evaluation of parallel programs. The same 24 program benchmark presented in Subsection 5.3 and Table 5.1 are considered.

6.2.3 Experimental Setup

All programs were implemented on top of the Æminium Runtime (Stork et al., 2014). Programs were evaluated on a Intel Xeon E5-2650 at 2GHz with 16 cores, 32 hyper-threads and 32GB of RAM. The machine ran Ubuntu 14.04 with Java Hotspot 64-bit Server 1.8. This processor features Intel Turbo Boost, where an idle processor can slow down to 1.2GHz, thus having an impact on performance.

To collect values, each program executes between 3 and 30 times, until the coefficient of variance was below 5%. Each program had a timeout of 1 hour. All programs were executed in the same conditions, changing only the cut-off algorithm. There was no other load on the machine besides the experiment and the operating

system. When not detailed, the mean is used to represent the distribution.

Power usage was collected using the SandyBridge RAPL performance counters (David et al., 2010). Power and time were measured at the same time, and sampling periods were smaller than the duration of the programs.

6.3 Results

In this section the results of this study are presented. First, the correlation between time and energy consumed by a program is presented. To understand behavior of granularity algorithms, synthetic benchmarks are used, firstly to represent balanced programs, and then unbalanced ones. Finally, a real world benchmark is used to validate our conclusions and choose the best cut-off algorithm overall.

6.3.1 Time versus Energy

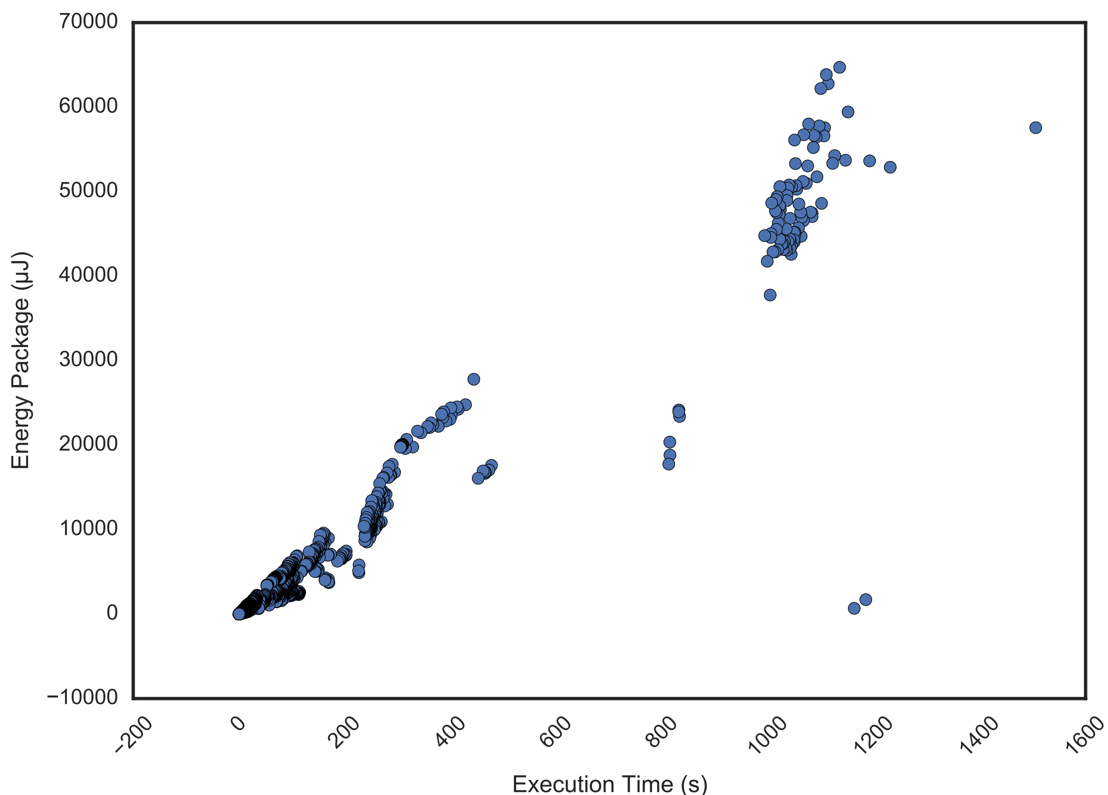

Figure 6.2: Distribution between duration of the program and its energy usage.

Figure 6.2 shows the execution time and energy consumption for all programs. Although there is a tendency of longer programs consuming more energy, there are many cases in which this correlation is not direct. Thus, it is important to identify in which cases a better performance can be obtained with the same or lower energy footprint.

6.3.2 Balanced programs

Using the synthetic program, each variable was varied at a time, keeping all others constant. Firstly, different amounts of work in each task were tested, designated as *before*. Two metrics were used: time and energy relative performance. The relative performance is a ratio between the mean measurements of the best cut-off and of the given cut-off, either in time or energy. A relative performance closer to 1 is better.

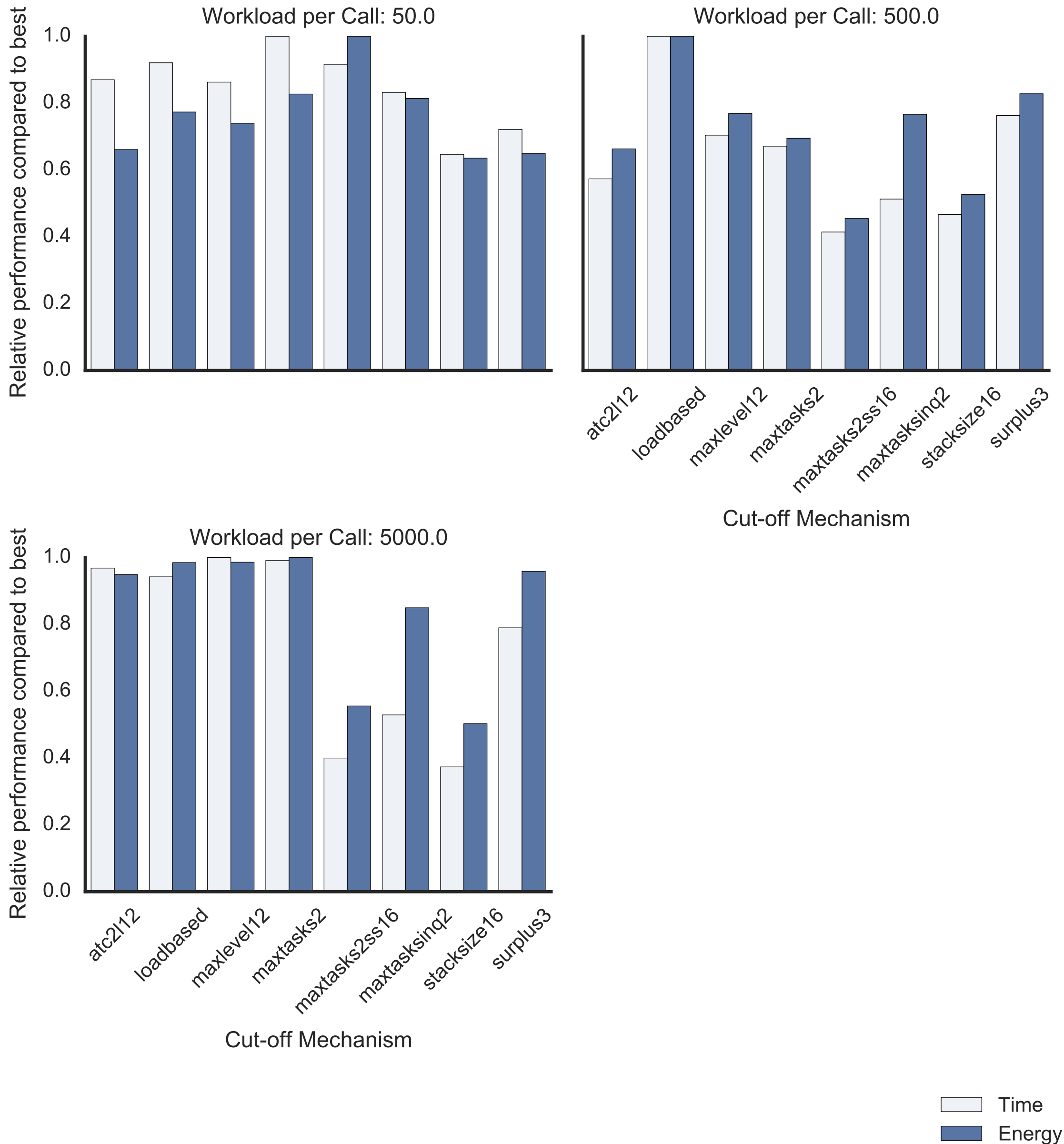

Figure 6.3: Time and energy relative performance for different task workloads. Higher is better.

Figure 6.3 shows relative performance between each cut-off approach and the best in the group, over three different workloads per task. The program has a maximum-cutoff of 10 and static branching of 2. In light workloads, several algorithms have good performance in both time and energy. With a workload per call of 50, it is possible to see that the best program in terms of execution time is not the same

as the program with the lowest consumption. Overall, MaxTasksInQueue is the algorithm with the lowest overhead, thus having a good energy and time efficiency. Max-tasks allows for more tasks to be created, resulting in a faster program, with a lower energy efficiency. With heavier workloads the best algorithms are MaxLevel, ATC and LoadBased. Since this is a regular uniform program, most depth-based algorithms can generate a good granularity. The LoadBased algorithm also performs well in both metrics with balanced programs because cores are always occupied after the initial distribution.

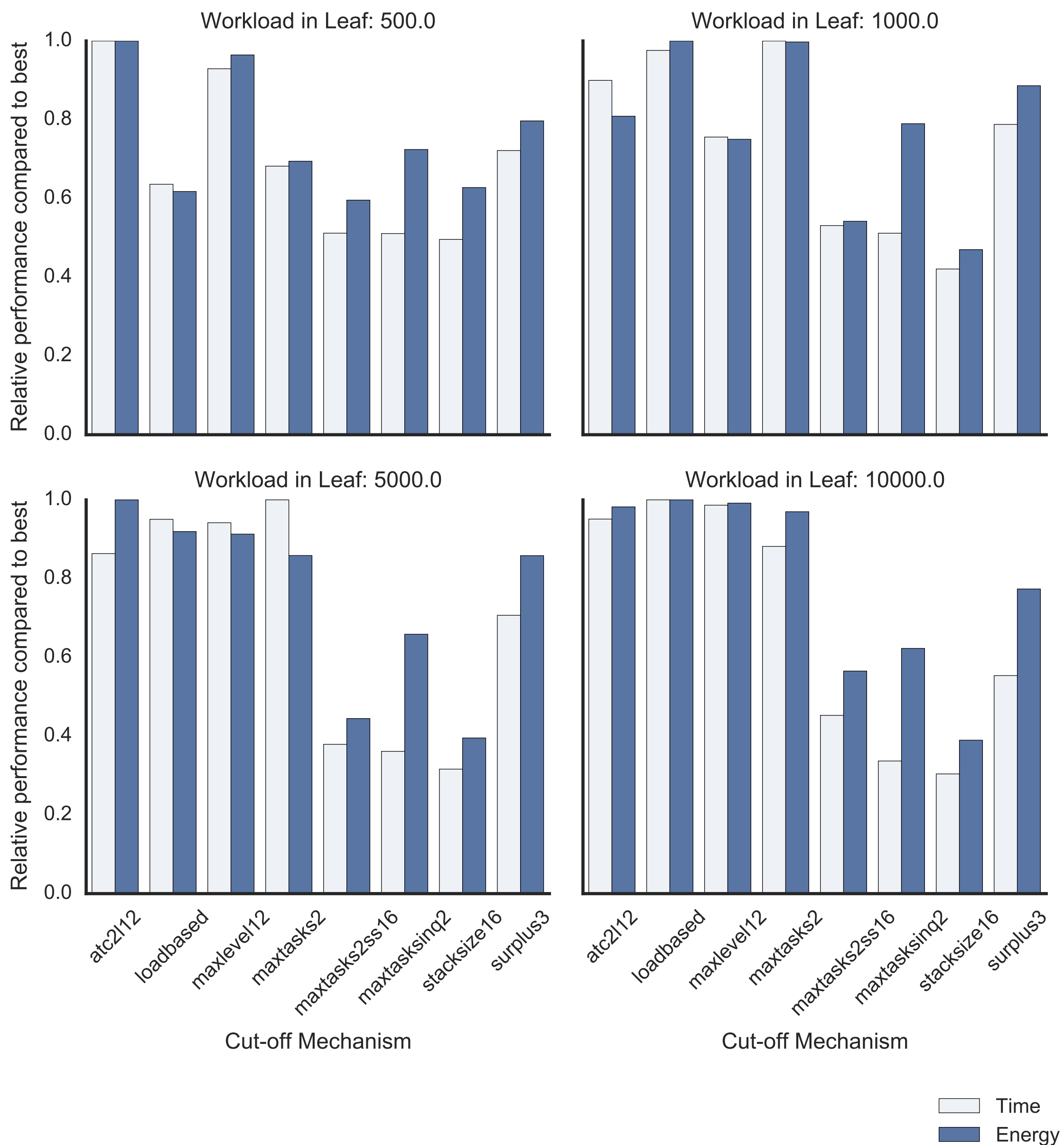

Figure 6.4: Time and energy relative performance for different leaf tasks workloads. Higher is better.

Figure 6.4 shows the variation of the amount of work in leaf tasks, up to 100 thousand iterations at leafs. The main difference to the previous version is that the computation tree is generated first, and work is only done at the leaf level. With a smaller workload, MaxLevel and ATC perform the best in both metrics. With

a medium workload, dynamic approaches such as LoadBased and MaxTasks are preferable. With higher workloads, any of the four previous approaches perform similarly. Doing work at the leaf level is not different from doing it at every task, in terms of the energy effiency. The same rationale can be used to justify the observed results.

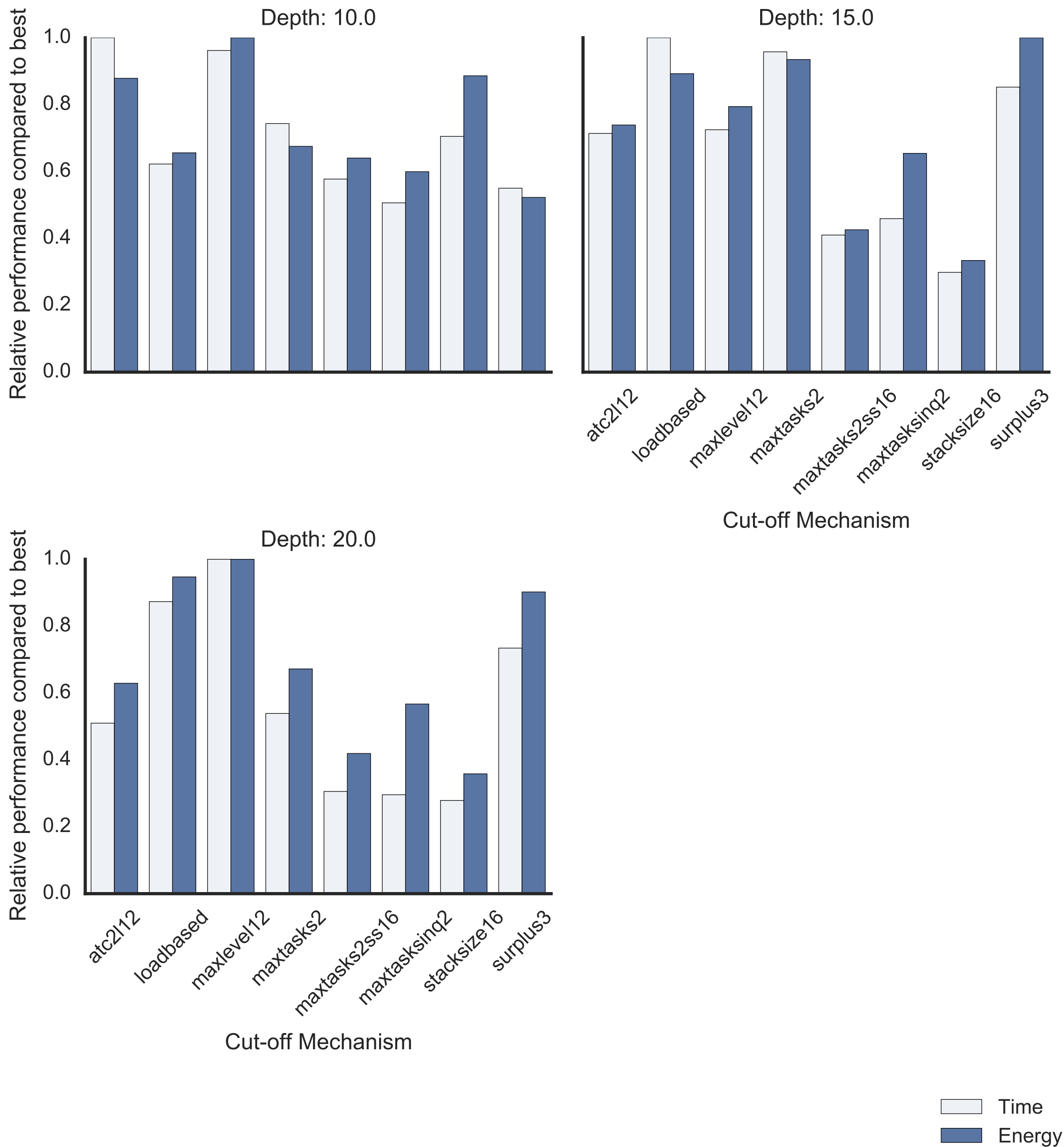

Figure 6.5: Time and energy relative performance for different depths with leaf workloads. Higher is better.

Figure 6.5 shows the variation of the amount of work with different depths. The workload also occurs on leaf tasks. Similarly to what was seen when varying the workload, with light and heavy workloads, MaxLevel is the best cut-off approach. With medium workloads, dynamic approaches such as LoadBased, MaxTasks and Surplus have a better performance. The reason for this is that in smaller workloads the overhead of using a dynamic method represents a large part of the computation. In heavy workloads, the depth cut-off is enough because tasks are sufficiently heavy

to mitigate the overhead of scheduling. However, medium workloads can be so small that the maximum depth level is not low enough to start grouping tasks together (A lower level threshold should produce better results by aggregating more). In this case, the MaxTasks cut-off adapts to this case and yields better results, just like LoadBased.

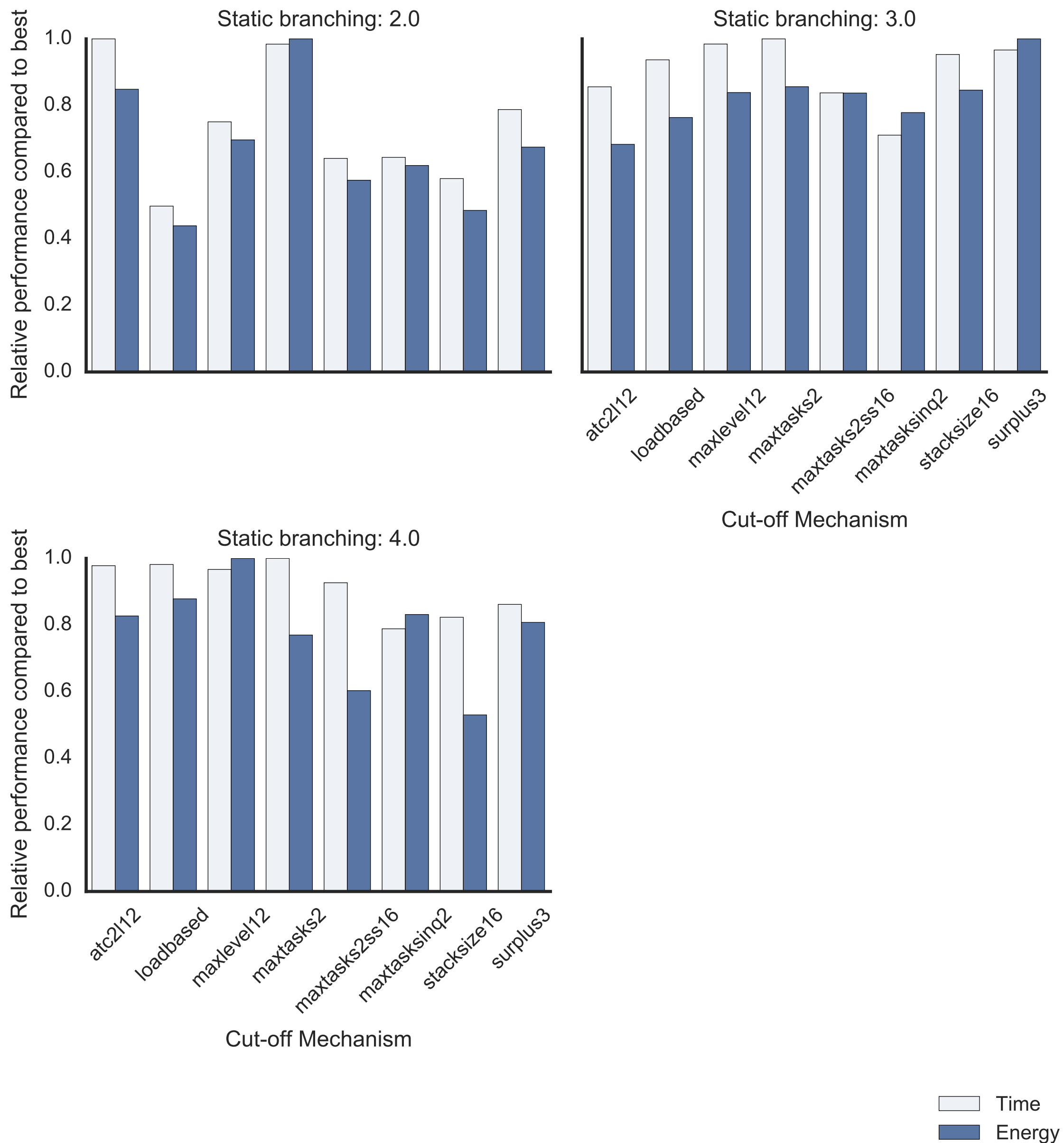

Figure 6.6: Time and energy relative performance for different branching factors with leaf workloads. Higher is better.

Figure 6.6 shows the impact of the branching factor on a balanced program. With a small branching value, MaxTasks and ATC have the best performance, with MaxTasks being the best combination for the two metrics. As the branching factor is increased, the difference between those two and LoadBased and MaxLevel decreases, given their low overhead when there is an overall large workload and relatively low stealing. This difference is most noticeable in energy.

6.3.3 Unbalanced programs

In unbalanced programs, different executions at the same recursion level generate a different number of recursive calls. The result is that the program will have more work, and possible tasks, in one side of the computation tree. The recursive Fibonacci program is an example of a very unbalanced program.

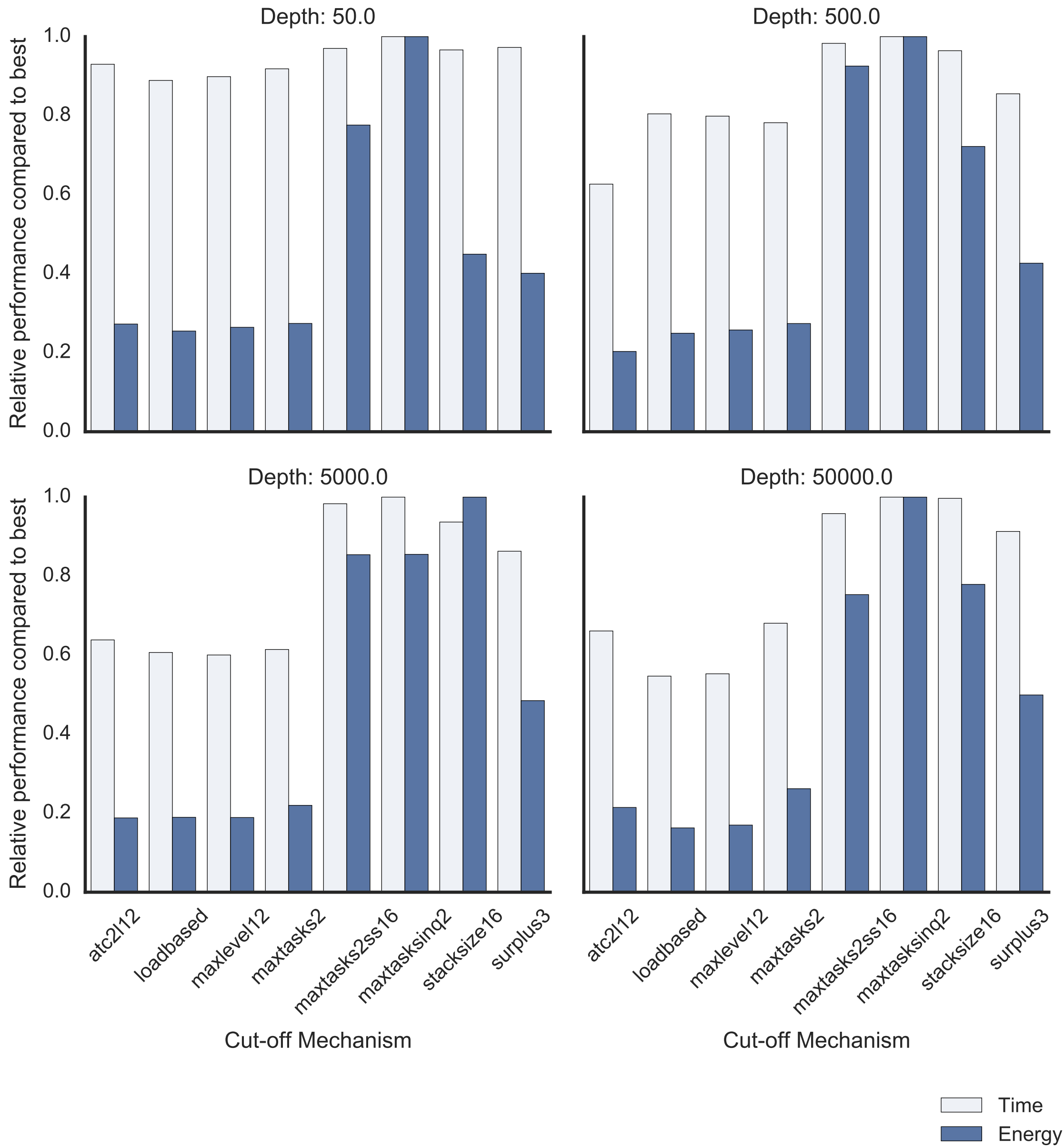

Figure 6.7: Time and energy relative performance for different depths with unbalanced binary branching. Higher is better.

Figure 6.7 shows the performance of time and energy of unbalanced programs with a light workload with different depths. Over all depths, the three novel approaches introduce in Chapter 5, MaxTasksinQueue, MaxTasks with StackSize and StackSize, perform better, especially in terms of energy. StackSize-based cut-offs provide a depth-based approach, but consider the internal runtime recursive calls instead of just program calls. This results in a better granularity, avoiding the cre-

ation of tasks in high depths of the program, except when the queue is available. MaxTasksInQueue, on the other hand, tries to limit the amount of tasks in a particular queue, the ideal case for these programs in which one worker can create a large number of tasks to be stolen by other workers.

6.3.4 Real-world programs

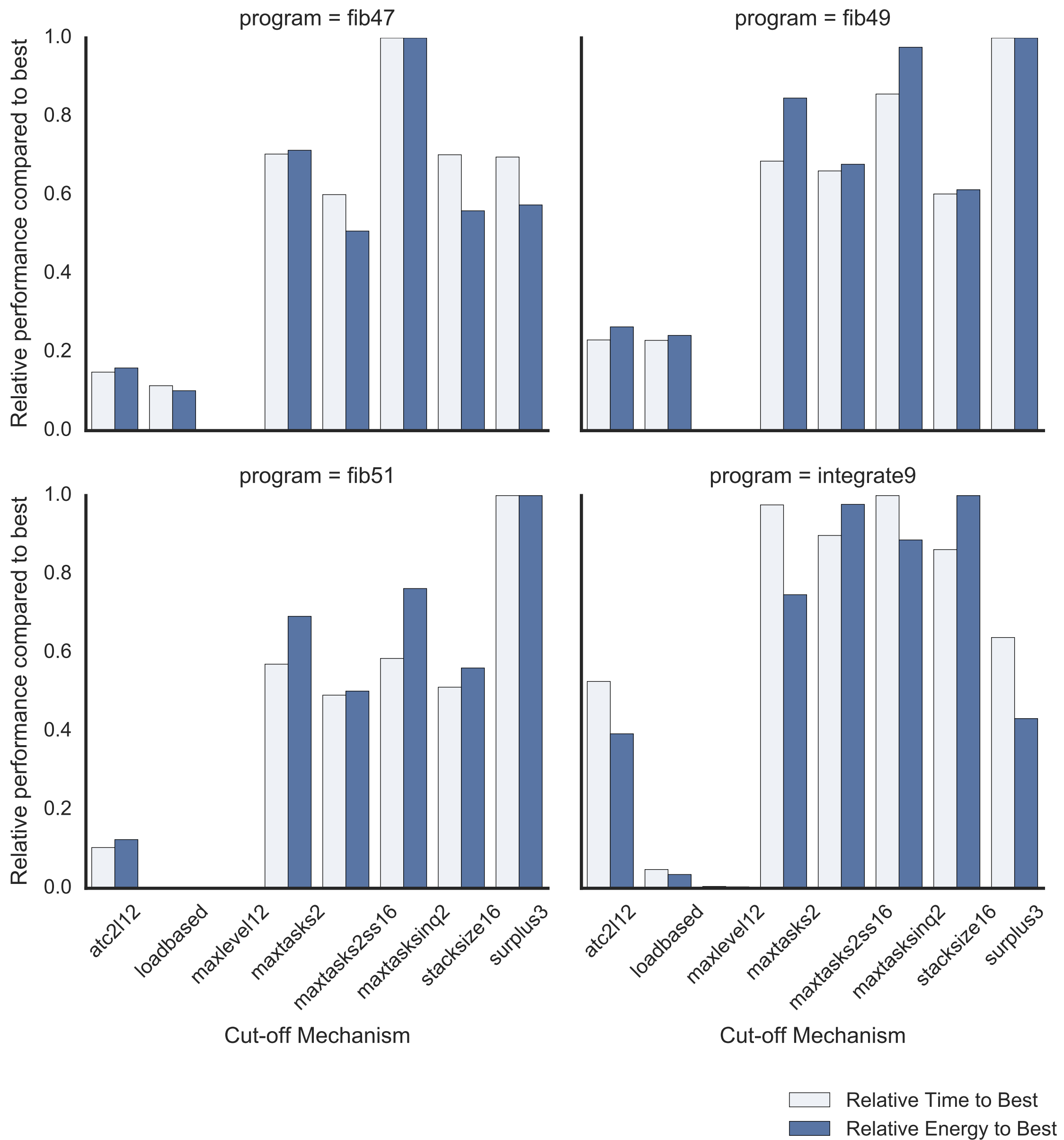

Figure 6.8: Time and energy relative performance in irregular real-world benchmarks. Higher is better.

Figure 6.8 shows the performance of different cut-off algorithms in terms of speed and performance on real-world irregular algorithms. The behavior of the synthetic program with unbalancing branching and lightweight work was confirmed in real programs.

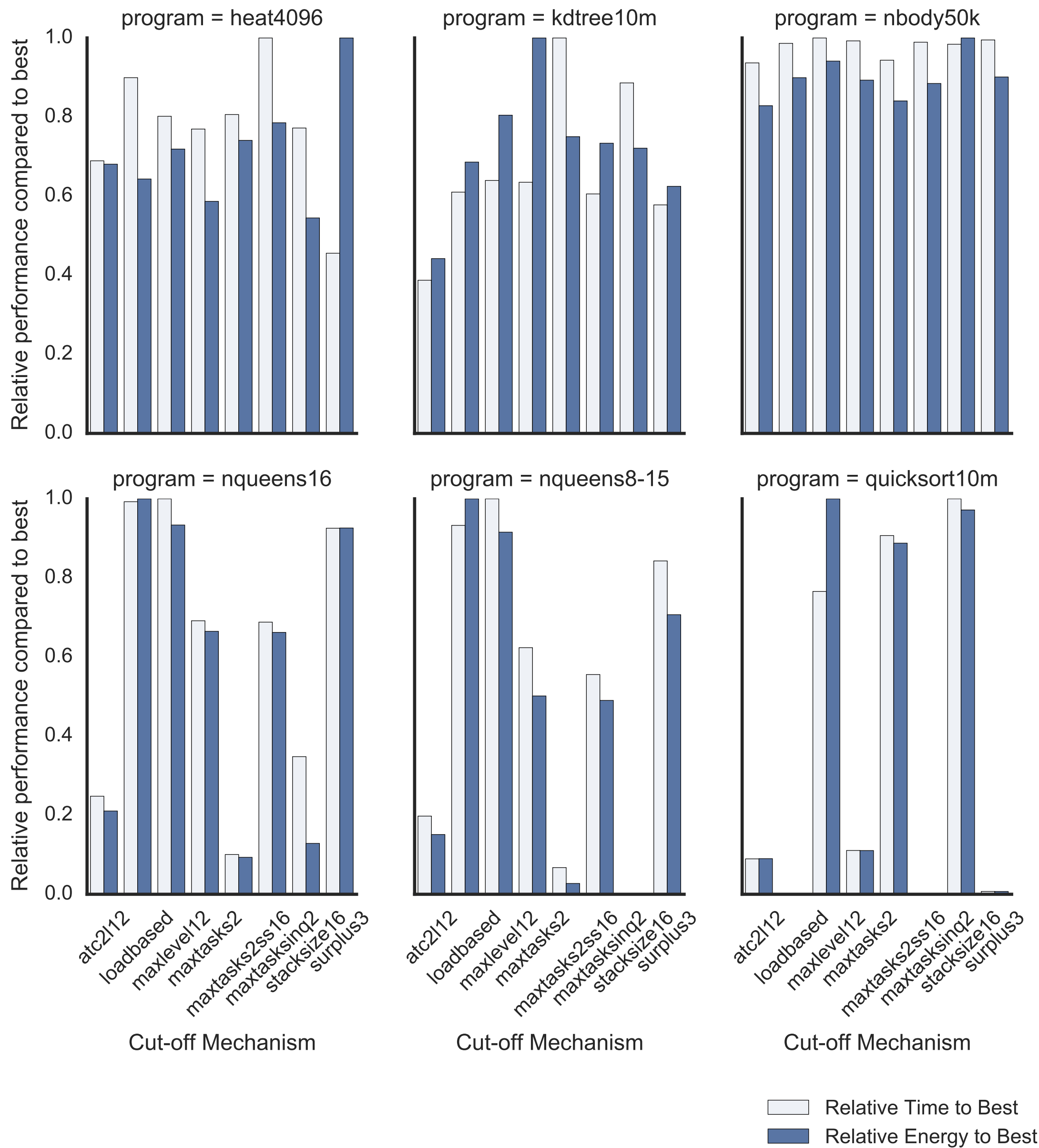

Figure 6.9: Time and energy relative performance in real-world benchmarks where the best time-efficient cut-off is not the most energy-efficient. Higher is better.

Real-world benchmarks could also confirm that there are cases where the best cut-off in terms of time is not the best in energy consumption. Figure 6.9 shows examples, with consistent differences in Nqueens, for instance. StackSize approaches generally achieve a better performance in speed than in energy.

It was clear that different programs performed better with different cut-off algorithms. If a programmer wants to optimize energy consumption, the best cut-off might not be the same that he would select for speed-up.

Figure 6.10 shows different metrics to evaluate the error of using each cut-off for all programs. The ratio of incomplete programs shows the cases in which the execution timed out (1 hour timeout, which is higher than the execution time of the serial version). The other two metrics are the percentage of time or energy that could have been saved by using the best cut-off for each program.

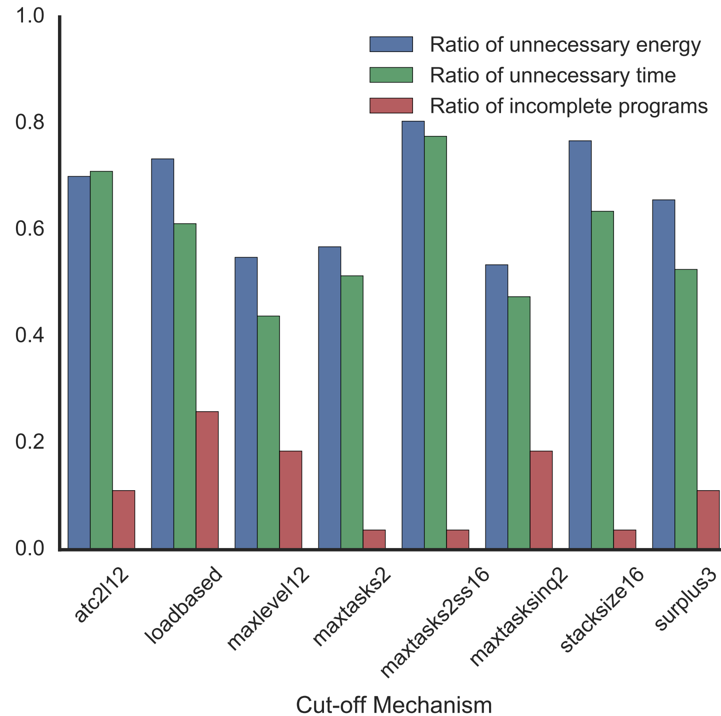

Figure 6.10: Evaluation of different cut-off approaches over the whole real-world dataset. Blue bars represent the ratio of time spent by that cut-off that could have been save by using the best cut-off for each program. Green bars represent the same ratio, but for energy. Red bars represent the ratio of programs that could not execute within the 1 hour threshold. Lower is better.

MaxTasks, StackSize and MaxTasks with Stack size had the lowest incomplete ratio, which means they are more general and can handle different types of programs, albeit not having the best performance in time or energy. Among the three, MaxTasks had the best performance in energy and time.

6.4 Conclusions

In this chapter, the time performance and energy consumption of different cut-off techniques have been evaluated. A synthetic benchmark has been used to emphasize individual program characteristics that can influence the choice of cut-off algorithms, and real-world benchmarks.

The synthetic benchmark was used to study the influence of branching, workload and depth on energy and time consumption. It has been concluded that for balanced programs, of small and high workloads, depth-based approaches such as MaxLevel and ATC tended to perform better. Medium workloads performed better under MaxTasks in both time and energy.

In unbalanced programs, the workload per leaf had a larger impact. The smaller the workload, the more irregular the program would be. Dynamic approaches such as MaxTasks, MaxTasksInQueue and Surplus performed better in those cases. In programs even more irregular, unbalanced with light workloads and a high depth, StackBased approaches were the best.

These conclusions has been confirmed on real-world benchmarks. Furthermore, cut-offs have been globally evaluated on the benchmark set. Despite MaxLevel having the best execution time, and MaxTasksInQueue the lowest energy consumption, both cut-offs failed to complete a large set of programs. For a more conservative choice that gives priority to finishing all programs instead of being faster in a few, MaxTasks and Stack-Based cut-offs are the best, with MaxTasks being the best in terms of time and energy.

Chapter 7

Using Evolutionary Algorithms to Optimize Granularity Algorithms

Chapter 5 showed evidence that existing and novel granularity algorithms all performed equally over all types of problems. Chapter 6 confirmed that, in both time and energy, the best cut-off algorithm was not the same for all programs. This chapter attempts to artificially create a granularity algorithm better than the existing, and to develop an approach to evolve a granularity and configuration for a single program, through the usage of a Genetic Algorithm (GA).

7.1 Introduction

As previously seen, different programs have a different best granularity control mechanism. It was also concluded that in some cases the combination of two different algorithms improves the performance in some programs, but not always. Selecting the best cut-off mechanism is also non-trivial as there are no features that are discriminatory enough.

Additionally to cut-off mechanisms, there are other work-stealing runtime configurations that can influence the behavior of a parallel program, such as the stealing algorithm, the parking interval, the Lazy Binary Splitting PPS value, etc.

Thus, when considering both cut-off and work-stealing configurations, looking for the best setup among the cross-product of all possible setups might not be feasible. Such task would be too overwhelming for humans and a test-and-run approach would be computationally too expensive to be useful.

In this work two problems are tackled: finding a global configuration that can efficiently handle regular and irregular programs and finding the best configuration for each program. This is done by applying a Genetic Algorithm (GA) where the genotype of each individual is represented as a combination of up to three cut-off conditions, stealing algorithm and values for runtime configuration. A random population is evolved using single-point recombination and mutation, and tournament and elitism are used for selecting individuals based on individual benchmark test performance.

Firstly, the problem is stated, including the definition of program configuration (Section 7.2). Then, the evolutionary approach is described, including parameters of the GA (Section 7.3). The GA is evaluated (Section 7.4) using a large testing set and, finally, conclusions are drawn (Section 7.5).

7.2 Parallel Program Configuration

Different cut-off algorithms have been presented and it has been shown how they influence the performance of a program. Moreover, combining different algorithms can be useful in some cases. Thus, the configuration of a program will include the usage of one or more algorithms in the configuration of the parallel program.

Another important configuration is the stealing algorithm, that can interact with the cut-off mechanism to impact the performance of the program. The stealing algorithm is used to select which queue will a worker-thread steal tasks from. There are several approaches: **SequentialReverseScan**, that tried to steal tasks in reverse order; **StealFromMaxQueue** tries to steal from the largest queue; **MinLevel** tries to steal the task with the smallest depth in the calling graph, based on the assumption that it has more work; and **Revenge** tries to steal from tasks which stole from this task earlier, trying to maximize cache locality.

Besides the cut-off and work-stealing algorithms, there are other configuration parameters such as the **unparking interval**, that represents for how long a thread sleeps until it checks if there is work available to steal elsewhere. Finally, the **maximum size of the queue** represents the maximum size of each queue, after which tasks will be executed instead of queued. It is important to distinguish between this limit, and the MaxTasks cut-off. MaxTasks prevents tasks from being created, while this limit only prevents them from being queued, but the overhead in task creation always occurs. Setting the maximum size of the queue to any value above the MaxTasks threshold will have no impact in the program.

In summary, it is possible to manage the execution of a recursive parallel program over a task-based runtime by configuring the following parameters:

- Cut-off Mechanism and its parameters, or a combination of several cut-offs
- The PPS threshold for Lazy Binary Splitting
- The stealing algorithm
- The maximum queue size
- The unparking interval

It is important to note that there is no defined rule on how to choose a certain parameter value. Parallel programs are very heterogenous and the best choice depends heavily on the program and platform.

This work addresses that issue by trying to find ways to find a global cut-off that has a good performance, and by finding how to improve the configuration of a single program. Considering a program with an average execution time among all configurations of 1 second, trying all possible combinations would take 48 thousand million years. Thus, it makes sense to use an heuristic-based approach.

7.3 An Evolutionary Algorithm for Parallel Program Optimization

The identification of the ideal configuration is a very complex problem, which cannot be obtained directly, given the dependency between the configuration parameters. This is a case of Search-Based Software Engineering (SBSE) (Clarke et al., 2003), in which metaheuristics search techniques are used to find near-optimal solutions for complex problems in the software engineering field, which otherwise would be extremely computationally expensive to obtain.

A Genetic Algorithm (Goldberg and Holland, 1988) is proposed to tackle this specific problem of finding the best configuration for a subset of programs. A genetic algorithm uses a representation of the solution (genotype) that can be evaluated using a fitness function. Starting with a pool of random individuals, each corresponding to a possible representation of the solution, the GA iteratively selects and combines existing individuals to generate new individuals, replicating the natural selection process. The overall algorithm is described in Algorithm 2.

Algorithm 2 General Genetic Algorithm

```
1: pop_0 ← POP_SIZE ∗{random_individual()}
2: for it = 0.. ITERATIONS do
3:     evaluate(pop_it)
4:     sort(pop_it)
5:     pop_it+1 ← {pop_e, for e = 0.. ELITISM }
6:     pop_it+1 ← NOVELTY ∗{random_individual()}
7:     for i = (ELITISM + NOVELTY)..POP_SIZE do
8:         if random() < PROB_RECOMB then
9:             parent1 = tournament(TOURN_SIZE)
10:            parent2 = tournament(TOURN_SIZE)
11:            child = crossover(parent1, parent2)
12:        else
13:            child = tournament(TOURN_SIZE)
14:        end if
15:        pop_it+1 ← {child}
16:    end for
17:    for i = 0..POP_SIZE do
18:        if random() < PROB_MUT then
19:            pop_it+1,i = mutate(pop_it+1,i)
20:        end if
21:    end for
22: end for
```

7.3.1 Genotype

Choosing the genotype is one of the main decisions when designing a Genetic Algorithm. Work-stealing and PPS parameters could be represented as integers, but the cut-off algorithm requires a more complex representation. A Genetic Programming approach, using a typed AST that could be compiled to a runtime Java expression

Table 7.1: Possible Cut-off Variables

Variable	Description
queueSize	Length of the current queue
level	Depth of the parallel recursion
stacksize	Number of allocated stacks
activeThreads	Number of threads executing tasks
idleThreads	Number of threads not executing tasks
surplus	Length difference between local and average queues
totalTasks	Number of active and pending tasks
emptyQueues	Number of empty queues
memory	Percentage of used memory in the machine
cpuLoad	Percentage of CPU load in the machine
true	-1000, so the condition is always true.

was considered. In early experiments, most of the iterations were introducing bloat, instead of finding better solutions. Since these evaluations are very time consuming, a solution with a higher learning rate was required. A fixed expression to represent the condition was developed, as all existing cut-offs could be represented using it: $\mathsf{Var}_1 < \mathsf{Threshold}_1 \dagger_1 \mathsf{Var}_2 < \mathsf{Threshold}_2 \dagger_2 \mathsf{Var}_3 < \mathsf{Threshold}_3$. This approach is able to combine up to three conditions, which was shown to be enough, as increasing the number of conditions would also increase the overhead in the decision process, which is executed once for each recursive call. † represents either the **and** or the **or** binary operators, following the Java semantics. Threshold can represent any number from 0 to 100, which was a sensible maximum for any of the possible variables. Var could take any of the following values available in the runtime listed in Table 7.1. The *true* option was introduced to allow cut-offs with less than three conditions, resulting in a condition where -1000 is always less than any number between 0 and 10.

The final genotype is described in Table 7.2.

7.3.2 Operators and general configurations

Recombination is performed by a single point crossover between two parents. A random point of the genotype is selected, and the child inherits the genes before that random point from parent 1, and the remaining from parent 2. The mutation operator is different per gene. The mutation of non-integer genes randomly selects an option from the alternative list. In the case of integers, it has a 75% chance of adding or subtracting a number within 5% of the maximum value, and 25% chance of replacement by a random integer within the range of that gene. Part of this mutation tries to simulate local-searching around that threshold.

The population size of the GA was set to 25, which is small. The reason for

Table 7.2: Genotype

Gene	Description
var_1	One of 11 options in Table 7.1
$threshold_1$	Integer between 0 and 100
$\dagger_1$	**and** or **or** binary operators
var_2	One of 11 options in Table 7.1
$threshold_2$	Integer between 0 and 100
$\dagger_2$	**and** or **or** binary operators
var_3	One of 11 options in Table 7.1
$threshold_3$	Integer between 0 and 100
PPS	Integer between 0 and 100
Stealing	One of: StealFromMaxQueue, MinLevel, Revenge and SequentialReverseScan
maxQueueSize	Integer between 0 and 1000
unparkingInterval	Integer between 0 and 1000

this, is that evaluating each configuration is very time-consuming, thus a larger population would penalize execution time. The algorithm executes 100 iterations, which is enough for fitness to stabilize.

The recombination rate is 90% in order to test several combinations of cut-offs and other configurations. The mutation rate is 50%, higher than the traditional for the same reason. Mutating thresholds is desirable as they impact the performance of programs.

In order to keep the best approach until the final generation, elitism was defined with value 2, and to always introduce new genetic material and avoid local minima, a new random individual is introduced at each generation.

7.3.3 Fitness Evaluation

In order to evaluate the fitness of each solution, the configuration is applied to the Æminium Benchmark suite (Fonseca, 2013). Table 7.3 describes the subset of the benchmark used in the training set. In the training set, programs have smaller inputs in order to reduce the evaluation time. The testing set will be used to evaluate the generalization capability of our algorithm.

The complete fitness evaluation incorporates compiling the cut-off expression to Java source code, and writing the other parameters to a configuration file. Then, each of the programs is executed once and recorded. Parallel programs can have a high standard deviation in regards to execution time. However, if two cut-off approaches are similar, either one or both can be kept, as the elitism value is 2.

Program	Training input size	Type	Balance
BlackScholes	1000^2	For-loop	Regular
Do-All	1 million	For-loop	Regular
FFT	131072	Recursive	Regular
Fibonacci	n=41	Recursive	Skewed
Health	l=4	For-loop	Regular
Heat	1000x1000, it=1024	For-loop	Regular
Integrate	error=10^{-11}	Recursive	Skewed
Matrix Multiplication	p=10000, q=r=1024	For-loop	Regular
N-Body	n=50, it=3	For-loop	Skewed
N-Queens	n=13	For-Loop	Irregular
Pi	n=100.000	For-loop	Regular

Table 7.3: Description of the programs used in the benchmark

Additionally, there is a timeout of 10 seconds per evaluation because the training benchmark was designed in order to have 1 second programs with a reasonably good cut-off and default parameters. Timed out programs are considered to have 1000 seconds of execution.

For each individual, the execution time in each program is recorded. The fitness of each individual is the sum of the execution times for all programs.

7.3.4 Selection operator

For selecting an individual for recombination, or for the next generation, a tournament operator (Miller and Goldberg, 1995) is used. Initially, a roulette wheel (DeJong, 1975) was being used, but at generation 50 all the population had the same genotype. A tournament among 4 individuals revealed to be a solution with higher diversity and achieved better results. However, the implemented tournament did not consider their fitness. Instead, each tournament compared the performance of different individuals in a single random benchmark. The reason for this custom tournament is to try to cross algorithms that solve different types of programs together.

7.4 Evaluation

In this section, the proposed evolutionary algorithm is evaluated on a large test suite to evolve a general configuration, as well as a specific configuration for individual programs.

7.4.1 Experimental Settings

These experiments were conducted on 3 different machines, each with different characteristics, featured in Table 7.4. *server32* and *server24* had Ubuntu 14.04 installed

Name	Processor	CPU Cores	Threads	RAM
server32	Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz	16 cores	32 threads	32GB
server24	Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	12 cores	24 threads	24GB
server8	Intel(R) Xeon(R) CPU E5420 @ 2.50GHz	8 cores	8 threads	16GB

Table 7.4: Details of the hardware used in the experiments.

while *server8* had CentOS 6.7. The experiments executed on top of the Æminium Runtime (Stork et al., 2014), executing on the Java HotSpot 64-Bit Server VM 1.8.

The time measure was for the parallel algorithm alone and runtime overheads, excluding the data setup required. Because of being time-consuming experiences, only one execution of each configuration-program pair was executed. In some cases, several seeds were used in the GA, and the results mention that.

7.4.2 Training dataset

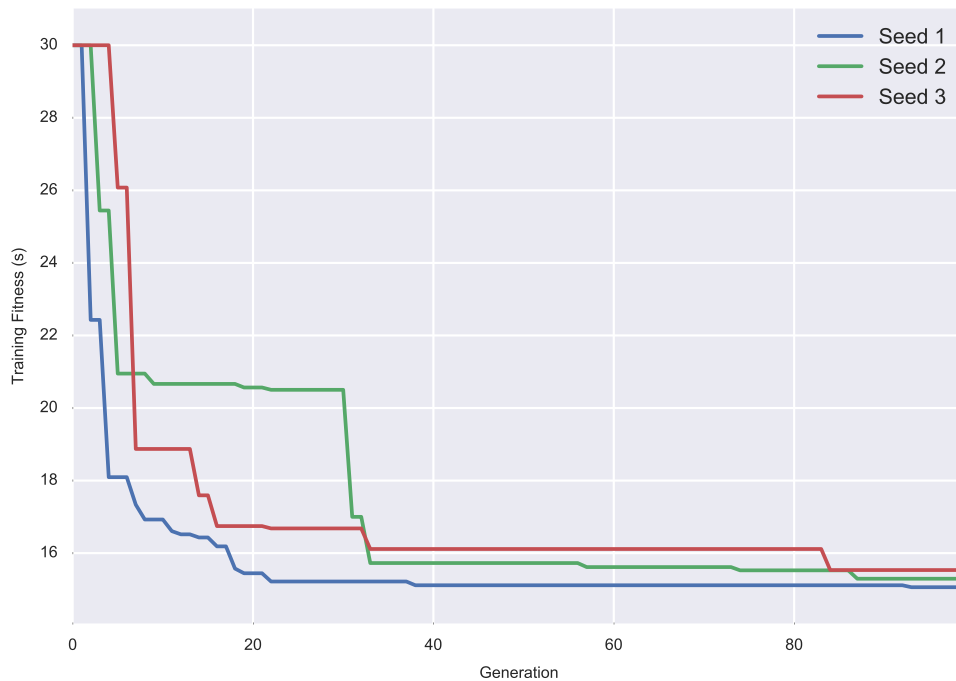

Figure 7.1: Fitness of the best individual of each generation of the training dataset on *server8*.

Figures 7.1, 7.2 and 7.3 show the fitness of the best individual of each generation, each figure showing 3 different seeds with 30 seconds as the maximum value possible. In all cases, the performance of the best configuration improved over time. *server8* had the biggest different in performance from the best random population to the last generation. Since it has fewer cores, the machine is more sensible to the cut-off algorithm. *server32* has the smallest different, reflecting the same conclusion.

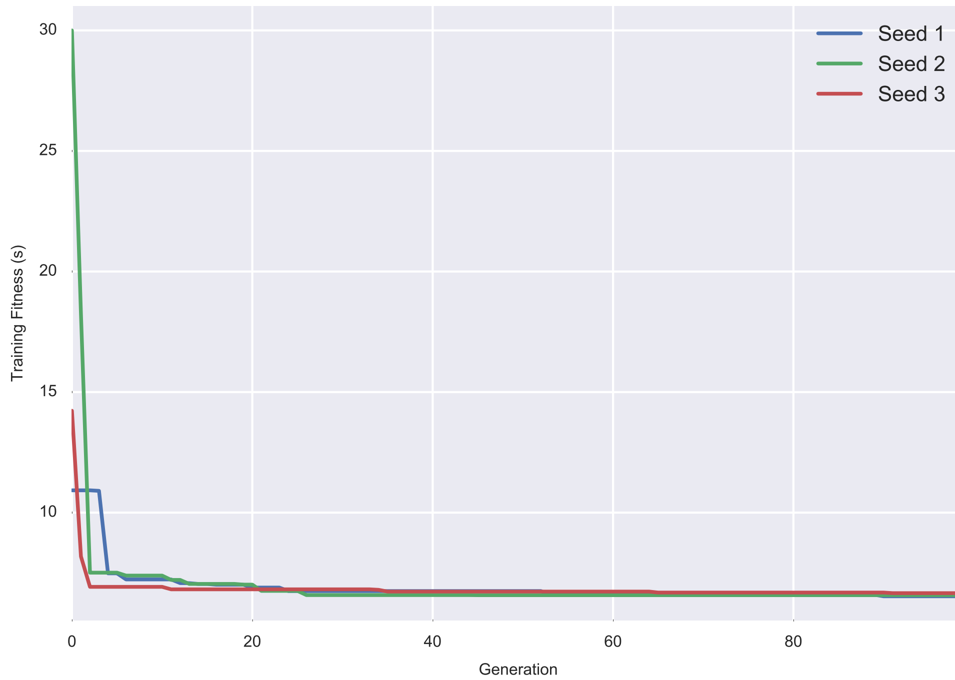

Figure 7.2: Fitness of the best individual of each generation of the training dataset on *server24*.

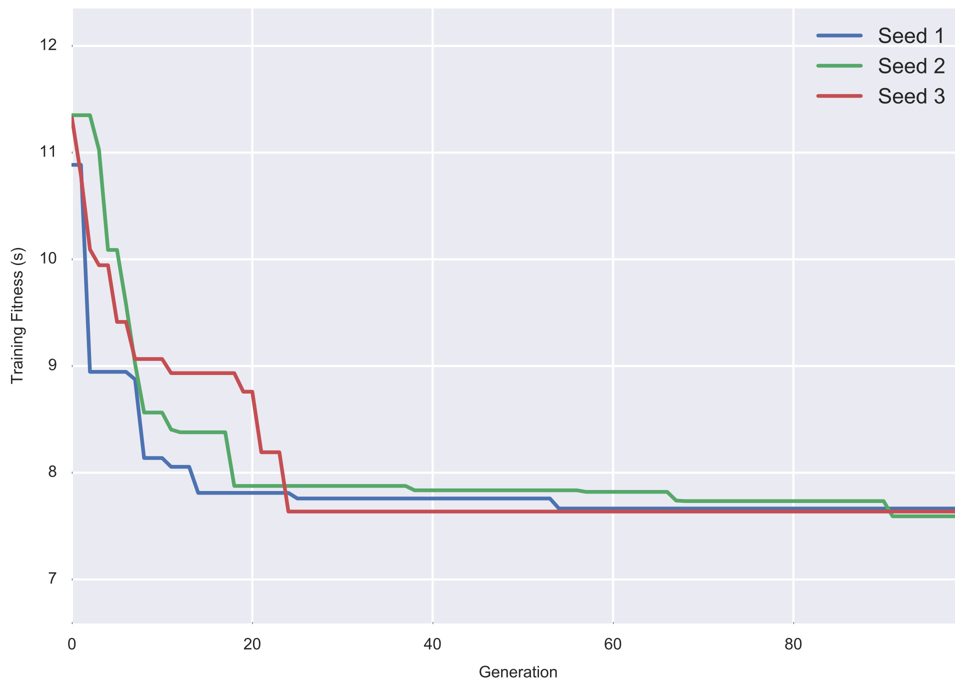

Figure 7.3: Fitness of the best individual of each generation of the training dataset on *server32*.

In *server24* and *server32*, the GA obtained a good configuration quite quickly, having less than 1 second of difference between generation 50 and 100. That 1 second could also result from variations inherent with parallel programs. From these results, only 50 generations were considered for other benchmarks.

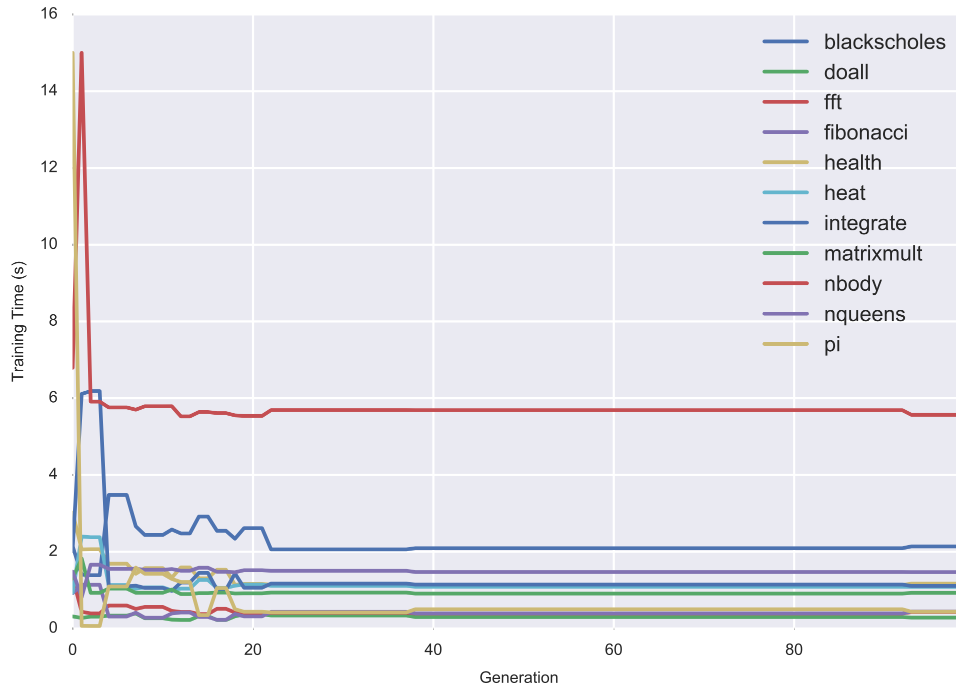

Figure 7.4: Execution time per training program of the best individual of each generation of the training dataset on *server8*.

Figures 7.4, 7.5 and 7.6 show the fitness of the best individual of each generation distributed by each benchmark program, with a maximum value of 15 seconds. After the first 10 generations, the GA is trading the performance of one algorithm for the others, resulting in small marginal improvements. These small improvements could be useful for improving a single program, but not necessarily for improving a whole benchmark. It can be concluded that the first generations improve over the majority of programs, but after some generations the GA is only changing configurations to achieve a better balancing of configurations. In Figure 7.5 and 7.6 it is possible to notice that the best performance of some programs were in the first generations. It can be concluded that the vast improvement of some programs is done at a smaller cost of other programs.

7.4.3 Testing dataset

The full dataset is introduced to evaluate the generalization capability of the GA. The testing benchmark has more programs and they have a larger input size, which results in long-running programs. Figures 7.7, 7.8 and 7.9 show the fitness on the testing benchmark of the best individual of each generation of the training benchmark, each figure showing 3 different seeds with 30 seconds as the maximum value possible.

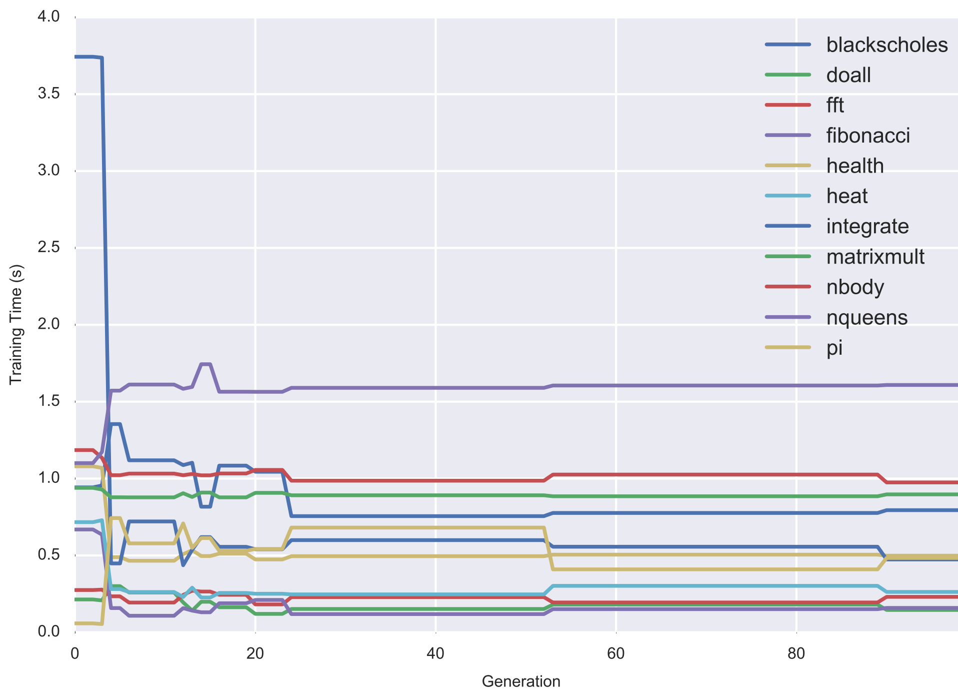

Figure 7.5: Execution time per training program of the best individual of each generation of the training dataset on *server24*.

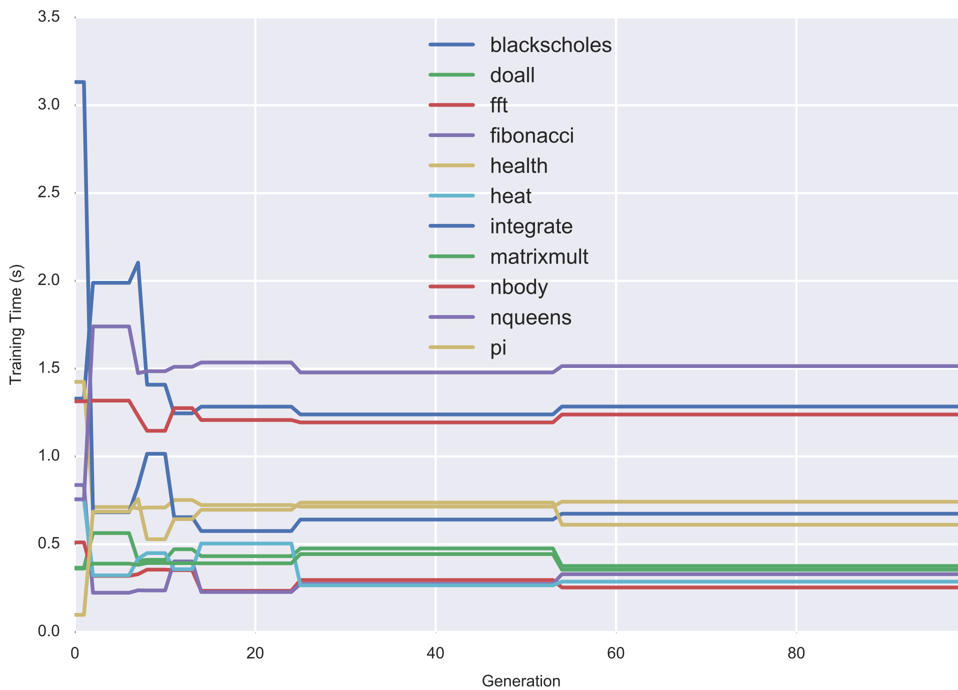

Figure 7.6: Execution time per training program of the best individual of each generation of the training dataset on *server32*.

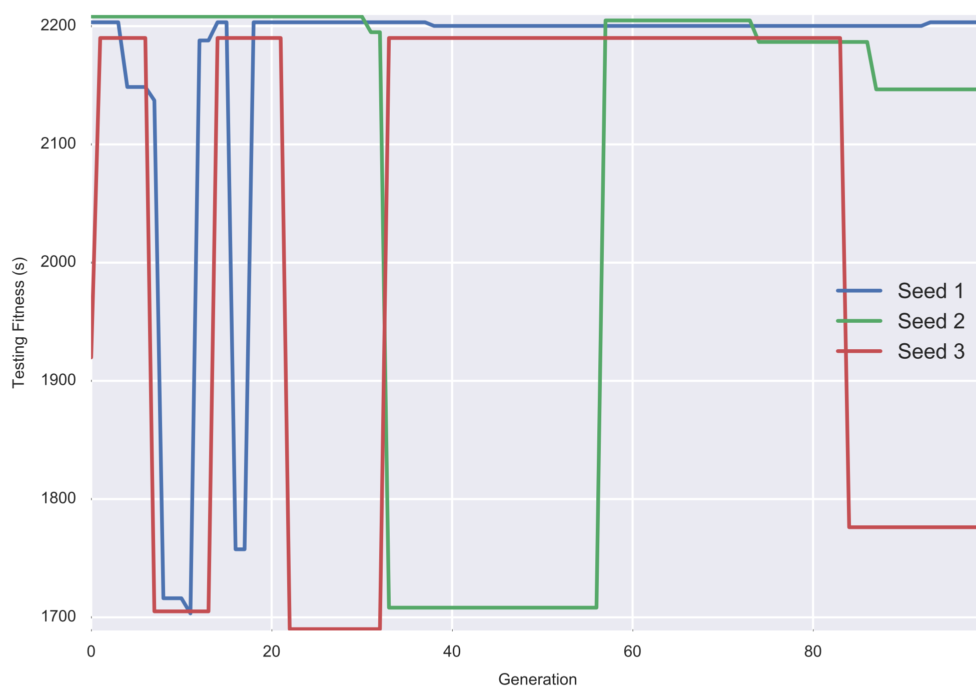

Figure 7.7: Fitness in the testing dataset of the best individual of each generation of the training dataset on *server8*.

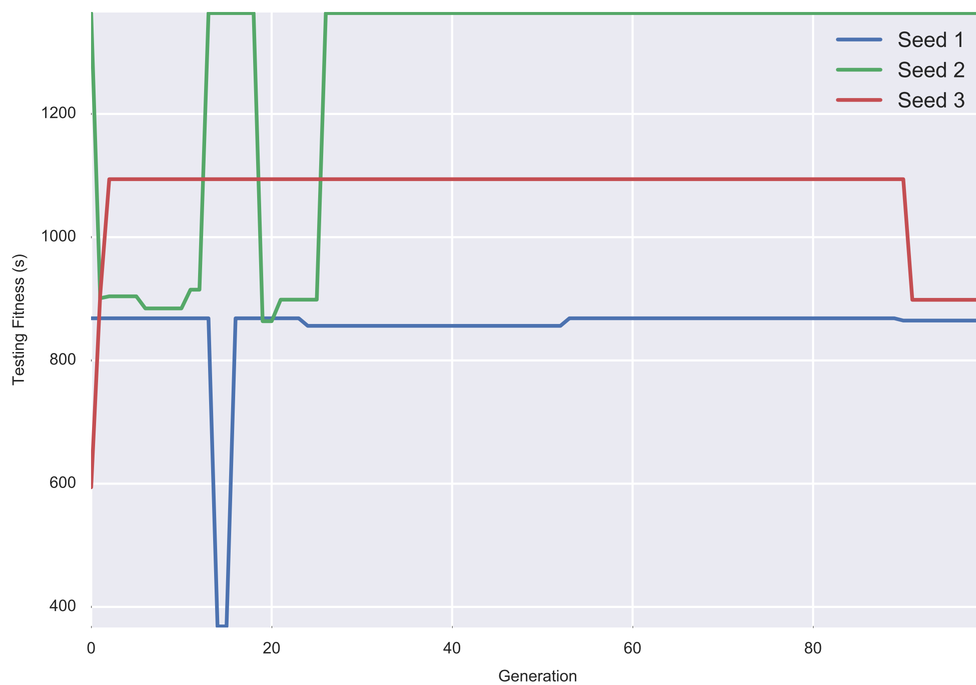

Figure 7.8: Fitness in the testing dataset of the best individual of each generation of the training dataset on *server24*.

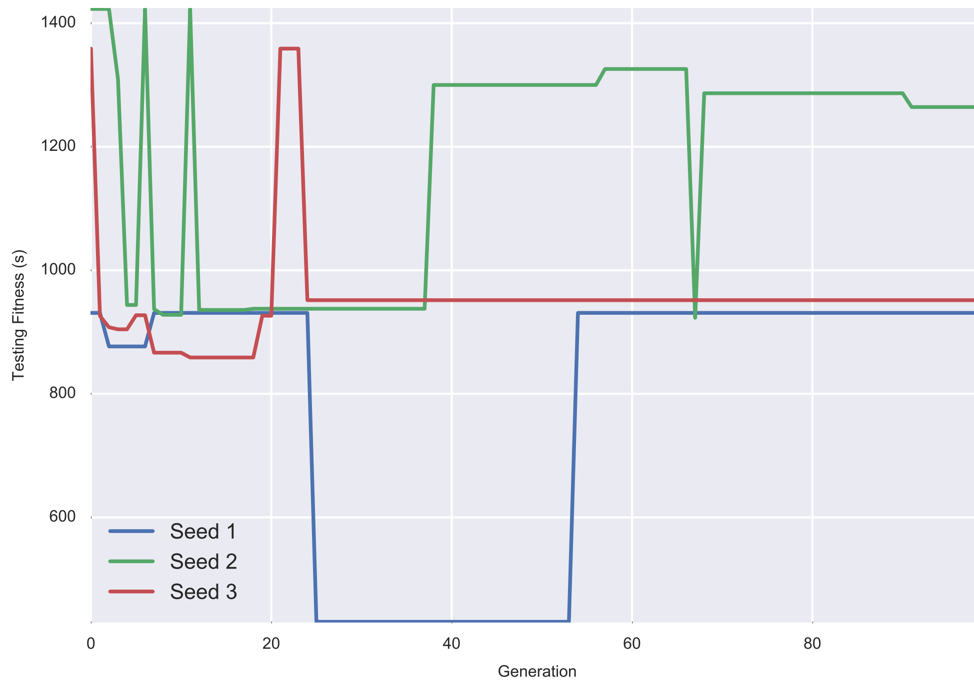

Figure 7.9: Fitness in the testing dataset of the best individual of each generation of the training dataset on *server32*.

It is possible to conclude that the GA does not achieve a good generalization, as the performance of the testing benchmark does not decrease along the generations. The reason for the lack of generation is twofold: the training benchmark is not representative enough, and configurations for short-running programs do not scale for long-running programs.

Considering the first reason, the training benchmark has 11 programs while the testing benchmark has 23 programs. While the training benchmark attempted to have as much diversity as possible, it is not representative enough. Secondly, a program that executes sequentially for X seconds divided in Y tasks will result in tasks of X/Y seconds. The same Y tasks for a larger X will result in larger tasks, which might not be optimal for the load balancing algorithm.

Figure 7.10 details the evolution of the performance on each testing benchmark program on one execution of the GA on *server8*. Most programs improve their performance, except for Neural Network, which is never executed below the time out threshold, FFT and BFS, which have random behavior. Despite degrading the performance of just two programs, the overall cost is high, because those two programs running slowly have a huge impact on the whole benchmark. Figure 7.11 shows the same results, but on *server32*. With more cores, the number of programs degrading performance is higher: 8 out of 23. In *server24*, results are similar to *server32*, which leads to the conclusion that with more cores, there is a low scalability from smaller inputs to larger ones.

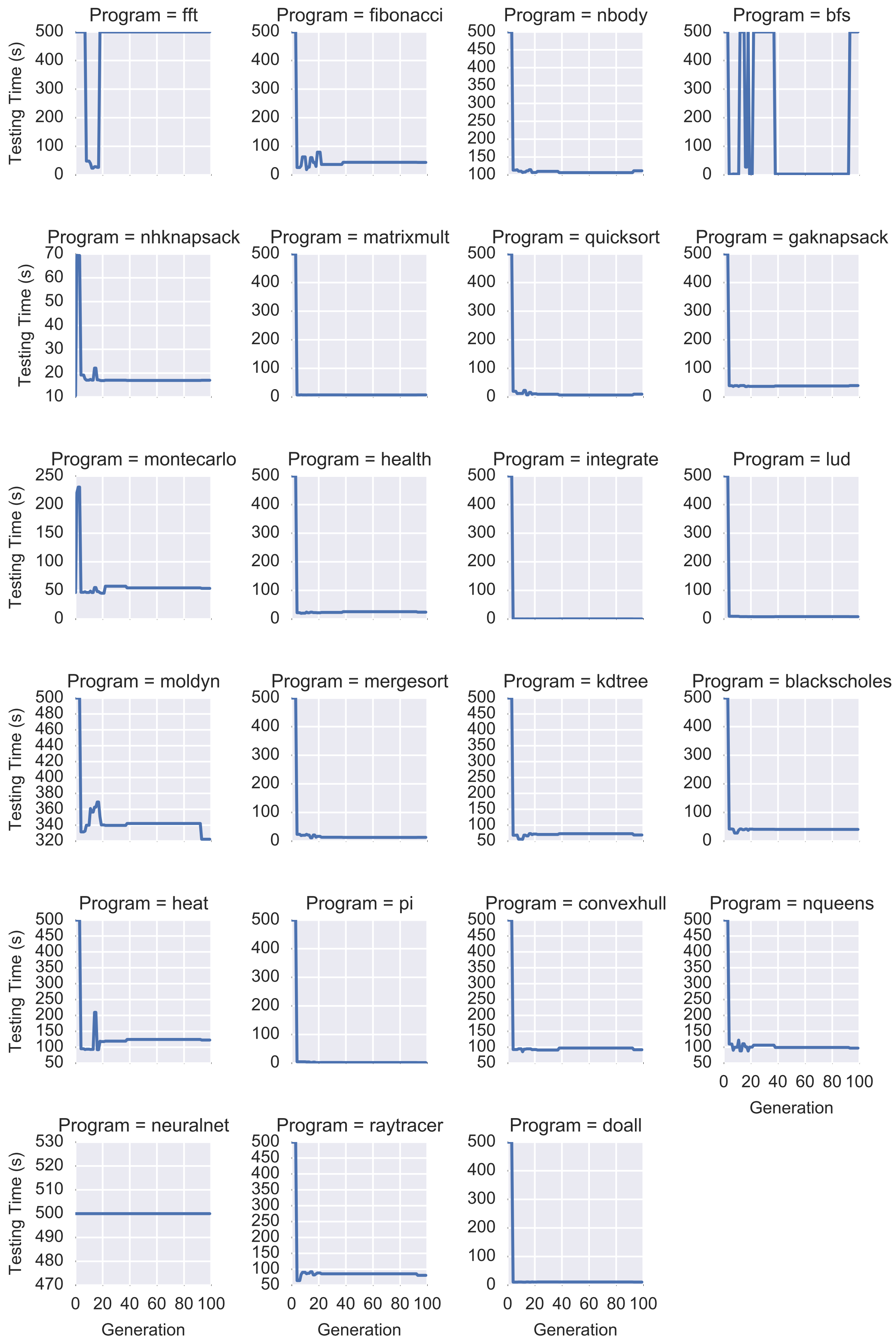

Figure 7.10: Execution time per testing program of the best individual of each generation of the training dataset on *server8*.

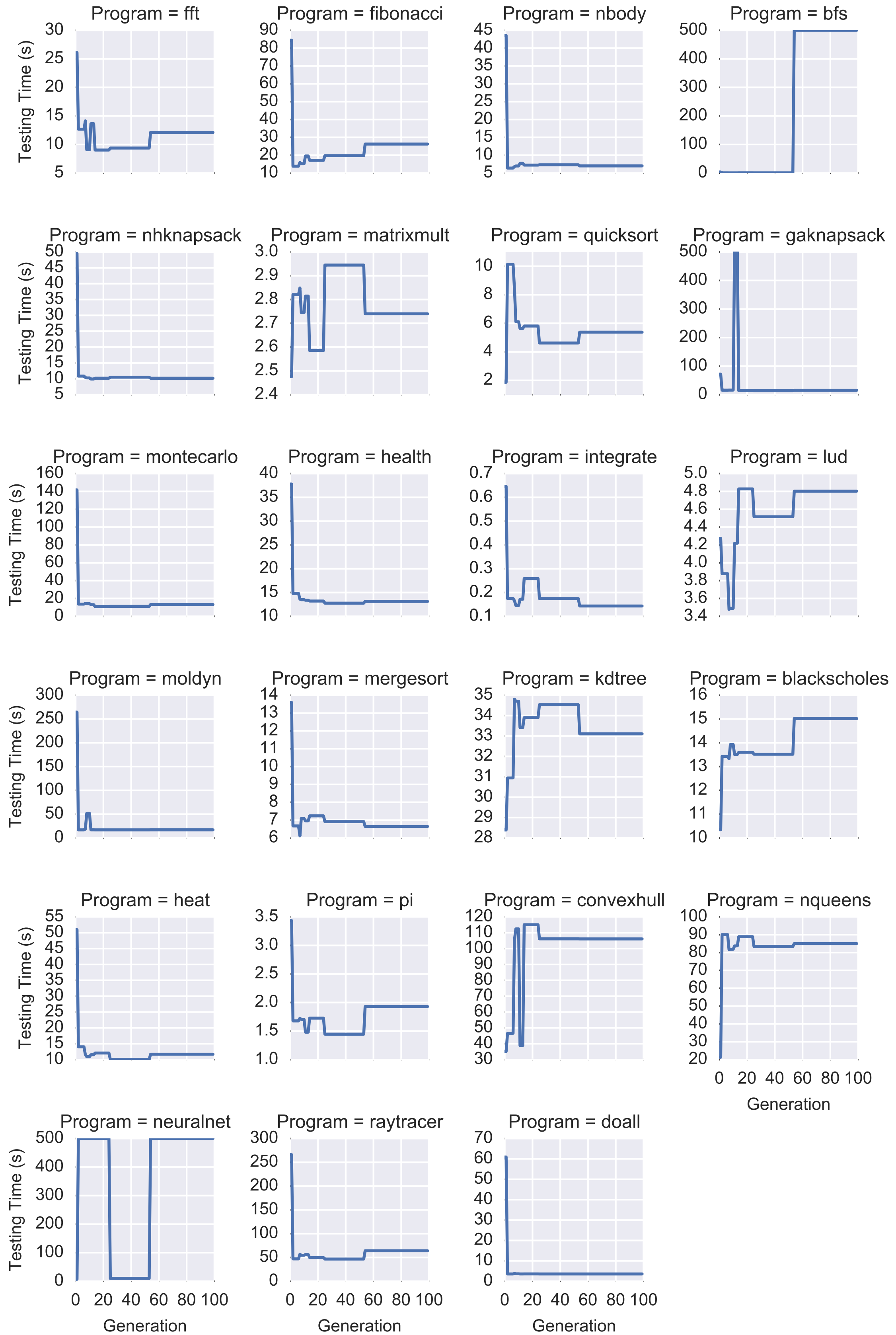

Figure 7.11: Execution time per testing program of the best individual of each generation of the training dataset on *server32*.

7.4.4 Evolving single programs

From the previous experiments, there were two important conclusions: The GA does not have a very good generalization because of a few edge cases; and the GA makes most of the relevant improvements in the first generations.

Based on these two factors, the GA can be used to improve individual programs within a small number of generations. Figure 7.12 shows the performance of the best individual for the application of the GA on different programs individually. All programs are efficiently improved, except for BlackScholes, in which it found a best solution in the initial population. This is probable in programs where a set of basic cut-off approaches have the same behavior, which is the case here.

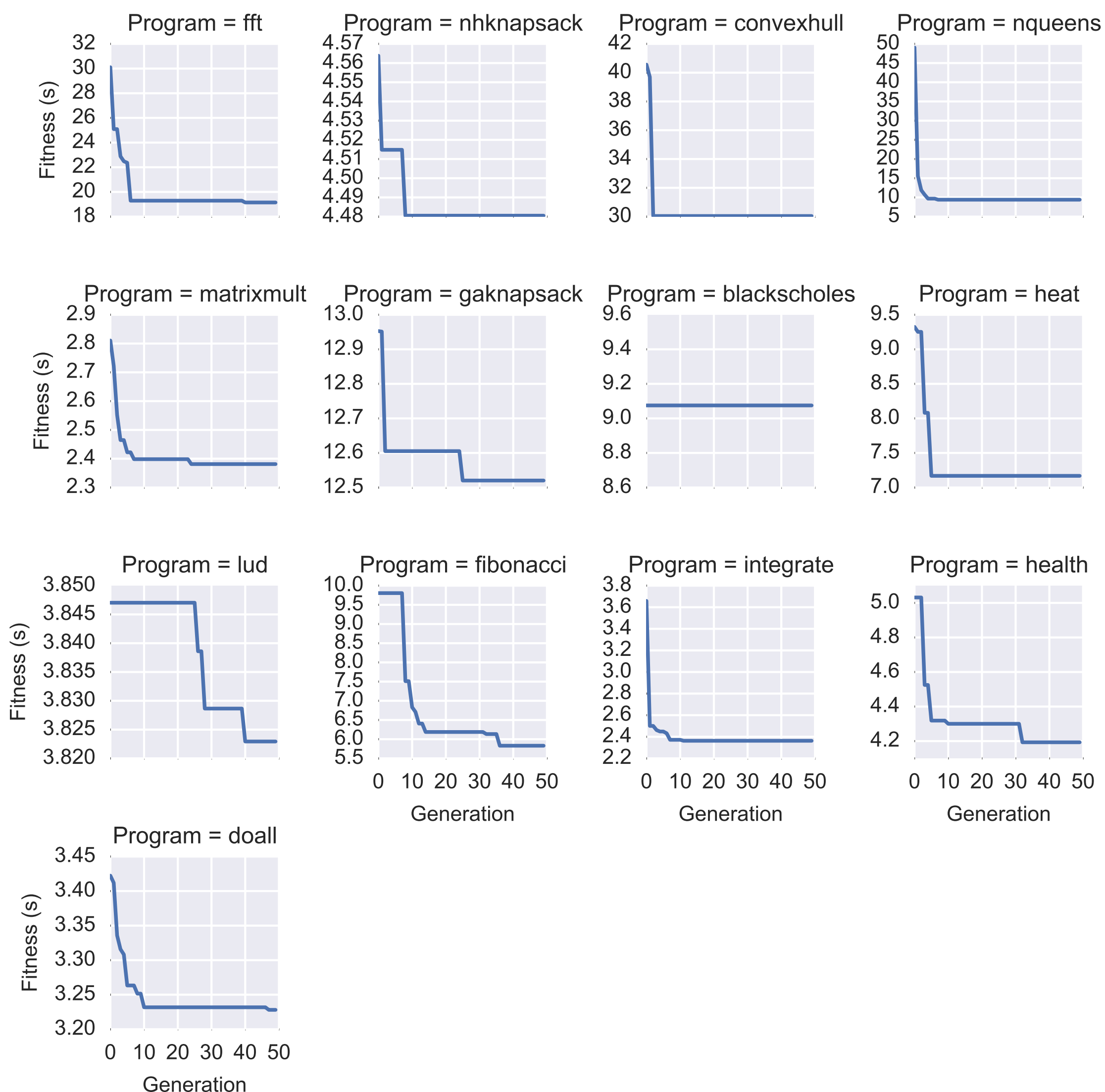

Figure 7.12: Fitness of the best individual over different instances of GA, one for each individual program.

7.5 Conclusions

A Genetic Algorithm has been proposed for the optimization of parallel programs based on the following configurations: combination of cut-off criteria, Lazy Binary Splitting intermediate iterations, workstealing algorithm, maximum recursion level and unparking interval.

The GA has been applied to a small training benchmark composed of 11 programs with small input sizes. The GA was able to successfully find a configuration that optimized the whole benchmark. The performance of some of the programs was degraded over the generations in order to greatly improve others. When the best individuals of each generation were tested against the whole 23 program benchmark, the overall performance did not improve. Most single-programs were improved, but a small number of programs would become slower, consistent with the training evaluation. In the testing benchmark, a configuration that slowed down one program, had an execution time orders of magnitude higher than the smaller improvements over the programs which were sped-up.

Additionally, the GA has also been applied to individual programs successfully. The performance of all programs was improved except for one program in which the minimum was found in the first generation.

Finally, it has been shown that despite the lack of generation of the GA for finding a universal configuration, when evaluating with this specific training and testing sets using global execution time as a metric, most of the programs were improved. Furthermore, individually applying the GA with a small number of generations (50) does improve the performance of programs and can be used for finding the best configuration for a long-running program. The proposed approach can use other metrics instead of just using the execution time, such as energy consumption.

Chapter 8

Selection of Granularity Control Algorithms

Chapter 5 concluded that cut-off algorithms generally have similar performance over all classes of problems. Chapter 6 also confirmed that in both energy consumption and execution time, the best cut-off was not the same for all programs. In Chapter 7 this conclusion was confirmed through the inability of evolving a generalized custom cut-off algorithm. This chapter attempts to identify which cut-off algorithm to use for any given set of problems, complementing the proposed model for efficient automatic parallelization.

8.1 Introduction

In the context of both automatic and manual parallelization, the choice of granularity is important and can impact the performance of the resulting program. Frameworks like Æminium and OpenMP also have the need to provide a default cut-off algorithm for programs written using their task-based API. An exhaustive evaluation of all possible cut-off algorithms is very time consuming and platform dependent. For instance, the benchmark suite used in Chapter 5 took over 3 months to test a subset of all possible cut-off algorithms.

Chapter 5 presented new cut-off algorithms that improved the performance of some parallel programs. Through the evaluation of cut-off algorithms in a set of different benchmark programs, it was concluded that no cut-off algorithm outperformed others in the whole benchmark suite.

Thus, in order to choose a cut-off algorithm for a program, it is important to understand which program features impact the choice of cut-off. In the previous evaluation, two program features have been identified as important in the decision: tree unbalance and usage of for-loops.

This Chapter addresses the choice of cut-off algorithms, considering the minimization of the **misclassification cost**, introduced in Chapter 4. Given the high variation of the performance of different algorithms across programs, it is acceptable that the best cut-off is not chosen. But in this case, the non-ideal cut-off algorithm should have as little performance penalty as possible.

This Chapter is based upon the results from Chapter 5. From these results, three approaches are presented for performing the cut-off algorithm decision (Section 8.2).

The three approaches are evaluated on their generalization ability using a synthetic benchmark (Section 8.3) and conclusions are drawn (Section 8.4).

8.2 Approaches for Automatic Granularity Algorithm Selection

Three approaches are presented for automatic granularity algorithm selection: a static approach that always selects the same algorithm, a rule-based approach based on feature analysis of the existing benchmark and a machine-learning approach.

These approaches are used to minimize two metrics: the program execution time and the program energy consumption.

8.2.1 A Static Approach

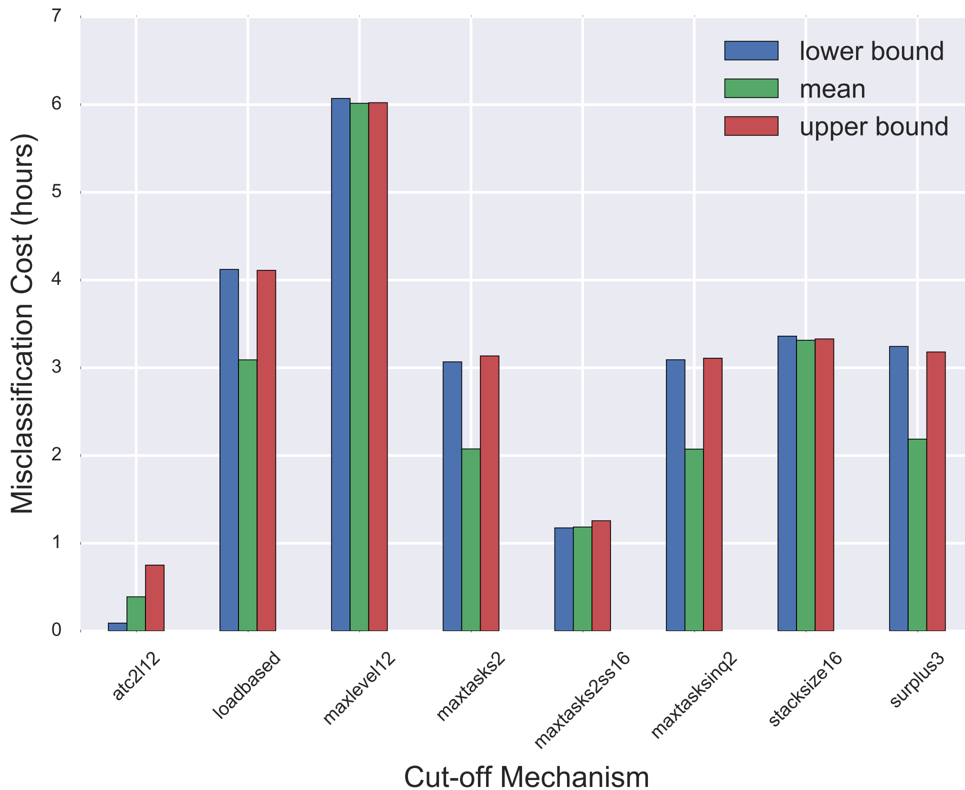

Figure 8.1: Misclassification cost for always selecting each algorithm on *server24* machine.

This section uses the results presented Chapter 5, evaluating different cut-off algorithms across a benchmark suite of 24 programs. Figures 8.1 and 8.2 show the misclassification cost for each algorithm on all programs. In the case of timeouts, the timeout value was considered for evaluation, thus a floor for the misclassification cost is used instead of the actual error. This is necessary because no cut-off algorithm was able to complete all programs on both machines (Figure 8.3 depicts the amount of completed programs). Since this is a minimization problem and all programs

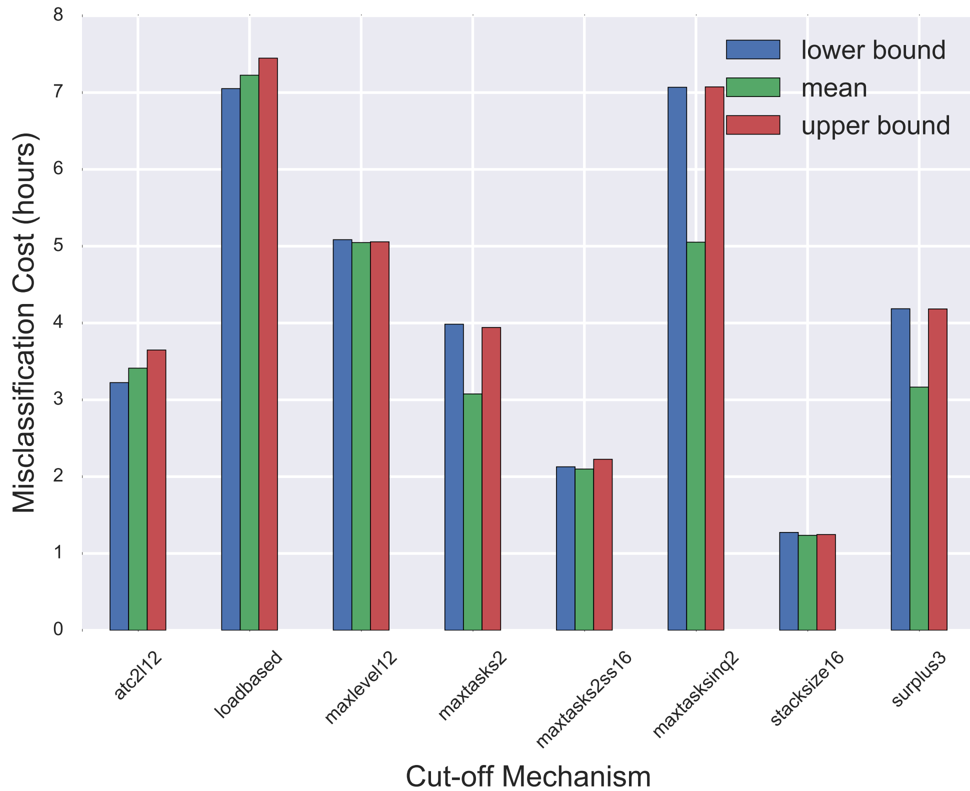

Figure 8.2: Misclassification cost for always selecting each algorithm on *server32* machine.

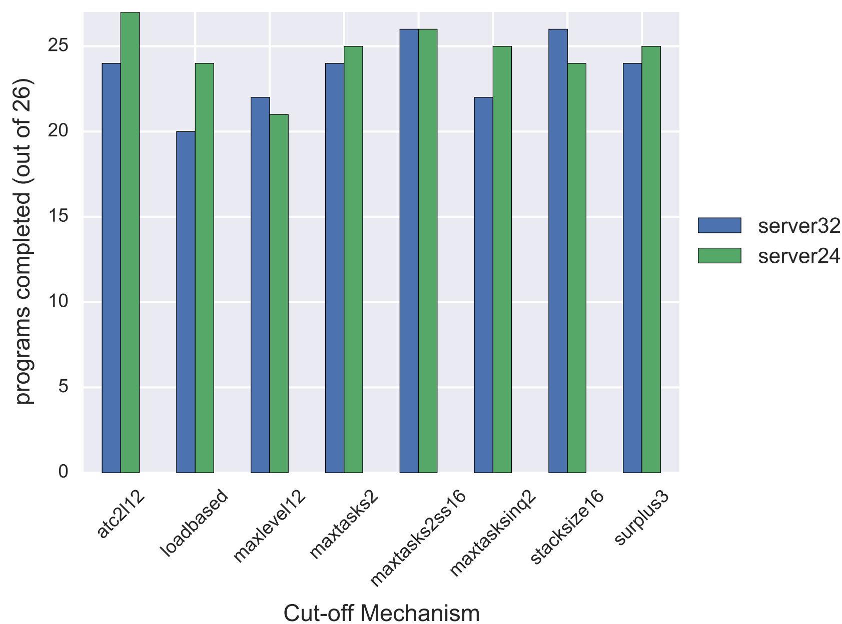

Figure 8.3: Number of programs completed in both machines per cut-off mechanism.

complete with at least one of the cut-off algorithms, considering a floor value has no major impact.

Taking into account the ability to complete the benchmark, and the misclas-

sification cost, ATC, MaxTasks with StackSize and StackSize are the overall best approaches in this benchmark suite.

8.2.2 A Feature-based Ruleset

The evaluation of Chapter 5 identified some features that impact the performance of programs. The following features were manually obtained from inspection of source code:

- *Type of For-Loop splitting mechanism* - For-loops can can be split and merged using an automatic Binary Splitting mechanism. For-Loops have less overhead than most binary recursive programs in this benchmark because there is no work done in merging results.
- *Unbalance* - A program is balanced if its children perform roughly the same amount of work. Unbalanced programs create more tasks in one of the cores, increasing the overheads in task-stealing from the other cores. These programs also have the potential to create several tasks very quickly if one of the sides creates tasks without performing any work.
- *Number of Kernels* - Some programs just perform one action, but others follow a pipeline of different parallel tasks. The later are harder to tune when using the same cut-off strategy for the whole program.
- *Branching Factor* - The number of children of a task. This is in most cases 2, although some programs can split in 4.
- *Nesting* - Whether a program has nested parallel loops or not, influences the potential amount of tasks, even if tasks only have one or two levels of depth.

Additionally, we introduced two other features based on the Cost Model presented in Chapter 3, for estimating the computational cost of Java operations, based on static analysis and program-independent micro-benchmarks. We analyzed each program and computed two metrics:

- *Seq* - the estimated cost of performing the computation sequentially
- *Overhead* - the estimated overhead of splitting the task and merging the results

Unlike other features, these values were obtained automatically by a compiler tool, using the same approach presented in Chapter 3. The values produced by the tool do not represent absolute values, but can be used for comparisons between methods and operations. Unlike the Cost Model granularity mechanism, in which the decision is done using these values, this approach uses these values, among other features, to identify the best dynamic granularity cut-off algorithm to use.

Figures 8.4 shows the best cut-off algorithm according to each one of these features. From this graph, there is no direct discrimination feature that can be used. The Cost-Model information is also not very useful in this analysis, as there are no clusters of best programs to use. One of the reasons is that only the best cut-off for a given program is shown, while there might be another algorithm that is generally better, but in that case could be a few milliseconds slower.

Figure 8.4: A distribution of each of the features per best cut-off approach, using the upper bound metric on *server32*.

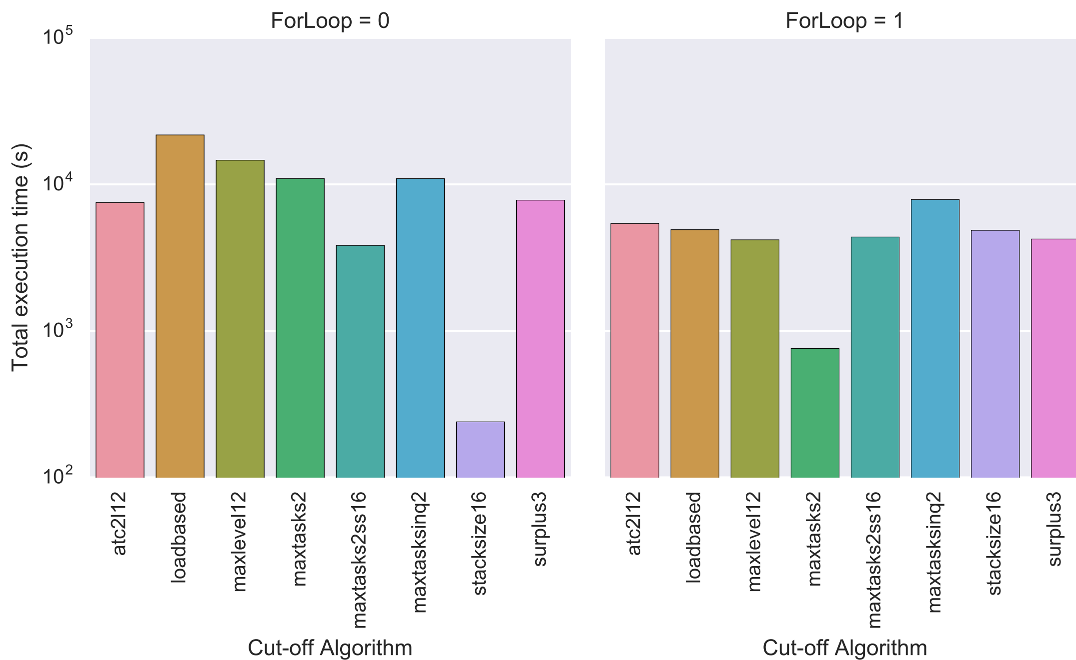

Figure 8.5: The misclassification time of programs by the ForLoop feature, on *server32*, considering the mean time of each program.

Thus, a feature-by-feature analysis can be performed to evaluate the misclassification cost of each cut-off algorithm. Figures 8.5, 8.6, 8.7, 8.8 and 8.9 show the

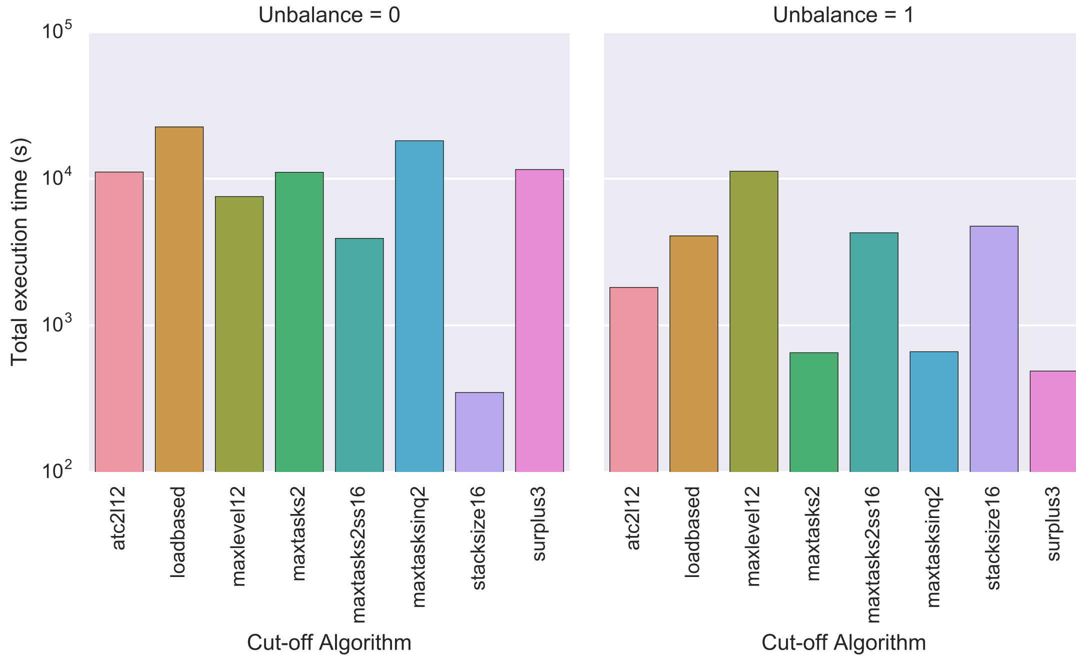

Figure 8.6: The misclassification time of programs by the Unbalance feature, on *server32*, considering the mean time of each program.

misclassification time of each cut-off algorithm aggregated by the different features. From the figures, it is possible to see the major differences in ForLoop and Unbalance features. ForLoop programs clearly have a lower misclassification cost when using MaxTasks as the granularity control algorithm. Non-loop programs have a lower penalty with the StackSize algorithm. Furthermore, unbalanced programs clearly perform better with StackSize, while Surplus has the lowest cost in unbalanced programs, followed by MaxTasks and MaxTasksInQueue.

Based on this information, it is possible to define the following rule for choosing a granularity:

$$\mathsf{cutoff} = \begin{cases} \mathsf{MaxTasks} & \mathsf{ForLoop} = 1 \\ \mathsf{Surplus} & \mathsf{Unbalanced} = 1 \\ \mathsf{StackSize} & \mathsf{otherwise} \end{cases}$$

Unbalanced and For-Loop tasks have the particularity of creating several tasks very quickly. Surplus and MaxTasks are approaches that perform well under those scenarios. StackSize is a conservative approach that performs well in stable scenarios.

8.2.3 A Machine Learning Approach

Selecting a cut-off approach based on program information is a classification problem. Machine Learning (ML) is frequently used to automatically classify instances based on collected data. Machine Learning is capable of understanding how combinations of features can impact the final classification. Given the difficulty of this problem, using ML classifiers can provide candidate solutions for the classification problem.

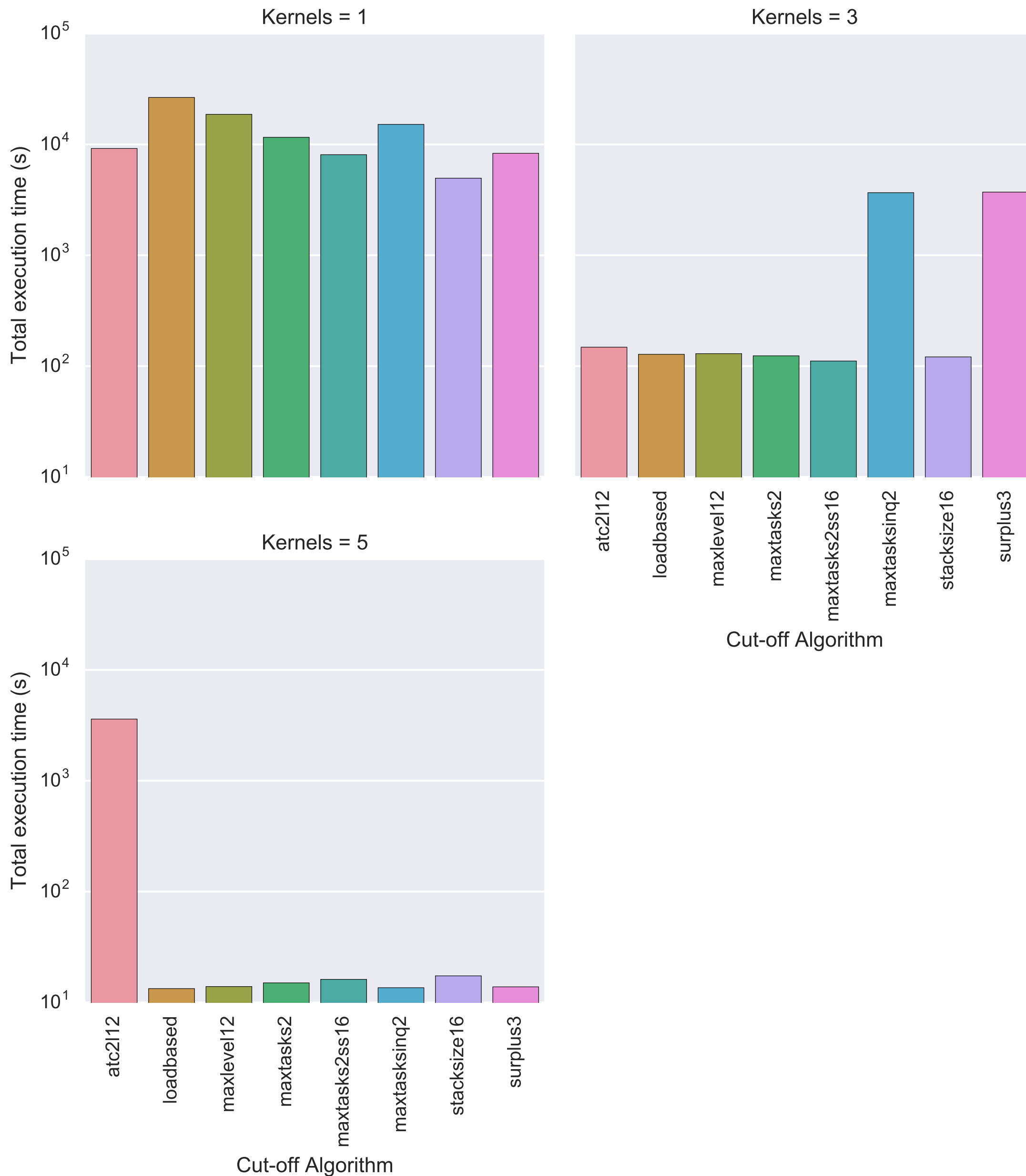

Figure 8.7: The misclassification time of programs by the number of kernels, on *server32*, considering the mean time of each program.

The proposed ML approach uses the same features identified as candidate features for the ruleset approach: *Forloop*, *Unbalance*, *Number of kernels*, *Branching Factor* and *Nesting*. These features are scaled between 0 to 1, as required by classifiers.

The same classifiers used in Chapter 4 have been evaluated: Random, Naïve Bayes, SVM, MLP, Decision Tree, Random Forests and Balanced Random Forests. The main difference is that this is a multi-class problem while the CPU-GPU decision only had two alternatives. For the Balanced Random Forests classifier, the instance weight in building the forests is the maximum misclassification of that instance.

The major shortcoming of this approach is the lack of data in this area. The benchmark used is the largest benchmark for cut-off algorithm evaluation. Even considering fork-join programs, the Fork-Join in the Wild study (De Wael et al.,

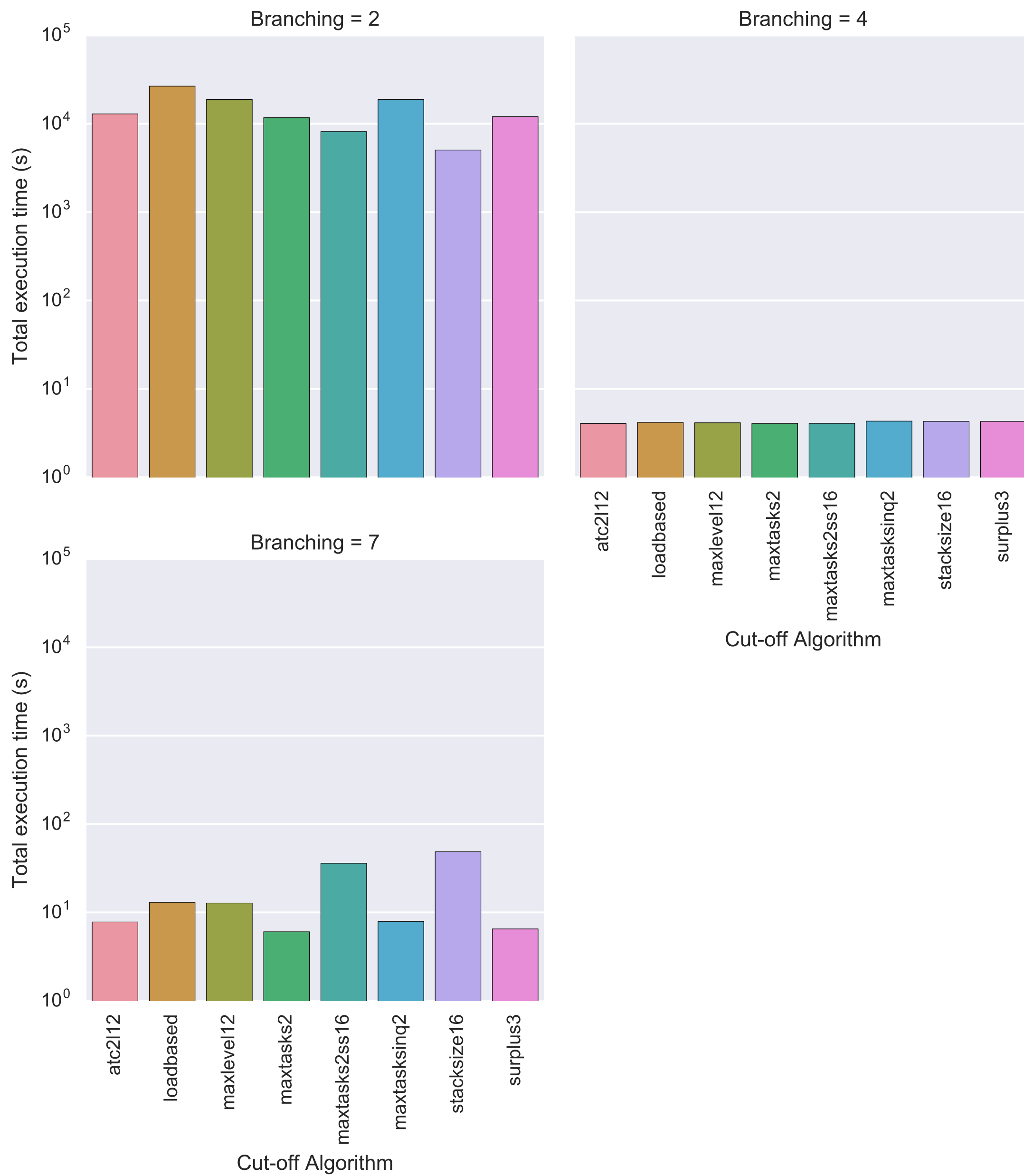

Figure 8.8: The misclassification time of programs by the branching factor, on *server32*, considering the mean time of each program.

2014) included 120 real-world programs, but most of them were simple homework exercises or had complex dependencies on large systems, in which the cut-off algorithm had no impact. Despite the limitation of a small benchmark suite, there is a high variety in that benchmark.

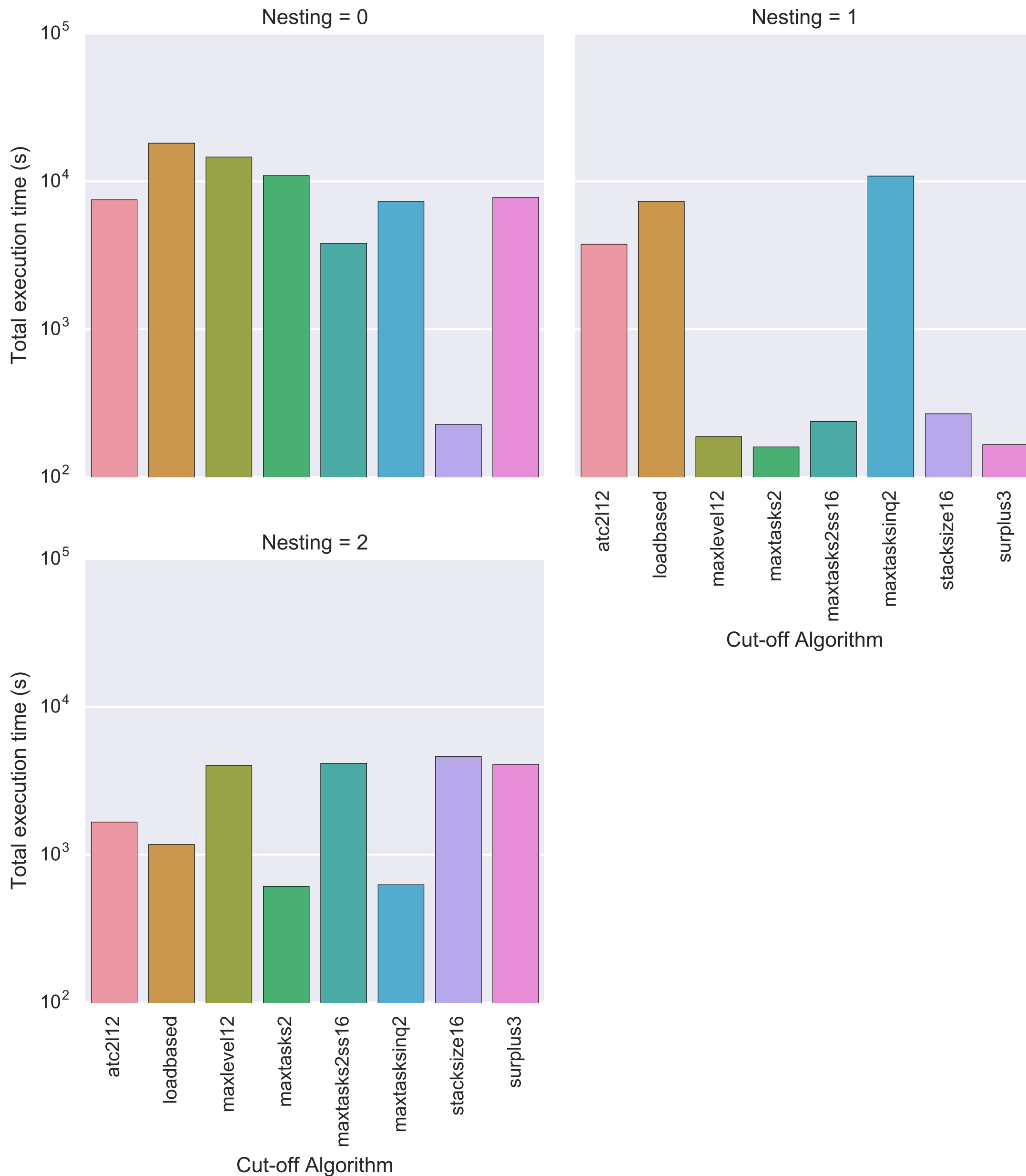

Figure 8.9: The misclassification time of programs by the number of nesting loops, on *server32*, considering the mean time of each program.

8.3 Evaluation

8.3.1 Methodology

Because the last two approaches are based on information learnt from existing data, it is necessary to define a new dataset of unseen data to evaluate both approaches. This allows to evaluate the generalization capabilities of data-based approaches.

The training set consisted in the benchmark suite used in Chapter 5 to evaluate cut-off algorithms, as it has already been used to develop the rule-based approach.

The testing data-set consisted in two synthetic programs that, according to their input parameters, simulate different types of programs, resulting in programs with different features. The first synthetic program generated recursive programs, either

balanced or unbalanced, with more or less work. This program is the same synthetic program used in Chapter 6 for the energy evaluation. The second synthetic program creates loop-based programs, balanced or unbalanced, with different leves of nesting, different number of kernels and more or less work. The synthetic dataset resulted in 2383 different programs executed with each of the cut-off algorithms considered. This dataset was retried from executing these two programs under the same conditions of the training dataset.

8.3.2 Results

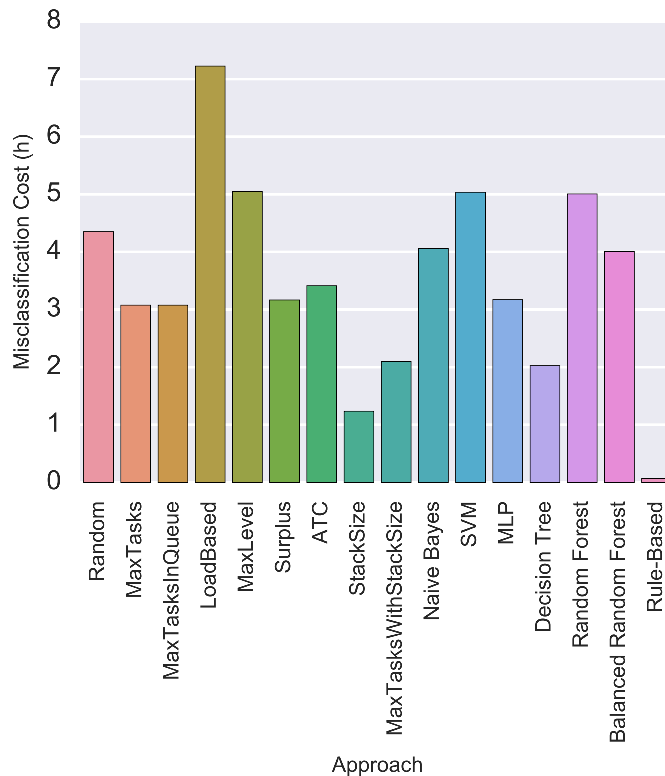

Figure 8.10: Misclassification cost of each decision mechanism over the training dataset, on *server32*.

Figure 8.10 shows the misclassification of each approach on the *server32* machine. All approaches were evaluated using a leave-one-out cross validation, and for non-deterministic classifiers, the mean misclassification cost of 30 executions was used. Of all the static approaches, the one which classified every program as StackSize was the best, reflecting that the StackSize had a lower misclassification cost, shown in Figure 8.2. Among ML classifiers, Decision Tree had the lowest misclassification score. A simple decision tree can perform better than a randomized forest by being less specific, resulting is less overfitting. The Balanced Random Forest classifier had a lower penalty than its non-balanced counterpart, showing that using weights can reduce misclassification cost. Despite its simplicity, the rule-based approach outperformed all other approaches.

Figure 8.11 shows the misclassification cost of each approach using energy consumption instead of execution time. Static stack-based approaches have a lower

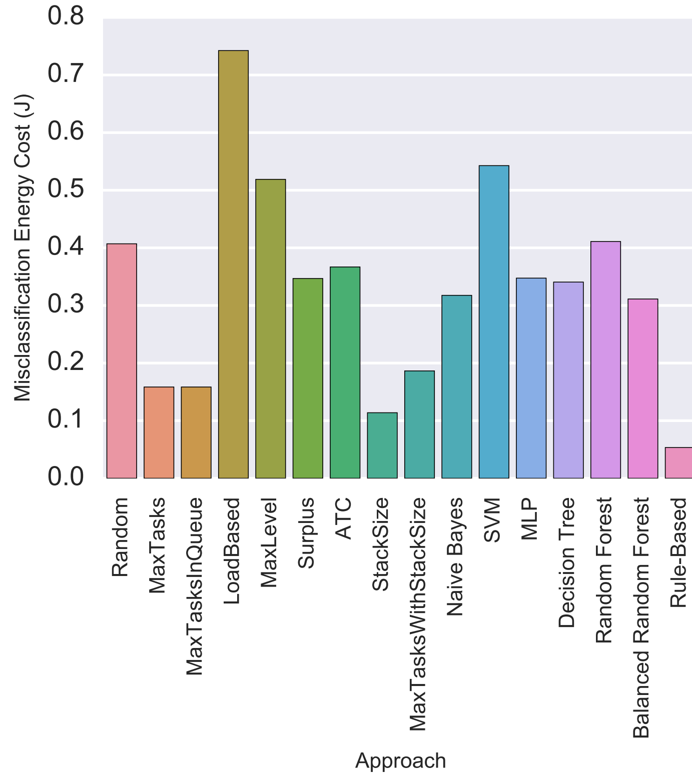

Figure 8.11: Misclassification energy cost of each decision mechanism over the training dataset, on *server32*.

consumption and the Rule-based approach is also able to achieve a lower misclassification energy cost than any other approach.

In order to evaluate the generalization capability of these approaches, a new evaluation was performed. Classifiers were fitted to the training dataset, and evaluated on the testing dataset over previously unknown programs. Figure 8.12 shows the misclassification error of the same approaches on this evaluation. The best static approach is LoadBased, followed by MaxLevel, the two worst static approaches on the training dataset. This result evidences the No Free Lunch Theorem for cut-off algorithms. Thus using a static approach based on either training or testing dataset would perform poorly on the other dataset.

The Ruleset and ML approaches were able to obtain a misclassification cost similar to the static StackSize approach. Balanced Random Forest also outperformed the Random Forest classifier, and even the predefined ruleset. The main reason for these results is that StackSize is the best cut-off algorithm in the training dataset, thus ruleset and ML classification are skewed towards that algorithm. In this new dataset, StackSize is the third best cut-off algorithm, showing that although it might not be the fastest in all programs, it is robust to new programs.

Figure 8.13 shows the misclassification cost in energy consumption for the testing benchmark, using energy consumption data instead of execution time. The amplitude differences among approaches are lower than when using execution time, resulting in a lower impact. Energy-wise, the rule-based approach is able to achieve energy consumption values similar to MaxLevel and Stacksize static approaches. Loadbased is still the best static approach. Both Naïve Bayes and MLP classifiers

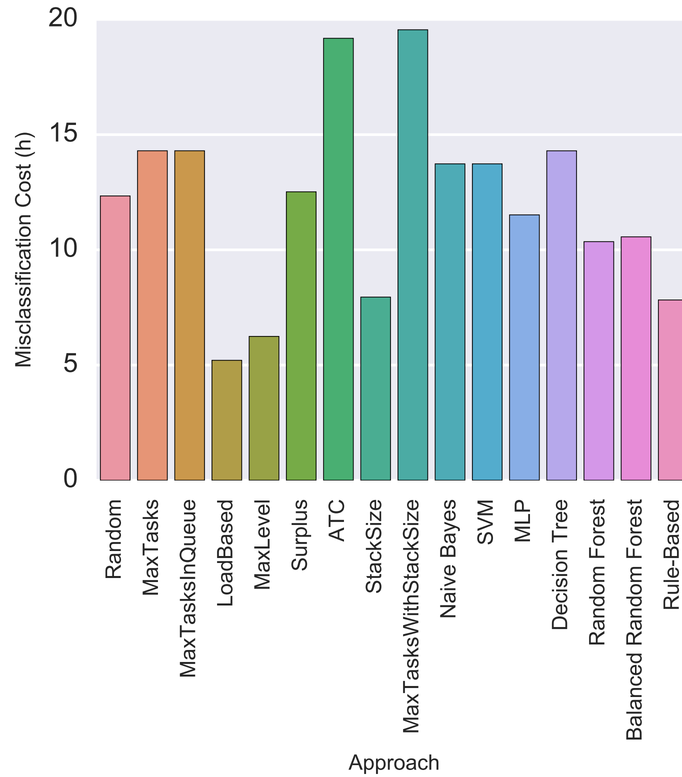

Figure 8.12: Misclassification cost of each decision mechanism on the testing dataset, on *server32*.

are able to obtain lower misclassification energy costs than other non-static approaches. Thus, these classifiers should be used when the values of each granularity algorithm are less distant.

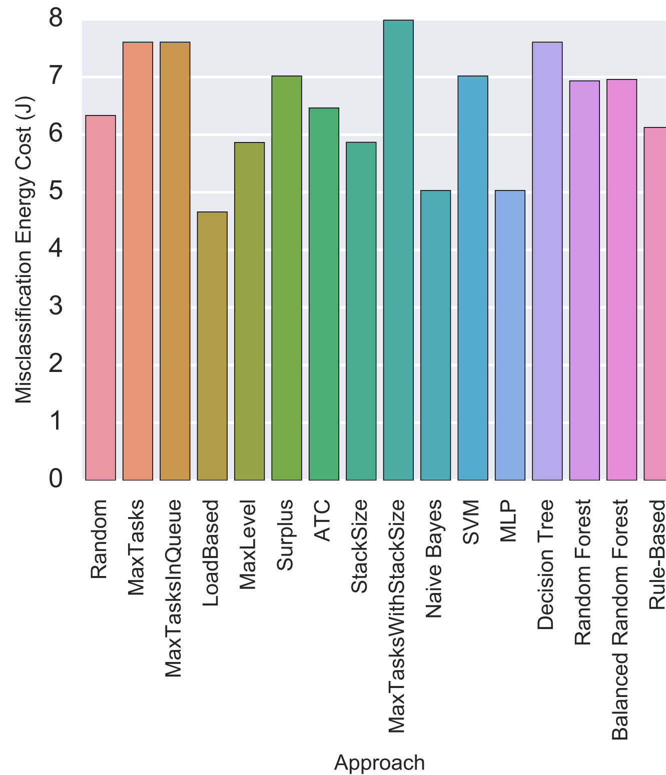

Figure 8.13: Misclassification energy cost of each decision mechanism on the testing dataset, on *server32*.

8.4 Conclusions

This Chapter presented and evaluated three approaches for selecting a cut-off algorithm for a given program, based on its features. The first approach, used as baseline, is the usage of a single cut-off algorithm for all programs. The second approach is the usage of a ruleset obtained from the misclassification analysis of each feature. Finally, the third approach consists in using ML classifiers to predict which cut-off algorithm to use.

The three approaches were evaluated in the same dataset previously analyzed, with the ruleset approach obtaining a lower misclassification cost than any other approach. This ruleset consists of three conditions which can be easily applied by programmers and compilers to perform the decision, instead of complex classifiers that require to be trained with pre-existing data.

The same approaches were evaluated on a synthetic testing benchmark, resulting in both Balanced Random Forests, trained with misclassification weights, and the Ruleset approach obtaining the best results from the dynamic approaches. The static approaches that had the best performance on the testing dataset were the ones with the worst results on the training dataset, revealing that they would not be good choices and confirming the No Free Lunch Theorem for cut-off algorithms. Energy-wise, ML classifiers obtained misclassification costs close to the best static approach for the testing benchmark.

Both developers and compiler and runtime systems now have a direct rule for deciding which granularity algorithm to use, that only requires one to know whether

the program uses for-loops and whether it is balanced or not.

Chapter 9

Conclusions and future work

This Chapter presents an overview of the thesis and discusses future work.

9.1 Overview

This thesis has presented an approach for automatic parallelization of parallel programs, and its optimization in regards to granularity control. Existing and proposed approaches for granularity management at runtime were extensively evaluated, as well as methods for selecting an algorithm for a given program.

In the field of parallel programming, automatic parallelization is a desirable goal because processors are increasing their core count instead of increasing the clock frequency, as they were before. Additionally, manually parallelizing a program is a large effort, which can be error prone. This thesis has presented a new automatic parallelization model based on access permissions, a representation of memory accesses to objects. From access permissions, AST nodes can be translated to a parallel version of themselves, with a focus on method invocations and loops where there is a higher gain in parallelization. Given the fine task granularity identified by this model, it is necessary to coarse granularity to generate efficient programs. This process can be done statically or dynamically. The presented approach has been applied to an existing language, Java, but can be applied to any procedural or object-oriented language. This model is an improvement over OoOJava (Jenista et al., 2011) which requires manual identification of possible tasks. Additionally, the proposed model does not rely on speculative execution by the runtime. This model is also an improvement over the concurrent by default language Æminium (Stork et al., 2014) because it does not require explicit access permissions from the programmer.

In the proposed automatic parallelization mode, as well as in Æminium, there was a over-parallelization of tasks, which would result in slower programs, compared to a manual parallelization process. In order to generate more efficient programs, a new hybrid granularity control algorithm model was introduced. This approach relies on static analysis performed during compilation to build a cost-model for each parallelization point. During execution the cost-model is specialized with concrete values from input data to make a prediction whether to execute the program in parallel or sequentially. This mechanism is an alternative to runtime-based approaches. It is also an improvement over Oracle (Acar et al., 2011), because it does not require

special annotations from the programmer. The cost-model methodology can also be applied in other parallel languages, like Æminium, or other automatic parallelization tools targeting sequential languages, like C.

The proposed automatic parallelization model is capable of targeting both CPUs and GPUs. In order to decide which processor should be used for each program, a Machine-Learning approach is proposed for deciding when to use the fine granularity of the GPU over the coarser granularity of multicore CPUs. The proposed model defines features that can be automatically or manually extracted from the program through static analysis or runtime inspection. The classifier that has shown better results was a Random Forest trained with the weights of the possible misclassification cost, resulting in over 95% of accuracy, and a low misclassification cost. Since this work was published, other works have confirmed the usage of the same or similar features for the same purpose, and also reached the conclusion that Artificial Neural Networks achieved better performance than Support Vector Machines.

Considering multicore CPUs, granularity control mechanisms are used to generate more parallel work when there are not enough tasks and to avoid scheduling overheads when there are enough. This thesis has introduced three new dynamic granularity control algorithms: MaxTasksInQueue, StackSize and an hybrid version of MaxTasks and StackSize. These approaches were evaluated against existing cut-off algorithms, concluding that they were able to improve the performance on some programs. The benchmark suite used is the largest that has been used for cut-off algorithm evaluation, and it has been made available for public use. This evaluation was extended to empirically verify the No Free Lunch Theorem application to cut-off algorithms. It was concluded that no cut-off algorithm is better than all others in all programs and, in fact, all algorithms have the same average performance over a large set of programs.

Apart from this evaluation that, like previous evaluations, only considered execution time, the energy impact of cut-off algorithms was also evaluated, concluding that different cut-offs have different energy efficiency. In balanced programs of light or heavy workloads, depth-based approaches such as MaxLevel or ATC perform better. In medium workloads MaxTasks is the best approach. In coarse irregular programs, MaxTasks, MaxTasksInQueue and Surplus have the best performance, but StackSize-based approaches are more suitable in highly irregular and lightweight programs. These conclusions were confirmed in a larger benchmark that included real-world and synthetic benchmarks. MaxTasks was considered the best approach in both energy and time performance, but StackSize was considered the more conservative approach to prevent programs from taking too much time.

Existing and proposed dynamic granularity control algorithms were defined by humans, based on runtime information from parallel programs. A Genetic Algorithm was developed to obtain a program configuration, which included a synthetic granularity control algorithm. The Genetic Algorithm was not able to successfully find a cut-off that would be better for any given program, confirming the No Free Lunch Theorem for cut-off algorithms. However, this approach proved to work in finding the best granularity control algorithm to improve the performance of a single program individually. 10 generations were considered enough to evolve into a stable solution.

Given the No Free Lunch Theorem for cut-off algorithms, in order to automatically obtain the best performance for a program, it is necessary to select the

granularity control algorithm more suitable for that specific program. Two new approaches have been presented: a ruleset created from the feature distribution of empirical results, and machine-learning classifiers applied to the same results. Within the real-world benchmark, the ruleset was able to achieve the best results. In a synthetic testing benchmark, the Random Forest classifier, trained with the maximum misclassification cost as weight per instance, was able to achieve better results than the ruleset.

Overall, using the proposed models, it is now possible for programmers and automatic parallelizing compilers to generate programs with the granularity control algorithms that result in the best performance.

9.2 Future Work

Automatic parallelization is still not able to always achieve better results than manual parallelization. The proposed automatic parallelization model can be improved in some areas. Firstly, IO operations are sequentialized in a conservative manner. It would be interesting to inspect those operations and identify whether they can be executed out-of-order or not. While there is work on transactional IO, it adds unwanted overhead to the system. Secondly, aliasing in Object-Oriented languages can be so common that it reduces the parallelism extracted by this model. Programming languages with explicit ownership, like Rust (Matsakis and Klock II, 2014), can make the application of this model more useful. Finally, exception handling is also sequentialized, and new programming language models for exception handling are required to improve automatic parallelization in those cases.

Another aspect left for improvement is the correct modeling of parallel memory allocation. Several of the programs in the benchmark spend a significant percentage of the time allocating memory, either domain-specific or task-related. In order to optimize the Cost-Model granularity control algorithm, it would be interesting to consider memory allocation.

Energy-wise, asymmetrical processors, like *big.Little*, are an interesting target to study. The energy-performance duality is different in this platform, which can lead to higher performance gains.

Finally, the evaluation of GPU-CPU decision and granularity control algorithm selection would be improved by using a larger number of real-world programs. While synthetic programs have been used to simulate different behaviors of programs, they are not guaranteed to have the same distribution as a real-world large benchmark.

Bibliography

Abdelrahman, T. S. and Huynh, S. (1996). Exploiting task-level parallelism using ptask. In *PDPTA*, volume 96, pages 252–263.

Acar, U. A., Blelloch, G. E., and Blumofe, R. D. (2000). The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM.

Acar, U. A., Charguéraud, A., and Rainey, M. (2011). Oracle scheduling: controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Notices*, volume 46, pages 499–518. ACM.

Acar, U. A., Chargueraud, A., and Rainey, M. (2013). Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Notices*, volume 48, pages 219–228. ACM.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr, G. L., Tobin-Hochstadt, S., Dias, J., Eastlund, C., et al. (2005). The fortress language specification. *Sun Microsystems*, 139:140.

Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J. O., Pasquier, F.-X., Péan, G., and Villalon, P. (2012). Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*.

Artigas, P. V., Gupta, M., Midkiff, S. P., and Moreira, J. E. (2000). Automatic loop transformations and parallelization for java. In *Proceedings of the 14th international conference on Supercomputing*, pages 1–10. ACM.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., and Zhang, G. (2007). A proposal for task parallelism in openmp. In *International Workshop on OpenMP*, pages 1–12. Springer.

Baldini, I., Fink, S. J., and Altman, E. (2014). Predicting gpu performance from cpu runs using machine learning. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 254–261. IEEE.

Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D. A., et al. (1993). Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Bik, A. J. and Gannon, D. B. (1997). Automatically exploiting implicit parallelism in java. *Concurrency - Practice and Experience*, 9(6):579–619.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996). Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69.

Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. (2008a). Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, pages 132–146. Springer.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008b). Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Cai, Q., González, J., Rakvic, R., Magklis, G., Chaparro, P., and González, A. (2008). Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 240–249. ACM.

Catanzaro, B., Garland, M., and Keutzer, K. (2011). Copperhead: Compiling an embedded data parallel language. *Principles and Practices of Parallel Programming (PPoPP)*, pages 47–56.

Cavazos, J. and Moss, J. E. B. (2004). Inducing heuristics to decide whether to schedule. In *ACM SIGPLAN Notices*, volume 39, pages 183–194. ACM.

Chafik, O. (2011a). Javacl opencl bindings for java. `https://github.com/nativelibs4java/JavaCL`. [Online; accessed 23-October-2011].

Chafik, O. (2011b). Scalacl. `http://code.google.com/p/scalacl/wiki/CLConvertibleLanguageSubset`. [Online; accessed 23-October-2011].

Chakravarty, M., Keller, G., Lee, S., McDonell, T., and Grover, V. (2011). Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM.

Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312.

Chan, B. and Abdelrahman, T. S. (2004). Run-time support for the automatic parallelization of java programs. *The Journal of Supercomputing*, 28(1):91–117.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Chen, C., Liaw, A., and Breiman, L. (2004). Using random forest to learn imbalanced data. *University of California, Berkeley*, 110.

Chen, M. K. and Olukotun, K. (2003). The jrpm system for dynamically parallelizing java programs. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 434–445. IEEE.

Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C., et al. (2007). Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM.

Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., et al. (2003). Reformulating software engineering as a search problem. *IEE Proceedings-software*, 150(3):161–175.

Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., and Wen, T. (2008). Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing*, pages 536–545. IEEE.

Cunningham, D., Bordawekar, R., and Saraswat, V. (2011). Gpu programming in a high level language: compiling x10 to cuda. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, page 8. ACM.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., and Midkiff, S. (2009). Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, (12):36–42.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE.

De Wael, M., Marr, S., and Van Cutsem, T. (2014). Fork/join parallelism in the wild: documenting patterns and anti-patterns in java programs using the fork/join framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 39–50. ACM.

DeJong, K. (1975). An analysis of the behavior of a class of genetic adaptive systems. *Ph. D. Thesis, University of Michigan.*

Dig, D. (2011). A refactoring approach to parallelism. *Software, IEEE*, 28(1):17–22.

Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., and Nieplocha, J. (2009). Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193.

Duran, A., Corbalán, J., and Ayguadé, E. (2008a). An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 36. IEEE Press.

Duran, A., Corbalán, J., and Ayguadé, E. (2008b). Evaluation of openmp task scheduling strategies. In *International Workshop on OpenMP*, pages 100–110. Springer.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguadé, E. (2009). Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *38th International Conference on Parallel Processing*, pages 124–131.

El-Ghazawi, T. and Smith, L. (2006). Upc: unified parallel c. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM.

Faxen, K.-F. (2010). Efficient work stealing for fine grained parallelism. In *2010 39th International Conference on Parallel Processing*, pages 313–322. IEEE.

Feautrier, P. (1996). Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103. Springer.

Fonseca, A. (2011). *AeminiumGPU–A CPU_GPU Hybrid Runtime for the Aeminium Language*. PhD thesis, University of Coimbra.

Fonseca, A. (2013). Æminium Benchmark Suite. https://github.com/AEminium/AeminiumBenchmarks. [Online; accessed 23-October-2013].

Fonseca, A. and Cabral, B. (2012). Handling exceptions in programs with hidden concurrency: New challenges for old solutions. In *Exception Handling (WEH), 2012 5th International Workshop on*, pages 14–17. IEEE.

Fonseca, A. and Cabral, B. (2013). Æminiumgpu: an intelligent framework for gpu programming. In *Facing the Multicore-Challenge III*, pages 96–107. Springer Berlin Heidelberg.

Fonseca, A. and Cabral, B. (2016a). Controlling the granularity of automatic parallel programs. *Journal of Computational Science*.

Fonseca, A. and Cabral, B. (2016b). Evaluation of runtime cut-off approaches for parallel programs. *VECPAR 2016 Proceedings*.

Fonseca, A., Cabral, B., Rafael, J., and Correia, I. (2016). Automatic parallelization: Executing sequential programs on a task-based parallel runtime. *International Journal of Parallel Programming*, pages 1–22.

Fonseca, A., Lourenço, N., and Cabral, B. (2017). Evolving cut-off mechanisms and other work-stealing parameters for parallel programs. In *European Conference on the Applications of Evolutionary Computation*. Springer. to appear in.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM.

Frost, G. (2011). Aparapi in amd developer website.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76.

Gerasoulis, A. and Yang, T. (1993). On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701.

Girkar, M. and Polychronopoulos, C. D. (1992). Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178.

Goldberg, D. E. and Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99.

Grewe, D. and O'Boyle, M. F. (2011). A static task partitioning approach for heterogeneous systems using opencl. In *International Conference on Compiler Construction*, pages 286–305. Springer.

Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., and Pouchet, L.-N. (2011). Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011.

Guo, Y., Barik, R., Raman, R., and Sarkar, V. (2009). Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE.

Guo, Y., Zhao, J., Cave, V., and Sarkar, V. (2010). Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.

Gupta, M., Mukhopadhyay, S., and Sinha, N. (2000). Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562.

Haghighat, M. R. and Polychronopoulos, C. D. (1993). Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 567–585. Springer.

Harris, M. (2010). Optimizing parallel reduction in cuda. http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html/. [Online; accessed 23-October-2013].

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. (2008). Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM.

Hogen, G., Kindler, A., and Loogen, R. (1992). Automatic parallelization of lazy functional programs. In *ESOP'92*, pages 254–268. Springer.

Hong, C., Chen, D., Chen, W., Zheng, W., and Lin, H. (2010). Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM.

Huang, X., Li, K., and Li, R. (2009). A energy efficient scheduling base on dynamic voltage and frequency scaling for multi-core embedded real-time system. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 137–145. Springer.

Jenista, J. C., Demsky, B. C., et al. (2011). Ooojava: software out-of-order execution. In *ACM SIGPLAN Notices*, volume 46, pages 57–68. ACM.

John, G. H. and Langley, P. (1995). Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc.

Joselli, M., Zamith, M., Clua, E., Montenegro, A., Conci, A., Leal-Toledo, R., Valente, L., Feijó, B., d'Ornellas, M., and Pozzer, C. (2008). Automatic dynamic task distribution between cpu and gpu for real-time systems. In *Computational Science and Engineering, 2008. CSE'08. 11th IEEE International Conference on*. Ieee.

Kerr, A., Diamos, G., and Yalamanchili, S. (2010). Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42. ACM.

Kimura, H., Sato, M., Hotta, Y., Boku, T., and Takahashi, D. (2006). Emprical study on reducing energy of parallel programs using slack reclamation by dvfs in a power-scalable high performance cluster. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kofler, K., Grasso, I., Cosenza, B., and Fahringer, T. (2013). An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 149–160. ACM.

Lea, D. (2000). A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM.

Leijen, D., Schulte, W., and Burckhardt, S. (2009). The design of a task parallel library. *Acm Sigplan Notices*, 44(10):227–242.

Leino, K., Poetzsch-Heffter, A., and Zhou, Y. (2002). Using data groups to specify and check side effects. *ACM SIGPLAN Notices*, 37(5):246–257.

Loh, W.-Y. (2011). Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23.

Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE.

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K., and Trinder, P. (2010). Seq no more: better strategies for parallel haskell. In *ACM Sigplan Notices*, volume 45, pages 91–102. ACM.

Marlow, S., Peyton Jones, S., and Singh, S. (2009). Runtime support for multicore haskell. In *ACM Sigplan Notices*, volume 44, pages 65–78. ACM.

Matsakis, N. D. and Klock II, F. S. (2014). The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM.

McKenney, P. E. (2011). Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*, page 7.

Mehrara, M., Hao, J., Hsu, P.-C., and Mahlke, S. (2009). Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices*, 44(6):166–176.

Miller, B. L. and Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212.

Mohr, E., Kranz, D. A., and Halstead, R. H. (1991). Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE transactions on Parallel and Distributed Systems*, 2(3):264–280.

Munshi, A. (2009). The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE.

Nascimento, A. P., Sena, A. C., Boeres, C., and Rebello, V. E. (2007). Distributed and dynamic self-scheduling of parallel mpi grid applications. *Concurrency and Computation: Practice and Experience*, 19(14):1955–1974.

Nichols, B., Buttlar, D., and Farrell, J. (1996). *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.".

Numrich, R. W. and Reid, J. (1998). Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM.

Nvidia (2007). Cuda programming guide.

Olivier, S. L. and Prins, J. F. (2009). Evaluating openmp 3.0 run time systems on unbalanced task graphs. In *International Workshop on OpenMP*, pages 63–78. Springer.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. (2015). Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.

Podobas, A., Brorsson, M., and Faxén, K.-F. (2010). A comparison of some recent task-based parallel programming models, in: 3rd workshop on programmability issues for multi-core computers, 2010.

Pottenger, B. and Eigenmann, R. (1995). Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM.

Rafael, J., Correia, I., Fonseca, A., and Cabral, B. (2014). Dependency-based automatic parallelization of java applications. In *European Conference on Parallel Processing*, pages 182–193. Springer International Publishing.

Rangasamy, A., Nagpal, R., and Srikant, Y. (2008). Compiler-directed frequency and voltage scaling for a multiple clock domain microarchitecture. In *Proceedings of the 5th conference on Computing frontiers*, pages 209–218. ACM.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".

Ribic, H. and Liu, Y. D. (2014). Energy-efficient work-stealing language runtimes. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 513–528. ACM.

Russell, T., Malik, A. M., Chase, M., and Van Beek, P. (2005). Learning basic block scheduling heuristics from optimal data. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 242–253. IBM Press.

Saad, M. M., Mohamedin, M., and Ravindran, B. (2012). Hydravm: extracting parallelism from legacy sequential code using stm. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*.

Seo, E., Jeong, J., Park, S., and Lee, J. (2008). Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1540–1552.

Shen, J., Varbanescu, A. L., and Sips, H. (2014a). Look before you leap: using the right hardware resources to accelerate applications. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pages 383–391. IEEE.

Shen, J., Varbanescu, A. L., Zou, P., Lu, Y., and Sips, H. (2014b). Improving performance by matching imbalanced workloads with heterogeneous platforms. In

Proceedings of the 28th ACM international conference on Supercomputing, pages 241–250. ACM.

Shun, J., Blelloch, G. E., Fineman, J. T., Gibbons, P. B., Kyrola, A., Simhadri, H. V., and Tangwongsan, K. (2012). Brief announcement: the problem based benchmark suite. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM.

Smith, L. A., Bull, J. M., and Obdrizalek, J. (2001). A parallel java grande benchmark suite. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 6–6. IEEE.

Sobral, J. L. and Proença, A. J. (1999). Dynamic grain-size adaptation on object oriented parallel programming. the scoopp approach. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 728–732. IEEE.

Software, S. (1997). Java on solaris 2.6. *White paper*.

Sridharan, S., Gupta, G., and Sohi, G. S. (2013). Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 337–348. ACM.

Steigerwald, B., Chabukswar, R., Krishnan, K., and Vega, J. (2008). Creating energy efficient software. *Intel white paper*.

Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73.

Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P., and Aldrich, J. (2014). Æminium: A permission-based concurrent-by-default programming language approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):2.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210.

Swaine, J., Tew, K., Dinda, P., Findler, R. B., and Flatt, M. (2010). Back to the futures: incremental parallelization of existing sequential runtime systems. In *ACM Sigplan Notices*, volume 45, pages 583–597. ACM.

Taura, K., Nakashima, J., Yokota, R., and Maruyama, N. (2012). A task parallel implementation of fast multipole methods. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 617–625. IEEE.

Tzannes, A., Caragea, G. C., Barua, R., and Vishkin, U. (2010). Lazy binary-splitting: a run-time adaptive work-stealing scheduler. *ACM Sigplan Notices*, 45(5):179–190.

van Dijk, T. and van de Pol, J. C. (2014). Lace: non-blocking split deque for work-stealing. In *European Conference on Parallel Processing*, pages 206–217. Springer.

Wagner, D. B. and Calder, B. G. (1993). Leapfrogging: a portable technique for implementing efficient futures. In *ACM SIGPLAN Notices*, volume 28, pages 208–217. ACM.

Wang, C., Li, X., Zhang, J., Zhou, X., and Nie, X. (2013a). Mp-tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(2):9.

Wang, Z. and O'Boyle, M. F. (2009). Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Sigplan notices*, volume 44, pages 75–84. ACM.

Wang, Z., Zheng, L., Chen, Q., and Guo, M. (2013b). Cap: co-scheduling based on asymptotic profiling in cpu+ gpu hybrid systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 107–114. ACM.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

Wolpert, D. H., Macready, W. G., et al. (1995). No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute.

Wu, Q., Juang, P., Martonosi, M., and Clark, D. W. (2005). Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *11th International Symposium on High-Performance Computer Architecture*, pages 178–189. IEEE.

Wu, T.-F., Lin, C.-J., and Weng, R. C. (2004). Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research*, 5(Aug):975–1005.

Yan, Y., Grossman, M., and Sarkar, V. (2009). Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *European Conference on Parallel Processing*, pages 887–899. Springer.

Zhao, J., Rogers, I., Kirkham, C., and Watson, I. (2005). Loop parallelisation for the jikes rvm. In *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pages 35–39. IEEE.