# Adaptive Parallelism Mapping in Dynamic Environments using Machine Learning

*Murali Krishna Emani*

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2014

# Abstract

Modern day hardware platforms are parallel and diverse, ranging from mobiles to data centers. Mainstream parallel applications execute in the same system competing for resources. This resource contention may lead to a drastic degradation in a program's performance. In addition, the execution environment composed of workloads and hardware resources, is dynamic and unpredictable. Efficient matching of program parallelism to machine parallelism under uncertainty is hard. The mapping policies that determine the optimal allocation of work to threads should anticipate these variations.

This thesis proposes solutions to the mapping of parallel programs in dynamic environments. It employs predictive modelling techniques to determine the best degree of parallelism. Firstly, this thesis proposes a machine learning-based model to determine the optimal thread number for a target program co-executing with varying workloads. For this purpose, this offline trained model uses static code features and dynamic runtime information as input.

Next, this thesis proposes a novel solution to monitor the proposed offline model and adjust its decisions in response to the environment changes. It develops a second predictive model for determining how the future environment should be, if the current thread prediction was optimal. Depending on how close this prediction was to the actual environment, the predicted thread numbers are adjusted.

Furthermore, considering the multitude of potential execution scenarios where no single policy is best suited in all cases, this work proposes an approach based on the idea of mixture of experts. It considers a number of offline experts or mapping policies, each specialized for a given scenario, and learns online the best expert that is optimal for the current execution. When evaluated on highly dynamic executions, these solutions are proven to surpass default, state-of-art adaptive and analytic approaches.

# Lay Summary

Modern computing landscape spans a wide variety of devices ranging from hand-held mobile devices to cloud and data centers. These systems are more powerful and hence the applications run faster owing to the presence of more number of processing units. However it is just not up to the hardware to improve programs. The software used in these applications should be able to leverage the massive powerful hardware resources to the maximum extent possible. On the other hand, a wide variety of programs such as graphic-intensive games, high performance computing, web-search, etc. execute on these parallel systems apart from day-to-day software applications. The resource requirements vary with the nature of the programs and desired goals. For example a web search using Google is expected to return the results instantly. Batch processing jobs are expected to process maximum records in a given time duration. Moreover, the entire computing platform is prone to dynamic changes that may negatively affect the programs. When many applications try to execute at the same time in the same system, it usually results in a contention for system resources. The programs are able to execute fast only if this contention is minimized.

This thesis aims to determine how many resources such as processing units are required for any parallel application so that it is able to use the hardware to its best potential and execute faster. There is no straightforward technique to determine this value. This work uses approaches based on machine learning where a model observes what applications are executing in the system along with the platform specific details and then determines the best resource requirements for the application.

# Acknowledgements

First and foremost, I thank my parents Krishna Rao Emani and Ramalakshmi Emani for their love and respect to my choices in life. Their constant encouragement and motivation inspired me to work hard and progress, no matter what the circumstances where. They are always there with me at all times.

I express my deep gratitude to my *guru* and advisor Professor Michael O'Boyle. As I see him, he is more than a supervisor: a mentor, leader, manager and most importantly, a cheerful person. His positivity, enthusiasm and charisma are inspiring. I just cannot thank him enough in words. He will be one of the most influential persons in my life.

I sincerely thank Dr. Björn Franke who was always there to guide me, and review my work. I am indebted to have Zheng Wang as my mentor during the initial stages of PhD. His collaboration is very helpful, resulting in parts of this thesis. I would like to thank Murray Cole and Vijay Nagarajan for the invaluable support they gave me during my PhD. I am fortunate to have Karthik as a friend, brother and guide. He was always there for me whenever I needed moral support. I cannot forget the memories of our trip to Isle of Skye.

I also would like to thank all my friends in the lab for their support throughout my stay. In no particular order, I thank Dominik, Kiran, Andrew, Vasileios, Konstantina, George, Praveen, Stan, Ursula, William. I am also grateful to all those who reviewed this thesis with constructive comments and feedback. I am happy to have Sunil Bhatta, Srinivas as my friends in Edinburgh who made my stay enjoyable.

I am also grateful to my brothers Ravi Kanth and Krishna Kumar who stood as strong pillars of support always. Above all, I would like to thank my wife Meenu, for making my life complete with her love and understanding. Her constant support and encouragement helped me to focus on pursuing my goals.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. I confirm that appropriate credit has been given within the thesis where reference has been made to the work of others. Some of the material used in this thesis has been published in the following papers:

- Murali Krishna Emani, Zheng Wang and Michael O'Boyle, "Smart, Adaptive Mapping of Parallelism in the Presence of External Workload", In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Februrary 2013, in *Chapter 4*.

- Murali Krishna Emani and Michael O'Boyle, "Change Detection-based Parallelism Mapping: Exploiting Offline Models and Online Adaptation", In $27^{th}$ *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2014, in *Chapter 5*.

- Murali Krishna Emani and Michael O'Boyle, "Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments", In $36^{th}$ *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2015, in *Chapter 6*.

(*Murali Krishna Emani*)

Dedicated to my parents and my advisor.

# Contents

# Chapter 1

# Introduction

Multicore-based parallel systems now dominate the computing landscape from hand-held mobile devices to cloud and data centers. A wide variety of programs such as graphic-intensive games, high performance computing, web-search, etc. execute on these parallel systems. The resource requirements vary with the nature of the programs and desired goals. However, one common characteristic across the wide spectrum of programs is often *parallelism*. Efficient matching of program parallelism to machine parallelism is critical. Over-provisioning the parallel work with excessive threads has a potential pitfall, in that the threads may have to spend a long time in wait queues to acquire the required resources. Under-provisioning may lead to poor system utilization as resources may be left idle with no thread actively using them. Hence, any mapping policy should try to avoid either of these, by determining the best thread number.

Real world systems are highly dynamic and unpredictable. These are often shared amongst programs competing for system resources. The dynamic parameters include program input data, hardware, software, external workloads, network contention, etc. For example, Figure 1.1 shows the workload behaviour in a real world high performance computing system [ECDF], where wide-variety of programs compete for system resources. In such dynamic conditions the complexity of mapping increases. Even if exclusive resources are allocated, effective system utilization is often compromised. Hence, to utilize the system most effectively, programs need to be co-located. The bottleneck here is an intense pressure for the system resources. This complexity is compounded by frequent changes in the system load due to dynamic workloads.

Dynamic allocation or mapping of parallel work to hardware is an extensively studied area where the policies can roughly be categorised as *offline* or *online*. Offline ap-

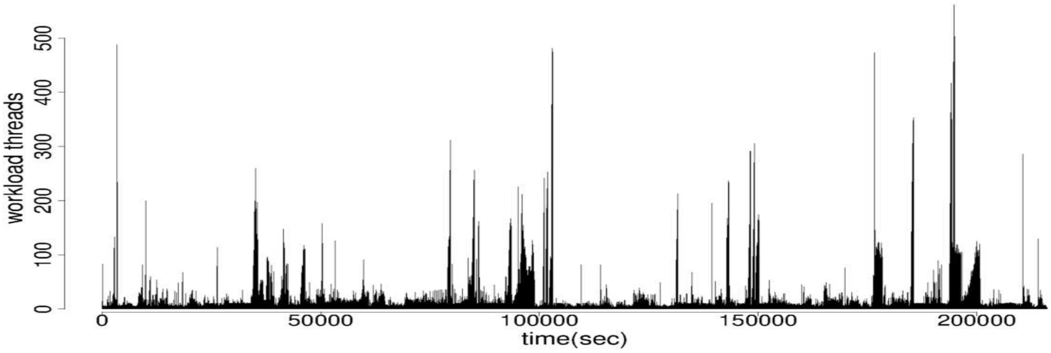


Figure 1.1: Highly dynamic system activity observed in a live system, showing number of threads vs. time.

proaches try to exploit as much prior knowledge about the system ahead of time as possible. Most compiler approaches fall into this category where program structure and machine characteristics are analysed to determine the best mapping of the program. Robust compiler technology to exploit the potential of multi-cores is still lacking despite the significant research contributions in the areas of automatic parallelization, mapping, scheduling and memory hierarchy optimization. The strength of the offline approaches is that a considerable effort can be expended into making a decision as this has no runtime overhead. However, they also typically make simplifying assumptions about resource availability and external workloads, often assuming a static and isolated environment. Thus, while these techniques are often program *specific*, they are often *fragile* in the presence of change.

Online approaches developed mostly as pure runtime systems, directly tackle this problem. They typically monitor the load on the system and adjust the mapping of the target program to fit the available resources. This often comes at the expense of not exploiting sufficient knowledge of a particular program to be mapped and hence using undifferentiated policies. Thus, while they are *robust*, they are often *generic* and may not deliver optimal performance compared with a program-specific approach.

**Predictive Modelling in Adaptive Mapping:** Machine learning techniques are promising in developing optimizing compilers and parallelism mapping. These encouraging outcomes highlight the use of machine learning models where there is no straightforward mapping mechanism. If the entire experimental set-up including the benchmarks, workloads, hardware along with optimal configurations, are known beforehand then mapping is relatively straightforward. However, in a realistic scenario, the chances of having exhaustive knowledge of the execution environment *a priori*, is highly unlikely. This makes the process of making the mapping decisions complex. Machine learning models are a good fit here, where a learnt model can easily be ported across different platforms and used by various unseen benchmark programs.

Manual mapping by expert programmers can result in effective implementations. However, this process is highly error-prone and expensive. There is also a lack of standard mechanisms to port such implementations to different hardware. Machine learning-based approaches, despite incurring an overhead in terms of offline training, approach the performance of manual tuning. Moreover, they are easily portable across multiple programs and hardware.

There are many tunable parameters that may influence a parallel program execution like thread placement policies, thread count, active or passive synchronization in the program, varying the frequencies of processors, etc. Optimal tuning of all these parameters is ideally the best approach for maximizing the mapping efficiency. The work in this thesis focuses on tuning one parameter, the degree of parallelism in terms of the number of software threads. The exact placement of these threads and scheduling mechanisms are left to the default operating system. This design choice is aimed at a higher level to OS scheduling which relieves the overhead in the scheduler in dealing with redundant threads which do not help in efficient program execution. However, optimizing the parallelism degree may be further combined with other parameters. For example, when the optimal number of threads is combined with affinity scheduling (see Section 6.6.7), the performance is improved further.

## 1.1 Problem

As discussed above, there are several issues involved in efficient parallelism mapping. The problems addressed in this thesis are listed here.

**Issue 1**: *Parallelism mapping with co-executing workloads is non-trivial.*

Program co-execution results in intense contention for resources such as processors, memory, bandwidth etc. When multiple multi-threaded programs start executing, the total number of software threads is often greater than the number of hardware threads. This over-subscription of resources results in increased queue length as threads wait for resources, which significantly increases the execution time. As the workloads vary in nature, determining the ideal number of threads is non-trivial. Existing policies

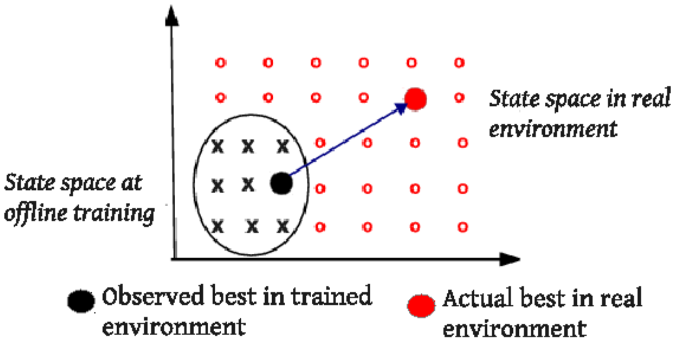

State space in real environment
State space at offline training
● Observed best in trained environment
● Actual best in real environment

Figure 1.2: State space: Offline models are trained only on partially observed states. In reality, state space can expand exhaustively. Hence, a model needs to learn continuously to adapt to new states.

assume isolated and static execution environments. During program execution, if the system is shared by varying workloads, these policies are highly unlikely to determine best mapping. Moreover, they exert pressure on external workloads, when trying to optimize a target program. This contention penalizes the workloads by assigning more resources to the target, with no due allocation to workloads. A simple heuristic to allocate free cores to the target after the workloads have been allocated appropirate number of cores was not optimal in atleast one-third cases observed in a limited study.

An ideal approach should determine the best mapping in terms of optimal thread count when a program co-executes with varying workloads. It should consider the runtime state along with program characteristics, and adapt to changing circumstances.

**Issue 2**: *Low-cost runtime monitoring and adaptation is challenging.*
Mapping policies are efficient when the training set used to build the policy is similar to the actual execution environment. Such approaches depend critically on the coverage of the training set. However, if the online environment departs significantly from expectations, there is no way of identifying that the system state is now radically different. This change invalidates the policy that is out-of-date and incorrectly determines the mapping. There is no "system utilization gauge" to monitor its accuracy online. Furthermore, the program continues executing not realizing that the policy is wrong and no longer valid. This results in a loss of performance in a new environment. Building a new policy from scratch during program execution may not be effective. It depends on how quickly the policy can reach an acceptable state of accuracy.

What is required here is an approach to exploit an existing offline model with continuous monitoring of its performance. As seen from Figure 1.2, once a significant change in the system is detected, the mapping need to be adjusted accordingly.

**Issue 3**: *Lack of mapping methods that fit all scenarios.*
Approaches to determine the best thread selection, in general, are characterised by a one-size-fits-all assumption. Such policies employ a single model that match a program to its parallel environment. There are no accurate ways to examine if this policy fits the current execution setting. Also, there is no mechanism to detect at runtime if any other policy may be more efficient at mapping. Even policies that have tunable parameters are highly unlikely to map effectively for future computing systems. Another critical issue is the difficulty in updating or extending existing models. Including additional expertise requires expensive retraining of the policy in all environments, which is highly infeasible. Another hurdle is a lack of effective ways to detect, at runtime, if a mapping policy is good, as the environment might have changed. Evaluating all policies during program execution is prohibitively expensive.

Here, a desirable approach should consider a set of policies, each fine-tuned to a specific execution state. It needs to intelligently choose the best policy corresponding to the current scenario. It also should provide ways to gracefully add additional expertise with minimal overhead.

## 1.2 Contributions

This thesis presents solutions to address the problems mentioned above. It aims to optimize the execution time of target programs by efficient *parallelism mapping*, i.e. determining how many threads to allocate to each parallel loop of the target program. Predictive models based on machine learning techniques are at the heart of these solutions.

Firstly, this thesis presents a smart and adaptive approach for mapping programs co-executing with dynamic workloads. The proposed approach uses a lightweight model, '*thread-predictor*', that predicts the ideal thread number. This greatly reduces the load in the system caused by multiple co-executing programs. This portable heuristic requires no additional retraining for new programs. It greatly improves program performance over state-of-art adaptive schemes when mapping a new program. An additional advantage of using this approach is not penalizing the co-executing workloads. Chapter 4 explains this technique in detail.

Next, this work considers scenarios when the execution environment changes drastically at runtime. The thread-predictor model trained in a sub-space of exhaustive state-space may turn sub-optimal in a new environment. Here the proposed solution, *CDMapp* (Change Detection based Parallelism Mapping), employs predictive modelling again. It develops a second model '*environment-predictor*', that predicts what the environment should look like assuming the thread-predictor was optimal. It then compares the predicted and actual environments at next decision stage, which serves as '*change_detection*'. This relative gap is used as the feedback to judge the quality of the thread-predictor and adjust the decisions i.e., new thread number, if necessary. This approach exploits existing offline model and adapts it online, and is discussed in Chapter 5.

Further, the next contribution is a novel approach to select the best mapping policy (*expert*) at runtime based on a "*mixture of experts*" approach. As no single policy is best suited for all dynamic environments, it is intuitive to collect a mixture of offline experts and, at runtime, use the expert that is most likely to be optimal in that instance. Any expert that is built with two mechanisms: thread-predictor and environment-predictor, can be included in the mixture, allowing for graceful inclusion of additional experts with no overhead. The expert selector uses the difference between the actual and predicted environments of each expert to choose the one that is most accurate. As these models are built with the same training data, the respective prediction accuracies are highly correlated. Chapter 6 presents this approach.

Finally, this thesis presents a detailed analysis of competitive co-scheduling using multiple mapping policies. It discusses in detail the impact of the program behaviour and the mapping policy on its execution. Exhaustive evaluations are performed on a per-program basis and shared-policy basis. Chapter 7 discusses this analysis and presents these findings.

## 1.3 Thesis Organisation

This thesis is organized into the following chapters.

**Chapter 2** introduces the basics of multi-core systems and widely used parallel programming models. It then describes machine learning techniques and how they are applied in parallelism mapping. Later it details a Markov decision process that can be used to describe mapping formally. Mixture of experts concepts are discussed later. This chapter ends with the evaluation techniques employed throughout this thesis.

**Chapter 3** lists the literature related to several areas covered by this work. It discusses highly regarded prior research classified broadly into parallelism mapping under isolation, program co-scheduling and online adaptation.

**Chapter 4** presents a predictive modelling-based approach for parallelism mapping. It shows how static code features can be used along with dynamic runtime features for optimal thread number prediction. The approach is then evaluated on different benchmark programs with highly dynamic co-executing workloads. This chapter is based on the work published in [Emani et al. 2013].

**Chapter 5** presents *CDMapp*, a novel technique for exploiting offline models and adapting them online based on changing executing conditions. It exploits an offline thread-predictor and then employs a novel environment-predictor to measure the efficiency of the thread-predictor. It detects changes in the executing system and adjusts the mapping on-the-fly, if required. This approach is evaluated on a variety of parallel benchmark programs with dynamic workloads and hardware resources. This chapter is based on the work published in [Emani and O'Boyle 2014].

**Chapter 6** describes an approach that selects the best expert from a mixture to suit to the executing environment. Each expert is a specialized mapping policy that is best suited for certain environments. The mixture of experts approach learns the best expert based on how accurately each expert predicts the forthcoming environment online. It then uses the mapping policy of the selected expert. This chapter is based on the work published in [Emani and O'Boyle 2015].

**Chapter 7** describes a detailed analysis of the impact of different scheduling policies on programs when the co-executing programs are self-adaptive. It analyzes the system state when each program adapts using its scheduling policy. It demonstrates that optimal mapping policies vary with the nature of the program and the execution scenarios.

**Chapter 8** concludes this thesis with the main findings and a summary of contributions. It then discusses challenges and pitfalls of these solutions followed by a description of potential future work ideas.

Appendix A lists the selected benchmark programs used for evaluating the work in this thesis.

# Chapter 2

# Background

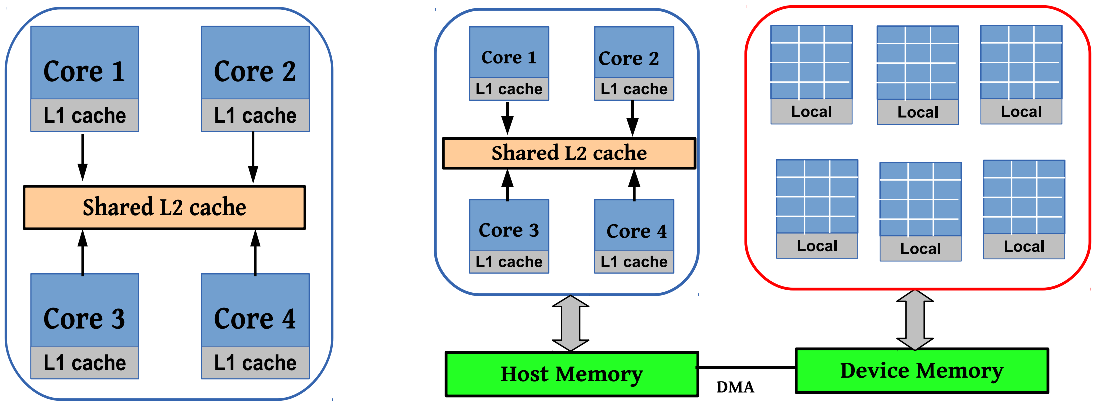
This chapter presents the necessary technical background for this work. It starts with an introduction to multi-core systems in Section 2.1. Next, different parallel programming models are discussed in Section 2.2, with an emphasis on OpenMP, that is primarily used in this thesis. Machine learning concepts and terminologies are presented in Section 2.3, followed by a description of Markov decision processes in Section 2.4 and Mixture of experts in Section 2.5. Later, the evaluation methodology is explained in Section 2.6. Section 2.7 summarizes this chapter.

## 2.1 Multi-core Systems

A multi-core processor is a single computing entity that is composed of multiple real processing units for enhanced performance. This is implemented by placing multiple cores in a single physical package. The cores may be tightly or loosely coupled based on design specifications. The advantage of multi-core systems lies primarily in the parallel execution of threads independently on multiple cores, which helps the applications run faster compared to a system with a single core.

In modern day computing systems, multi-cores have become an integral component. For example, they may consist of two cores (Intel Core 2 Duo, AMD Phenom II X2, Snapdragon 400), four cores (Intel core i5, i7, AMD Phenom II X4, Snapdragon 800) or the latest eight core Intel Core i7-5960x and Snapdragon 615. Two classes of memory types are: (a) shared, where all threads share common memory space that enables faster data exchanges and (b) distributed, in which each core has a small local memory where threads need to communicate with other processors.

Multicore systems are typically homogeneous or heterogeneous. **Homogeneous**



(a) Homogeneous system

(b) Heterogeneous system

Figure 2.1: (a) Homogeneous multi-core with four identical cores (b) Heterogeneous multi-core with a mix of processors of different architectures.

**multi-cores:** These consist of multiple identical cores that have the same processing power, frequencies and cache sizes. The same instruction set is used across all cores. Since each core contains the same hardware these are easy to produce. A 4-core system with private $L_1$ caches and shared $L_2$ cache is shown in Figure 2.1(a).

**Heterogeneous multi-cores:** These contain multiple non-identical processors of different capabilities and architectures. Two types of heterogeneous cores are widely used: (i) multiple general purpose CPUs that differ in operating frequencies, computing capabilities and power efficiencies: for example, ARM's big.LITTLE architecture (ii) a blend of general purpose and special purpose processors like Graphic Processing Units (GPUs). Such systems are more suited for applications that have different resource requirements during execution and those which contain sections of code that can execute more efficiently on the specialized cores. Figure 2.1(b) shows a heterogeneous system comprised of a mixture of general and specialized processors. The work in this thesis targets different homogeneous shared memory multi-core processors belonging to Intel Xeon family [Intel], however, this work makes no assumptions based on a specific type of multi-core architecture. That is, it may be replicated on alternate homogeneous multi-core platforms.

How the potential of multi-cores is realized depends primarily on the algorithm and its implementation. Embarrassingly parallel problems can achieve speedup factors equal to the number of cores assuming the problem is divided evenly enough to fit in the caches of each core. However, most applications need to be re-factored and carefully mapped to gain significant speedups on multi-cores. Developing applications that can fully leverage the potential of multi-core systems is non-obvious [Hennessy and Patterson 2003]. The next section describes multiple parallel programming models that attempt to enable an application to make the best use of the underlying multi-cores.

## 2.2 Parallel Programming Models

Parallel programming allows computation to be carried out simultaneously, and in parallel, by using *multi-threading*. The computable tasks are split into multiple chunks, where each thread performs the computation on its assigned chunk of code, independent of others. Few widely used models are `OpenMP` [Dagum and Menon 1998], `MPI` [Pacheco 1996], `Pthreads` [Nichols et al. 1996] for shared and distributed memory systems, `OpenCL` [Stone et al. 2010] and `CUDA` [Nickolls et al. 2008] for heterogeneous many-core systems. Other models include `Chapel` [Chamberlain et al. 2007], `StreamIt` [Thies et al. 2002] and `X10` [Charles et al. 2005].

The work presented throughout this thesis is implemented for C programs using OpenMP, which is one of the widely used parallel programming models. This model allows a finer control on how the parallel work is partitioned among parallel threads using simple constructs. This work could equally be applied to any parallel programming models which allow changes to thread numbers at runtime.

### 2.2.1 OpenMP

OpenMP [OpenMP] is a programming model that is widely used for data-parallel programs. This portable, scalable model provides programmers with an easy to use interface for building parallel applications.

More specifically, it is a collection of APIs that supports multi-platform shared memory multiprocessor programming. These APIs are available to use in C, C++ and Fortran programming languages. It consists of a set of compiler directives, libraries and environment variables. In C/C++, sections of code that are executed in parallel are declared using '*#pragma*' directive. When the pragma '*omp parallel*' is invoked a *master* thread with thread ID 0 forks additional *slave* threads to compute the work mentioned in the construct. These slave threads then run concurrently sharing the task and the runtime environment allocates these threads to different processors. When the parallelized code finishes executing, the slave threads join to the master thread which continues further until the end of the program.

For example, a parallel execution of independent iterations in a loop is declared as '*#pragma omp for*' just before the loop header. It assigns a set of worker threads to execute these chunks of iterations in parallel. This is known as loop or data parallelism [Grama et al. 2003]. Listing 2.1 shows such an example.

OpenMP is composed of several constructs for thread creation, work-sharing, synchronization of threads, data environment management, user-level routines and environment variables. By default, each slave thread executes a parallelized code independently. By using several work-sharing constructs, a task can be divided amongst different threads which enables both data and task parallelism to be realized.

Assigning threads to processors is managed by the runtime system based on several factors such as CPU usage and machine load. The number of threads can also be assigned by environment variables or within the code. Environment variables are used to change the parameters of an OpenMP program at runtime such as the number of threads or controlling loop iterations. For example 'OMP_NUM_THREADS' is used to specify the number of threads for an application as shown in Listing 2.2. The number of threads can be assigned to each parallel section independent of others using a user-level routine '*num_threads*'. This is shown in Listing 2.3. Throughout this thesis, the thread number for any parallel loop is changed using this clause.

For a data parallel loop that is computed by a number of worker threads, there are four types of scheduling policies available in OpenMP:

- *Static*: The loop iterations are evenly divided into chunks according to the number of work threads. Each worker thread is assigned a separate chunk.

- *Cyclic*: Each of the loop iterations is assigned to a worker thread in a round-robin fashion.

- *Dynamic*: The loop iterations are divided into a number of chunks with a small size. Chunks are dynamically assigned to worker threads on a first-come, first-served basis as threads become available.

- *Guided*: The loop iterations are divided into chunks where the size of each successive chunk is exponentially decreasing until the default minimum chunk size, 1, is reached.

```
#include<omp.h>
.........
#pragma omp parallel for
for (int i=0; i<1000; i++)
        C[i] = A[i] + B[i];

$ gcc –o omp_example –fopenmp omp_example.c
$ ./omp_example
```

Listing 2.1: A data parallel loop with OpenMP pragma.

```
#include<omp.h>
#pragma omp parallel for
for (int i=0; i<1000; i++)
        C[i] = A[i] + B[i];

$ gcc –o omp_example –fopenmp omp_example.c
$ export  OMP_NUM_THREADS=4
$ ./omp_example
```

Listing 2.2: Use of environment variables to set thread number at runtime.

```
#pragma omp parallel for num_threads(4)
for (int i=0; i<1000; i++)
        C[i] = A[i] + B[i];

$ gcc –o omp_example –fopenmp omp_example.c
$ ./omp_example
```

Listing 2.3: Use of num_threads() to modify thread number for a parallel loop.

### 2.2.2 Pthreads

Pthreads [Nichols et al. 1996], [Lewis and Berg 1998] is a POSIX standard API for creating and manipulating threads. It consists of C-style types and procedures. Unlike processes that often are costly to create and manage, a thread can be created with less overhead by the operating system. Moreover, thread management requires fewer resources than process management. Multiple threads that constitute a process share the same address space and hence have effective inter-thread communication. This is a low level API compared to OpenMP; hence it requires fine-grained thread management (create, join, fork).

### 2.2.3 OpenCL

Parallel programming on heterogeneous systems is facilitated by the Open Computing Language (OpenCL) [Stone et al. 2010]. It targets massive data parallelism that can be obtained on the graphical processing units (GPUs). OpenCL is a framework containing C-like functions called *kernels*. Here a program is comprised of a host and target code: the former dealing with input, output and data initialization, and handling the data transfers between main memory and GPU memory. The target code expressed in terms of kernels describes the data-parallel task to be executed on the GPU.

## 2.3 Machine Learning

Machine learning, [Mitchell 1997], [Bishop 2006], techniques give computing systems the ability to learn without the need of explicit programming. They are developed by learning models from input data sets obtained from training experiments.

Two major approaches to build machine learning models are (1) supervised learning, for labelled data and (b) unsupervised learning for unlabelled data. In this thesis, supervised learning is used for learning the models and hence, this approach is explained in detail.

### 2.3.1 Supervised Learning

Supervised learning is a method of inferring a function from training data that is labelled. Each constituent item in the training data referred as a training example, consists of an input set of values and a preferred output predicted value. This is termed *labelling*, where the desired output is known for a given input data set. A supervised learning algorithm works on the examples in the training data. It analyzes the inputs and generates an inferred function. This function now can be used to correctly estimate the output value for a new unseen input instance.

Each training example is in the form of an ordered pair *(a,b)* where *a* is a *feature vector* consisting of numerical values for each corresponding input *feature*. The task of

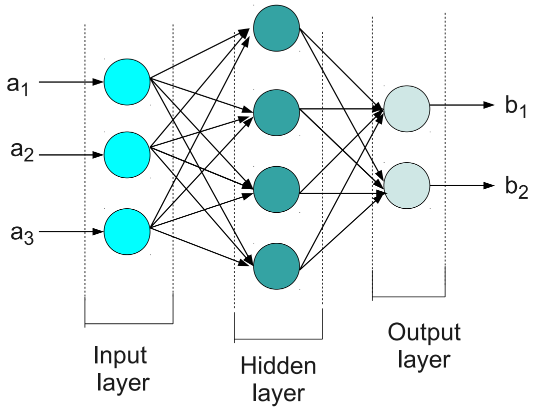


Figure 2.2: A feedforward Artificial neural network with three layers: input, hidden and output layer.

this learning method is to generate a function $f$ to map input vectors to output values:

$$f : a \rightarrow b$$

This section describes the supervised learning techniques, artificial neural networks for classification and simple linear regression, used in this thesis.

### Artificial Neural Networks

Artificial neural networks (ANNs) [Bishop 1995] inspired by biological learning systems provide a robust approach for modelling complex systems. They provide a mechanism to approximate real-valued functions. An ANN is built from a densely interconnected set of simple units called *nodes* or *neurons*. Each such unit takes a number of real-valued inputs and produces a single real-valued output. These nodes are spread in different layers. Each connection of these nodes are associated with a tunable *weight*. The output of a node is based on the weights and the associated function with the neuron. Figure 2.2 shows a three-layered ANN with one input, output and a hidden layer. The ANN learns from the training data by adapting the weights of the connections. Even though an ANN is slow to train, it is fast to run and performs well for many classification problems. It is also quite robust to noise in the training data.

In this thesis, a three-layered ANN is realized as a Multilayer perceptron (MLP). This model is a feedforward ANN that consists of multiple layers of nodes in a directed graph where each layer is fully connected with the next layer. The data is fed in a forward direction from the input layer to the output layer through the hidden layer. MLP employs a technique called backpropagation for training the network

[Russell and Norvig 2003]. Each node, except for the input nodes, is associated with a non-linear activation function. More precisely, given a vector of inputs $a_1$ through $a_n$ and weights $w_1$ to $w_n$, the output function is determined as

$$o = \sigma(\vec{w}, \vec{a})$$

where

$$\sigma(b) = \frac{1}{1+e^{-b}}$$

'σ' is often called the sigmoid function where the threshold output is a continuous function of the input. Like a perceptron, this sigmoid unit first computes a linear combination of its inputs and then applies a threshold in the result.

$$\hat{b} = \sigma(\vec{w}_j.\sigma(\vec{w}_i, \vec{a}))$$

Given a fixed set of units and interconnects the backpropagation algorithm learns the optimal weights for the multilayer network. It employs a gradient descent technique for minimizing the squared error between the network output values and the actual values for these outputs. In this thesis a neural network is employed to predict the optimal thread number for a parallel program described in Chapter 4.

#### Regression Techniques

Regression analysis [Draper and Smith 1981] is a commonly used statistical technique to estimate a relation between variables. This is a generic method to fit a model to observed data to quantify the relationship between two types of variables: independent and dependent. As shown in Figure 2.3 regression analysis shows how a set of independent variables can effect a dependent variable. It helps to understand how the typical value of dependent variable varies with change in one or more independent variables. Once this heuristic is built, when new values for independent variables are given, this model can be employed to predict the value of output dependent value.

If $\boldsymbol{X}$ denotes a set of independent variables and $\boldsymbol{Y}$ a dependent variable, a regression model relates $\boldsymbol{Y}$ as a function of $\boldsymbol{X}$ and error $\beta$ as:

$$Y = f(X, \beta)$$

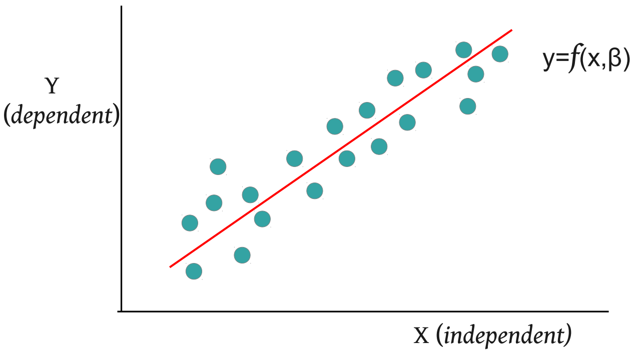


Figure 2.3: A simple linear regression model showing correlation between a dependent and an independent variable.

**Multiple Linear regression:** A multiple linear regression approach is used to model the relationship between the scalar dependent variable and a vector of input independent variables. This technique is a generalization of simple linear regression where the number of independent variables is two or more. Given *m* data points, the basic model for a multiple linear regression can be expressed as,

$$Y = \beta_0 + \sum_{i=1}^{m} \beta_i X_i + \varepsilon$$

where ε denotes error variable that adds noise to the relationship.

In this thesis, the least squares method is employed to fit the model to the training data. This is a standard approach to the approximate solution for optimizing overdetermined systems where there are more equations than the number of unknown values. As shown in Figure 2.4, the solution generated by the least squares method minimizes the sum of the squares of errors in results of each equation.

The objective of least squares is to adjust parameters of the model based on the difference between the actual value and predicted value, termed as a *residual (r)* . Given *m* data points *i=1,2,..m*; ***X***, a set of independent variables and ***Y***, a dependent variable, the relationship ***Y*** = ***f***(***X***, β ) has *m* adjustable parameters in vector β. The difference between the actual value and predicted value is termed a *residual*. The least squares method aims to minimize the sum *S* of the square of residuals as,

$$r_i = y_i - f(x_i, \beta)$$

$$S = \sum_{i=1}^{m} r_i^2$$

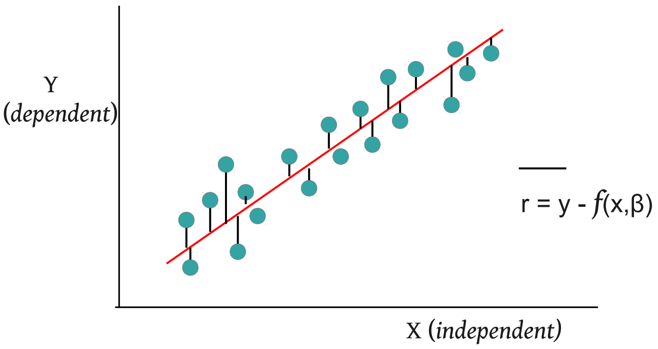


Figure 2.4: A simple linear regression model with residuals. Least squares method fits the data to minimize the sum of square of residuals.

### 2.3.2 Unsupervised Learning

Unsupervised learning deals with learning hidden structures in unlabelled data. Here an algorithm tries to find correlations with no external inputs apart from the raw data and could be able to cluster data into different classes. Few widely used approaches to unsupervised learning include clustering, hidden markov models [Ghahramani 2004].

### 2.3.3 Feature Selection

The efficiency of any machine learning model is dependent on the quality of the training data which is in part a collection of feature vectors. Not every feature in the input feature vector is essential in building the model. Hence, it is critical to choose only the rich subset of the total collected features that is most useful to discriminate between classes and ignore the redundant. *Feature selection* [Guyon and Elisseeff 2003] identifies the essential features from a set of all possible potential features. In this thesis, the features are selected based on *information gain ratio* metric, that determines how important is a given feature for the model relative to the rest.

### 2.3.4 Feature Impact

The *feature impact* is defined as the normalized percentage value of the difference of the prediction accuracies between a model built using all selected features and a model built using selected features without '*f*'. Hence, this value measures the drop in prediction accuracy of the model when a specific feature is not included in the training set. Thus, feature impact gives an insight into how important is a feature for the model.

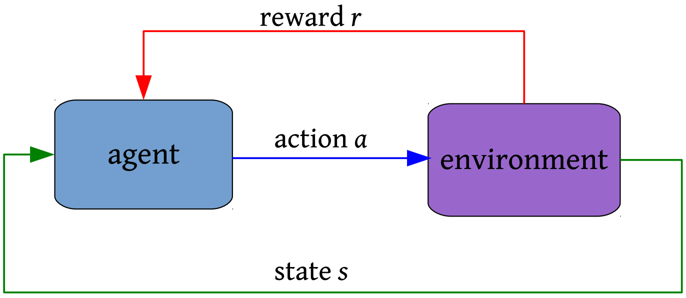


Figure 2.5: A Markov Decision Process showing the agent-environment interaction.

## 2.4 Markov Decision Processes

A process is *Markovian* if the next state depends only on the current state and current action; it is independent of previous actions as these are captured by the current state. A Markov Decision Process (MDP) is a discrete-time stochastic control process, [Puterman 1994], [Papadimitriou and Tsitsiklis 1987]. It is used as a mathematical framework for modelling decision-making process where the outcomes are associated with probabilities. MDPs are useful in studying a wide range of stochastic optimization problems solved via dynamic programming and reinforcement learning. Figure 2.5 shows an example of an MDP.

An MDP $(S,A,R,P)$ is a control decision process where decisions have probabilistic outcomes. At each time step $t$, the process is in a state $s \in S$ and must decide an action $a \in A$ that takes it to a new state $s' \in S$. The reward for such an action is denoted by $R(a,s,s')$. The goal of an MDP is to choose a sequence of actions $a_k$'s that maximise the total reward or value $V = \sum_a R$; this is known as the policy $\pi$. The probability that the process moves from one state to another given an action is given by the state transition function $P(s,s',a)$. The policy $\pi$ can be evaluated as:

$$\pi(s) = \arg\max_a \sum_{t=0}^{\infty} \gamma^t R(a,s,s')$$

where $\gamma$ is the discount factor such that $0 \leq \gamma < 1$.

MDPs are widely used to formally study resource allocation and scheduling problems [Huh et al. 2013, Yu et al. 2005, Liu and Ulukus 2006]. In this thesis, runtime scheduling of parallel code is modelled as an MDP. This formal mechanism is described further in Chapter 5.

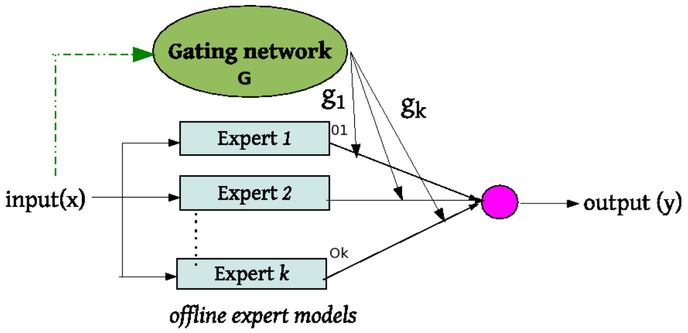


Figure 2.6: Mixture of experts: Depending on the input $x$, the online gating network chooses an expert most likely to select the best output $y$.

## 2.5 Mixture of Experts

Mixture of Experts [Jacobs et al. 1991], is a supervised learning technique based on divide-and-conquer method. The problem space is stochastically divided into number of sub-spaces by means of an error function. Experts are generated on each sub-space. A gating network uses a switching mechanism to select different experts for different sub-spaces. This technique improves predictions over using one model that covers the entire training space. The expert combining methods can be biased or unbiased with corresponding estimation errors [Masoudnia and Ebrahimpour 2014]. Expectation-maximization algorithm can be used to adjust parameters to each expert as discussed in [Jordan and Jacobs 1993]. Here, the learning process is treated as a maximum likelihood problem. Boosting technique [Edakunni et al. 2011] describes a probabilistic model of improving learning by using a product of experts that greedily selects models incrementally.

Figure 2.6 shows a mixture of experts model where the input $x$ is fed to $k$ experts $E^1,..,E^k$ with corresponding outputs $o_1,..,o_k$ and the gating network $G$. This network switches between experts based on probabilities associated with each expert. The output of the corresponding expert is then chosen as the output of the model. The gating model selects the best expert by learning the probabilities $g_1,..,g_k$ and adjusting them based on the current problem sub-space.

$$0 \leq g_1, g_2.., g_k \leq 1, \sum_{p=1}^{k} g_p = 1$$

$$G(x) = o_p | argmax_{g_p} E^p, p = 1,..,k$$

## 2.6 Evaluation Methodologies

This section describes the methods used in this thesis to evaluate the machine learning models for determining corresponding prediction accuracies. It also describes how the optimization metric, *speedup* is computed.

### 2.6.1 Cross Validation

Cross validation [Kohavi 1995] is a statistical technique that is used to measure the accuracy of supervised machine learning models. The residual evaluations do not give performance measurement of these models when they predict on new and unseen data. Here the entire training-set is split into non-overlapping sets: training-set and test-set. The model is learnt using the training-set and then is evaluated using the test-set. One of the widely used technique is *leave-one-out* cross validation (LOOCV). In this method, given a training data set of $K$ examples, LOOCV removes one sample from this set and trains the model with the other $K-1$ examples. This learned model is then evaluated to make a prediction on the removed sample. This process is repeated for all examples in the training-set. The accuracy of the model is then computed by the average of the prediction accuracy of all samples.

### 2.6.2 Performance Measurement

In this work, the primary objective is to optimize a parallel program by efficiently mapping threads. This relates to minimizing the total execution time of the program. To judge the extent to which the execution time is minimized, program performance is measured in terms of attained *speedup*. Speedup of a program is a relative metric comparing two techniques. Let the time taken by the baseline approach be $t_{baseline}$ and by the evaluated approach $t_{approach}$. The speedup obtained by evaluating this approach is computed by

$$speedup = t_{baseline}/t_{approach}$$

A value greater than 1 implies that the program's performance is improved over the baseline by reducing its execution time and vice-versa. Hence, this is a “larger the better” metric to evaluate any approach. The baseline used in all the experiments is the OpenMP default policy. Machine learning models are usually evaluated based on their prediction accuracy. Although this is an important metric, resulting speedup is more important in measuring program performance. However, there is a direct correlation between highly accurate machine learning models and larger speedup values.

The baseline used in all experiments throughout this thesis is the OpenMP 3.0 default policy with passive synchronization. The number of threads assigned by this policy at every parallel section is equal to the number of maximum available hardware threads.

It is also interesting to know the upper bound of achievable performance improvement to evaluate the efficiency of any approach. In this thesis, the *oracle* is a thread configuration for a program that achieves maximum speedup which highlights the room for improvement over existing techniques. However, it is not always possible to evaluate exhaustive combinations of threads a priori. Here the oracle is observed by evaluating configurations obtained by random sampling of the state space [Dubach et al. 2009]. Any mapping policy is compared against this observed oracle.

## 2.7 Summary

This chapter has introduced and described basics of multi-core architectures and how to program them using the OpenMP programming model. It then discussed basic concepts of machine learning with a detailed description of few algorithms used in this work. It has also outlined the methodologies and metrics used to evaluate the approaches presented in this work. The next chapter provides a detailed discussion of related prior work.

# Chapter 3

# Related Work

This chapter presents an overview of the related work. It starts with a description of techniques to map isolated parallel programs in Section 3.1. Relevant works are classified into static, dynamic and hybrid solutions. Section 3.2 discusses techniques to co-schedule parallel programs. This is followed by a description of work on online adaptation of parallel programs in Section 3.3. Here, the discussed approaches are broadly classified by the underlying mechanism into feedback-driven and machine learning-based. Finally, the chapter concludes in 3.4.

## 3.1 Parallelism Mapping

This section discusses approaches to map parallel programs under isolation. Static schemes determine fixed mapping policies of parallel work partitioning and mapping to underlying hardware. The mapping policy is unchanged till the end of a program execution. Dynamic schemes develop policies that can be modified at runtime in response to varying system and phase changes. Hybrid techniques combine static code features obtained either by offline analysis or compilers along with the runtime information to adapt during program execution.

### 3.1.1 Static

Parallelizing compilers such as SUIF [Hall et al. 1996], POLARIS [William et al. 1996] use heuristics to decide if a loop is profitable when parallelised. Analytical models are used to estimate the cost of computation and communication of a data parallel loop and this information is used to decide the distribution of loop iterations across threads.

[Hall and Martonosi 1998] discussed parallelism mapping using compiler-parallelized code. AutoTune [Miceli et al. 2013] is a tool that provides extensible plug-in mechanism for auto-tuning parallel programs. It recommends how serial and parallel codes can be tuned which can later be integrated with the production version of the code. This technique requires extensive offline profiling of the programs which may be highly expensive. Adaptive optimization technique proposed in [Arnold et al. 2005] is restricted to virtual machines and cannot be implemented on an NUMA machine.

In [Radojković et al. 2012] an approach for estimating the optimal performance of task assignment using statistical inference techniques by random sampling, is presented. This work is useful for estimating the upper bound of the performance of task assignment offline. However, this approach requires several thousand random task assignments a priori for an accurate estimation.

Data-parallel program mapping on multiprocessor system-on-chips (MPSoCs) for different criteria such as performance, power and energy is studied in detail in the work in [Chandramohan and O'Boyle 2014]. This work proposes a solution to partition the data statically amongst multiple threads during compilation. For mapping embedded streaming applications on MPSoCs with platform constraints, an algorithm for determining the optimal degree of parallelism is proposed in [Zhai et al. 2013]. When the streaming program is modelled as a directed graph, It determines the mapping based on graph alterations by task unfolding and the number of available processing elements at compile-time. This technique becomes fragile in presence of dynamic hardware changes. Mechanisms to exploit coarse-grained task, data and pipeline parallelism in stream programs are studied and discussed in [Gordon et al. 2006]. IWRAP framework [Balasundaram et al. 1991] and OpenUH compiler [Liao and Chapman 2007] frameworks use analytical models to partition data parallel loops.

[Wang and O'Boyle 2010] discusses an automatic compiler-based technique for partitioning StreamIt applications. In the process of compilation, the machine learning based partitioner forms a program feature vector and then predicts the optimal partitions. A solution to effectively port OpenCL programs based on a compiler transformation thread-coarsening is proposed in [Magni et al. 2014]. Based on the kernel static features, their model determines the optimal number of threads to merge in an OpenCL program.

### 3.1.2 Dynamic

A *factoring* scheme to schedule data parallel loops is proposed in [Hummel et al. 1992]. Here the iterations are split into chunks and are assigned to tasks on-the-fly with no lag waiting for unfinished tasks. Other runtime scheduling techniques primarily include guided self-scheduling [Polychronopoulos and Kuck 1987] and dynamic self-scheduling [Fang et al. 1990].

Work-stealing [Blumofe and Leiserson 1999], used in programming models such as Cilk [Blumofe et al. 1995], Intel TBB [Reinders 2007], is a technique that ensures under-utilized threads steal jobs from remaining threads that enables faster completion of tasks. The Galois programming model [Kulkarni et al. 2007] has a runtime scheduler that schedules tasks based on priority defined by the programmer. This approach is prone to produce sub-optimal schedules as the priorities may change during the program execution. HERMES [Ribic and Liu 2014] is a work-stealing language runtime that adjusts the task sharing between the threads during program execution. Effective co-operation between threads results in energy-efficient program execution. This scheme is primarily aimed at preventing underutilization and may not cope when threads are burdened with massive workloads.

In [Corbalán et al. 2000] a runtime system is proposed that assigns processors to OpenMP loops dynamically. This system assigns a given number of processors to loops and an analytical model is built based on the sampled information on the execution time. This model estimates performance improvement when the number of processors is varied. Speculative parallel execution is proposed in [Luo et al. 2009]. It uses hardware performance counters to monitor speculative threads, i.e. the threads that seem to be dependent but can be executed in parallel till the execution is verified at runtime. If there is any dependency violation, the systems rolls back to the previous stable state. A framework is developed where the runtime collects the performance characteristics of the speculative threads and fine tunes them dynamically. This approach adds significant overhead when the speculation violates some dependency adds overhead to the runtime and degrades the performance.

Petabricks [Ansel et al. 2009] is a programming framework that determines the best algorithm for the program at runtime to adapt to a dynamic system. This approach requires different implementations offline, which may not be possible for every program, hence narrowing the scope of its applicability. Here, the auto-tuning mechanism [Ansel et al. 2012] performs online tuning at runtime. This is achieved by executing a safe configuration along with tunable configurations and choosing the best amongst these. The main source of speedup improvement comes from PetaBricks framework rather than the online auto-tuning mechanism.

AKULA [Zhuravlev et al. 2010b] is a framework that provides an API for developing and testing various thread scheduling algorithms in real time. These algorithms decide the mapping of threads to cores by using thread affinity system calls provided by the kernel. Different scheduling algorithms include contention unaware, dynamically optimized schedule approaches. This approach does not consider the dynamic nature of the programs.

Default thread schedulers do not consider the locks held by various threads while migrating them to different cores. This results in a degraded performance for programs that have high lock contention. Shuffling [Pusukuri et al. 2014] is a framework for efficient thread scheduling based on lock contention. It migrates threads across sockets so that those threads that seek locks find the locks mostly on the same socket. This approach results in decreased execution time in shared data in critical sections. Techniques for adaptive scheduling for integrated CPU-GPU processors adapting to different program characteristics are proposed in [Kaleem et al. 2014]. It uses online profiling to automatically partition data-parallel work between CPU and GPU. It does not repartition the work at runtime.

Runtime scheduling on heterogeneous systems discussed in [Jiménez et al. 2009], proposes a user-level scheduler for heterogeneous machines based on past performance history and three scheduling mechanisms for tasks to any processing elements. Initial *n* calls to functions are executed on *n* processing elements (PE) and the performance is recorded. Once a task is ready to be scheduled on a PE, the ratio of its performance on one PE to rest of the PEs is computed based on the past performance history. If the ratio is greater than a threshold, the current task is scheduled to run on the current PE.

Techniques for adaptive parallelism in distributed memory environments are discussed in [Scherer 2001]. Piranha [Carriero et al. 1995] uses processors from idle machines to adjust the parallelism and withdraw when their owners need them.

### 3.1.3 Hybrid

Compiler-based techniques have been applied to assist runtime parallel scheduling in works such as [Sussman 1992], [Voss and Eigenmann 2000]. A system for adaptive load balancing that combines compiler and run-time support on distributed shared memory programs is proposed in [Ioannidis and Dwarkadas 1998]. Here, compiler hints assist the runtime system which distributes parallel loops across processors to minimize communication and page sharing. [Dave and Eigenmann 2009] describes a framework to tune parallel program performance. It relies on a source-to-source parallelizing compiler, Cetus along with a runtime tuning algorithm, Combined Elimination, that measures the effect of serializing loops which are parallelizable. A compiler and runtime framework for dynamic reconfiguration of multiple streaming programs is proposed in [Hormati et al. 2009]. However, it does not consider the impact on the external workloads.

Thread tailor [Lee et al. 2010] is a dynamic system that adjusts the number of threads to increase system efficiency. It uses offline analysis to finding out what threads might exist during runtime and their communication patterns. The runtime stitches together threads basing on their communication patterns thereby removing the unnecessary synchronization overheads. However, the offline-generated graph does not reflect the actual thread communication patterns that exist at runtime.

The mapping solution discussed in [Ye and Li 2010] to enable dynamic mapping of parallel program execution, uses runtime information with static parallel performance models. This is in the context of several simulation algorithms trying to improve the system throughput. Here, analytical models are used to predict on-the-fly the next best configuration but detailed performance issues are not dealt with, that can give a detailed insight into the generated model. However, this technique is not evaluated on unseen workload programs. PEMOGEN [Bhattacharyya and Hoefler 2014] is a framework that learns program performance models at runtime. This technique greatly reduces the storage and time taken to generate such models by complementing offline data collection with online model generation. The values of the parameters used are generated by static analysis and not updated at runtime.

Knowledge-based approach to exploit loop level parallelism by integrating existing loop transformations and scheduling algorithms is described in [Yang et al. 1995]. Here the knowledge is expressed in the form of production rules called IPLS (Intelligent Parallel Loop Scheduling). A program is profiled to extract code features such as the number of iterations, maximum and minimal time of iteration. This extracted information is used to modify the attributes of existing loop scheduling algorithms and then in deciding a specific scheduling method.

Approaches that determine adaptive hierarchical schedulers for parallel loops are presented in [Zhang and Voss 2005]. They propose fine tuning mechanisms of OpenMP programs for hyper-threaded architectures. These include a (1) scheduler that makes decisions for loop scheduling at individual parallel loop level and (2) hardware counter directed scheduler that uses hardware counters to sample performance of parallel loops and decide on a schedule using a decision tree that is trained off-line. The time taken for determining the optimal schedule adds to the overhead as it has to sample performance of different scheduling methods at runtime.

The technique proposed in [Grewe et al. 2013] maps data parallel programs to OpenCL for heterogeneous systems. It automatically generates optimized OpenCL code from data-parallel OpenMP code. A technique for OpenCL task scheduling on heterogeneous systems in proposed in [Wen et al. 2014]. Based on static code features, it predicts kernel's speedup on underlying processors. It uses this hint along with runtime input data to schedule tasks on CPU/GPU.

## 3.2 Program Co-scheduling

When multiple programs are executed together in the same system, these are said to be co-scheduled or co-executed. How resources are allocated to each program has a deep impact on different metrics that include programs execution time, throughput and system utilization. Static approaches determine optimal co-scheduling policies prior to execution and are not altered later. Dynamic schemes determine how programs can be co-executed at runtime based on the contention caused on system resources due to multiple running programs. Robust approaches that blend both are presented later.

### 3.2.1 Static

In [Grewe et al. 2011] a model to predict the number of threads to use when launching an application that is co-executed with external workloads. This, however, is an entirely static approach which is unable to adapt to varying program environment. Code optimization techniques when mapping multiple programs are proposed in [Coons et al. 2008], by assuming the program runs only on an unloaded machine. However, this technique penalizes the external workloads as the model is untrained in a loaded environment.

A mechanism for co-scheduling of memory-intensive programs is described in the work proposed in [Ebrahimi et al. 2011]. It proposes prefetch-aware resource management techniques. Here applications that have smaller demand requests are prioritized for scheduling so that they can get back quickly to computing phases. It also restricts the number of prefetches as more inaccurate prefetches negatively impacts system performance and fairness.

Bubble-up [Mars et al. 2011] quantifies the performance degradation caused by the interference due to co-location of a pair of workloads. This metric is then used for devising an optimal scheduling policy for both co-located programs. This approach assumes a priori knowledge of the applications and lacks adaptability to the dynamic nature of programs.

### 3.2.2 Dynamic

ParallelismDial [Sridharan et al. 2013] is a model for runtime parallelism management. The parallelism degree is obtained by executing three different configurations for a fixed duration and selecting the best. This local optimum is then discarded by evaluating new configurations, which consumes significant time to reach the optimal thread number in response to change in the system. This work is improved in [Sridharan et al. 2014] that developed an analytic model to determine the degree of parallelism at runtime. Based on observed instantaneous performance, it executes for fixed time intervals with two randomly chosen thread numbers. The new thread number is then estimated using regression techniques. Such exploratory process incurs significant overhead.

CoAdapt [Hoffmann 2014] tries to optimize multiple parameters such as performance, power and accuracy adapting to changes in program phases and goals. It relies on feedback control to fine tune configurable knobs. However, the monitoring process is discrete (fixed time-steps) and slow to react to system changes. Techniques to improve program performance and utilization proposed in works such as [Tang et al. 2011] reduce resource contention caused by the programs. However, this approach does not examine the dynamic nature of workloads.

Co-location of compute intensive workloads is detailed in [Breslow et al. 2013]. Quasar [Delimitrou and Kozyrakis 2014] improves system utilization by efficient resource scheduling once the amount of interference is determined. This system monitors the workload performance and changes the resource allocation when needed. This work requires programmers expertise in defining constraints on workloads.

In [Leverich and Kozyrakis 2014], a system to analyze QoS violations arising due to workload co-location is presented. It proposes interference-aware scheduling mechanism by replacing Linux CFS scheduler to provide latency guarantees. DeepDive [Novaković et al. 2013] allocates resources to programs by identifying and managing the interference between virtual machines that are co-located on the same system.

A detailed study of benefits of simultaneous multithreading with dynamically varying number of threads in the system is discussed in [Eyerman and Eeckhout 2014]. It concludes that within same power budget, a homogeneous multi-core processor with few high performance simultaneous multithreading (SMT) cores outperforms heterogeneous multi-cores consisting of a mixture of big and small cores without SMT when the number of active threads vary over time. A price theory-based framework for dynamic power management for heterogeneous multi-cores is presented in [Somu Muthukaruppan et al. 2014]. It coordinates between load balancing, task migration and other optimization techniques. The three step iterative process for every round of bidding for resources adds significant lag in reaching a stable state. A dynamic scheduling policy described in [McCann et al. 1993], re-allocates processors to programs based on the available parallelism in the programs. It also highlights the kernel and application synergy in allocating resources to applications.

Parcae [Raman et al. 2011, Raman et al. 2012], is a dynamic tuning framework that generates flexible parallel programs that can be dynamically reconfigured during execution. At execution time, the runtime monitors parallel task executions and uses hill climbing method to adjust the thread count iteratively for each task until an optimal thread configuration is found. Although this approach to robust to any system changes, it is slow to react and reach the optimal state.

In [Zahedi and Lee 2014] an approach is proposed for fair resource sharing amongst co-scheduled jobs. This is based on Cobb-Douglas utility functions that benefit all jobs. Techniques to improve QoS are proposed in [Tang et al. 2011] and [Tang et al. 2013] that reduce resource contention caused by the programs sharing the system. However, they do not consider the dynamic nature of workloads. When memory-intensive threads from co-running programs are competing for computing resources on an NUMA machine, a contention-aware scheduler is built in [Blagodurov et al. 2011]. A detailed analysis of state-of-art approaches for allocating resources to software applications applied to self-optimizing autonomic systems is presented in [Maggio et al. 2012]. Bubble-Flux [Yang et al. 2013] co-locates programs by measuring the pressure on shared resources and estimating its impact on the performance. At every decision interval, programs are regularly paused to monitor the workloads performance with and without interference, that may lead to lag in the execution in the workloads.

Basics of job scheduling on large-scale parallel systems and different algorithms are discussed in [Weinberg 2006]. It defines a job, a scheduling policy, a good schedule in terms of performance, fairness and predictability and also discusses space-sharing and time-sharing modes of scheduling. Load balancing in dynamic environments is discussed in [Boyer et al. 2013]. It changes the partitions in the program automatically responding to fluctuations in the device. Initially, a small portion of the parallel work is sent to each device and based on the observed execution behaviour, it further partitions the work. If this work portion does not reflect the nature of the entire work, the partitions generated will be sub-optimal.

Symbiotic job scheduling finds the best mix of jobs [Snavely and Tullsen 2000, Eyerman and Eeckhout 2010] on SMT processors. This is a fine grain scheduling technique that targets CPU time allocation. Jobs that benefit from co-scheduling are identified by observing the performance behaviour sampled at random intervals that takes a considerable portion of the execution time. It, however, does not need to know the program characteristics in advance. Throughput-Driven Fairness (TDF) scheduling policy in [Rangan et al. 2011], is a hardware mechanism that aims to maximize the system throughput while providing uniform level performance to the software. This is achieved by creating an illusion that the processors operate at a single frequency. However, this may lead to degradation in the per-program performance. An approach to determine runtime program schedules to minimize energy, using local search heuristics is proposed in [Bhadauria and McKee 2010]. This approach assumes programs to be executed are known ahead of time.

### 3.2.3 Hybrid

Sambamba [Streit et al. 2012] is an automatically parallelizing compiler and runtime system to adapt parallelization to program input and execution environment. Initially it analyses the program for identification of potential parallel implementation and executes them speculatively. Conflicts are resolved using a software transactional memory (STM) system and the data obtained is used for further optimizations. There is no monitoring mechanism to judge the quality of identified parallel sections and the overhead in employing an STM system is non-negligible.

ReSense [Dey et al. 2013] uses resource sensitivity to map dynamic workloads of co-located applications. A sensitivity score obtained by offline characterization per-program is used to determining thread mappings at runtime. This approach assumes all co-executing programs are known in advance, and scores do not reflect the sensitivity changes during co-execution. In [Zhuravlev et al. 2010a] a classification scheme for co-scheduling threads is developed to determine the effect on one program on the other when they compete for shared memory resources. The scheduling policy first sorts co-scheduled applications based on cache miss rates. It then re-distributes them across cores, such that the total miss rate of all threads that share a cache is the same across all caches.

Memory-aware scheduling [Merkel and Bellosa 2008] identifies memory characteristics of programs and run-queues per core. It uses this information to co-schedule programs to minimize overall energy by changing frequencies of cores at runtime. In [Merkel et al. 2010], a novel co-scheduling technique combines applications that use complementary resources to improve performance and energy efficiency. When such co-scheduling does not improve program performance, then the processor frequencies are further fine tuned.

## 3.3 Online Adaptation

Here mapping techniques that adapt during program execution at runtime are presented. In particular, approaches that are feedback-driven and machine learning are discussed.

### 3.3.1 Feedback-driven

Feedback-driven threading [Suleman et al. 2008] determines the number of threads needed for a parallel section of an OpenMP code. This number is determined at runtime and hence is adaptive to phased behaviour of the programs, which are known ahead of time. This approach is tested on a similar program but no mention of the result when a new program is encountered is mentioned. Also, the time spent in the training phase during runtime adds to the overhead in the program execution time. Feedback driven identification of potential profitable parallel sections of Haskell code is described in [Harris and Singh 2007]. After profiling the program execution and promising parallel sources are found; it is recompiled and executed speculatively using work-stealing system.

A framework to generate a parallel runtime optimization policy is discussed in [Penry et al. 2010]. It uses program parallelism and locality information defined in Exposed Parallel and Locality information (EPL) and online performance feedback to map the application to underlying architecture. In [Lattimore et al. 2014] they propose optimal resource allocation mechanisms based on semi-bandit feedback policies. The mapping is such that the number of jobs expected to complete within required constraints are maximized. The proposed policy is built from scratch at runtime and suffers from considerable delay in reaching the optimal allocation. Other approaches deal with reliable feedback techniques [Diniz and Rinard 1997], supervisory control mechanism [Karsai et al. 2003], [Ţăpuş et al. 2002].

In [Hoffmann and Maggio 2014] an approach to optimize program performance with power constraints through efficient resource management. In [Nathuji et al. 2007] a framework is proposed that tunes resource allocation to programs based on a feedback-driven policy to minimize interference effects. On a heterogeneous data center, this system predicts the workload performance across various platform architectures and determines the best allocation of resources to workloads.

### 3.3.2 Machine learning-based

Machine learning-based techniques for mapping data and streaming parallelism to multi-core machines are proposed in [Wang 2011]. Advanced machine learning techniques such as Reinforcement learning, [Sutton and Barto 1998], where the system learns on-the-fly are used for mapping in solutions developed in [Vengerov 2009]. Pure online approaches like these, suffer from a significant initial drop in performance, it takes some time to reach an acceptable level of performance. They spend most of execution time in exploiting best mapping policy at every mapping decision instance. In a partially observed environment, a technique to performance tuning using reinforcement learning is presented in [Vengerov 2008]. Fast heuristic construction using active learning is discussed in [Ogilvie et al. 2014]. This technique significantly reduces the training overhead in generating the mapping heuristics to decide the device to port a parallel code.

In [Long et al. 2007] a machine learning based approach for parallel workload allocation amongst Java threads is presented. Assumptions include that programs with similar workloads are likely to benefit from the same number of threads / parallel scheme. This approach extracts static features for a newly encountered program and compares with existing learned examples. It then applies the parallel scheme i.e. determines the number of threads to share the workload. So only static feature comparison is considered for evaluation. But the actual program execution behaviour is known only at runtime; the lack of this information leads to sub-optimal performance.

An approach proposed in [Curtis-Maury et al. 2006] uses a regression model based on hardware performance counters to determine the best number of threads allocated to a single parallel program for energy-efficiency. In [Moore and Childers 2012] linear regression-based utility models are built to predict the thread counts for parallel programs. They build a model for each application using profile data. Self-optimizing memory controllers based on reinforcement learning are proposed in [Ipek et al. 2008]. This low-level approach adds overhead in mapping decisions.

Milepost GCC [Fursin et al. 2008] is a machine learning-based self-tuning compiler that adapts to multiple architectures. It uses empirical iterative compilation techniques using program features to predict good optimizations for new, unseen programs. In [Agakov et al. 2006] a technique is proposed to select specific parts of the program optimization space that are likely to give good performance. This approach increases the speed in optimization space pruning. This approach relies mainly on static code features. Iterative optimization for mapping is evaluated across 1000 data sets in [Chen et al. 2010], which states that there exists at least one combination of compiler optimizations that achieve at least 86% best possible speed-up. This work makes the program optimization across data sets simpler. A workload allocator based an iterative search technique, simulated annealing, is proposed in [Gordon et al. 2002b].

## 3.4 Summary

This chapter has provided an extensive review of prior work in related areas looked into in this thesis, to the best of the author's comprehension. It describes the work in mapping a single parallel program followed by co-scheduling multiple programs, both using static, dynamic and hybrid approaches. It then lists out approaches for online adaptation using feedback-driven and machine learning-based techniques. The cited works span from early theoretical methods to state-of-the-art techniques. The main contributions in this thesis are discussed in detail from the next chapter.

# Chapter 4

# Parallelism Mapping in the Presence of Dynamic Workloads

This chapter presents a predictive modelling-based approach to improve parallelism mapping when a target program co-executes with varying workloads. At the heart of this approach is a machine learning model that determines the optimal number of threads for any parallel section of a program.

This work is introduced in Section 4.1. Next, Section 4.2 highlights the motivation behind tuning this parameter for improved efficiency and shows the existence of large room for improvement over existing state-of-art approaches. The ML approach is described in Section 4.3. The evaluation methodology is described in Section 4.4 followed by discussing experimental results in Section 4.5 and analysis in Section 4.6. This chapter concludes with final remarks in Section 4.7.

## 4.1 Introduction

Effective use of multicore-based parallel systems that dominate computing landscape is challenging. There has been much success in compiler-directed approaches to map parallel program mapping: determining optimal number of threads and how to assign work to these threads. One widespread assumption in such approaches is the availability of the entire machine. In reality, several workloads co-execute as well that leads contention for system resources.

Entirely static compiler-based approaches [Gordon et al. 2002a] are likely to perform poorly as they do not foresee the actual runtime environment. When a program demands too many system resources it is penalized due to a rift between the compiler and the operating system. A compiler which maximizes a program's performance may inadvertently adversely affect it. Dynamic runtime scheduling approaches [Raman et al. 2012], [Sridharan et al. 2014] match program parallelism to available resource: expand when there is space available and shrink when resources are used up. However, these approaches are generic and use undifferentiated mapping policies.

As the workloads are dynamic in nature, the degree of contention changes throughout the lifetime of the application. Solutions to improve program scheduling in the presence of external workloads, [Grewe et al. 2011] do not consider the fact the workloads can be dynamic and hence do not change the mapping at runtime if required. It seems sensible, therefore, for a compiler and runtime to improve performance without adversely affecting the external workload.

The proposed approach takes program and runtime system information to determine the best number of threads. It aims to improve target program performance with minimal effect on co-executing workloads.

## 4.2 Motivation

Determining best parallelism mapping is undoubtedly critical in maximizing programs' performance. This section illustrates this requirement. It also stresses that determining best thread number is non-trivial, and it depends on both the program being scheduled and the dynamic system load.

### 4.2.1 Example

Consider a scenario shown in Figure 4.1 where the upper and lower graphs describe the target and workload behaviour over time, where x-axis represents time in seconds and y-axis, the number of threads. The programs are chosen from NAS parallel benchmarks (see Appendix A). The diagrams show the number of threads of an OpenMP program `lu` co-scheduled with varying external workloads `cg` as $W_1$ with 8 threads and `mg` as $W_2$ with 6 threads on a 12 core machine. Figure 4.2(a) shows a zoomed-in view for the first four seconds. Here the change in thread numbers can be clearly observed.

The upper graph shows the behaviour of the target program using two scheduling policies: the OpenMP default and a robust adaptation technique [Raman et al. 2012]

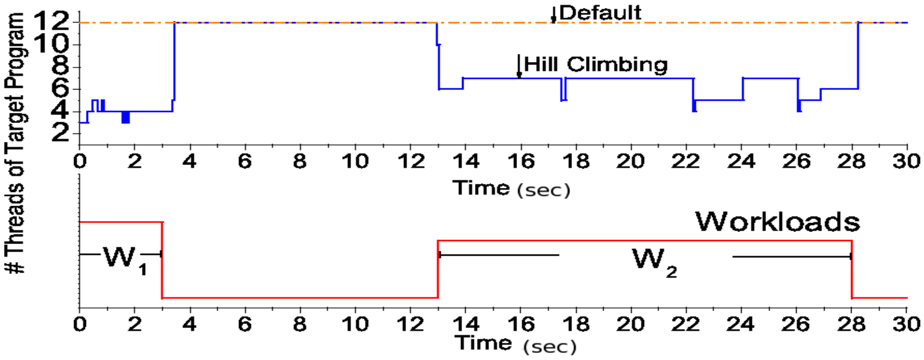
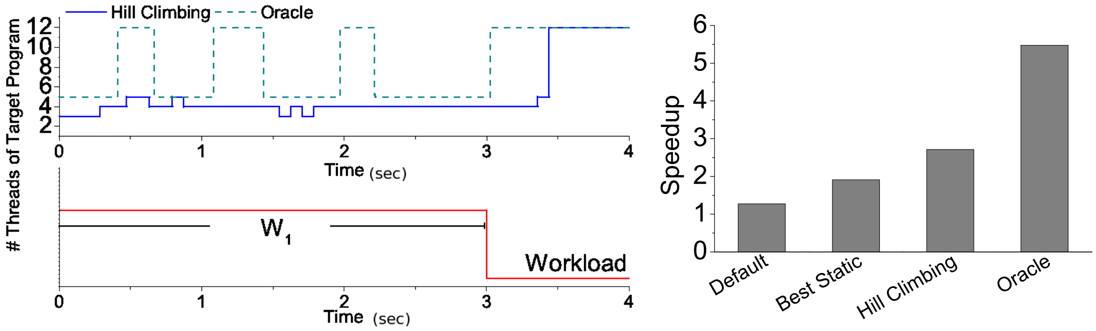


Figure 4.1: Target thread configurations implemented by different schemes in reaction to change in workload.



(a) Reaction to Workload (Zoom in View)

(b) Achieved Performance

Figure 4.2: The close view (a) compares default, online schemes with the best possible thread configuration. Corresponding speedup graphs (b) show the scope for improvement.

that uses hill-climbing technique. The observed speedups are shown in Figure 4.2 (b) where x-axis represents the scheduling policy and y-axis shows the speedup over sequential scheme.

OpenMP default policy allocates the same number of threads as the number of cores, 12. This can be observed a constant number of threads, 12 in Figure 4.1. This translates to a speedup of 1.12x. The best number of threads for this workload scenario is 4 which gives just a 1.4x speedup. The online adaptation approach reacts to the changing workload by increasing or decreasing the number of threads of each parallel section in unit steps based on loop execution time in previous three instances. This is repeated till no further improvement is observed. This scheme achieves an improvement of 2.13x on average.

It is also important to know the upper bound of achievable speedup. To realise this “oracle” policy, an exhaustive evaluation of all possible thread numbers were assigned

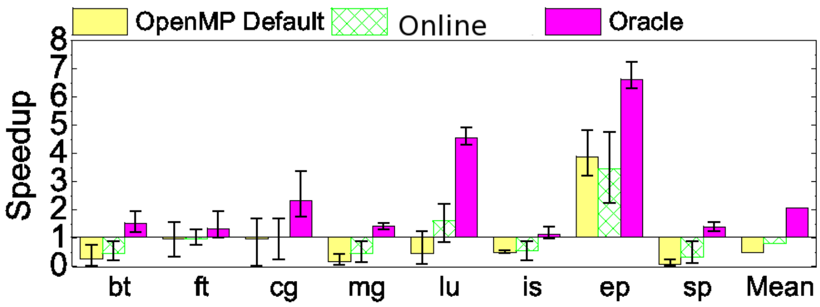


Figure 4.3: Performance comparison of existing approaches. There is significant room for performance improvement.

to each instance of a parallel loop. This can be achieved by switching instantly to optimal thread numbers at every parallel loop based on the contention caused by external workload. Figure 4.2(b) shows that if optimal number of threads were chosen, it can improve program performance by 5x times. This example highlights that though existing dynamic policies achieve better performance than pure static schemes, there is still a large room for improvement.

### 4.2.2 Room for Improvement

Previous section highlighted the existence of an ample room for improvement for a specific program. However, it is important to verify if this room exists across in wide cases. In a controlled limit study, the existing approaches are evaluated on NAS benchmarks and compared to idealised oracle. Figure 4.3 shows the average speedup achieved over sequential execution on a 12-core multi-core system (see section 4.4) with each workload. The workloads consist of just one program from the benchmark suite executed with 1 to 12 threads. Each bar in the diagram, therefore, corresponds to the performance of the target program averaged over $8 \times 12 = 96$ distinct workloads runs with different scheduling policies, repeated 10 times. So it incurred a one-off cost of $8 \times 8 \times 12 \times 12 = 9216$ experiments (8 target programs, 8 workload programs, 12 thread configurations of both target and workload) . To keep the experiments tractable, the target and workload are started at the same time and ensured that both execute till the other finishes. This is to maintain the contention consistent throughout the lifetime of the target program.

As an exhaustive experimentation is implausible in a dynamic system, this evaluation is limited to controlled static experiments. The default scheme performs poorly,

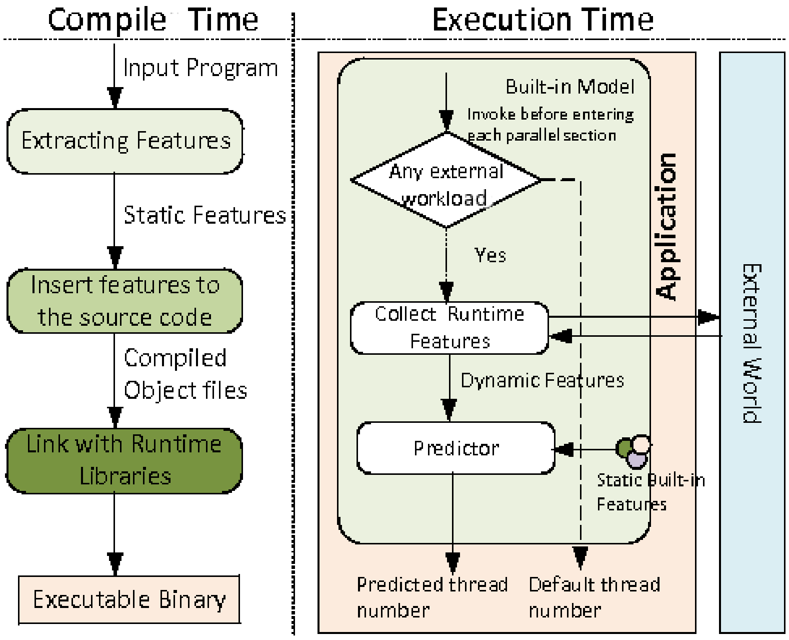


Figure 4.4: Flow chart of deploying the machine learning-based thread-predictor.

but the online scheme improves over it only with a meager improvement. This is due to a slow reaction to workload variations. On average across all target programs, the best scheme achieves 2x speedup that is the maximum achievable improvement. Hence even in the limited case of one workload, there is a significant room for improvement across programs.

## 4.3 Predictive Modelling-based Mapping

The approach presented in this chapter develops a predictive modelling-based heuristic '*thread-predictor*' that predicts best thread number for every parallel loop. It imbibes program-centric behaviour by considering static code features and dynamic runtime features in its prediction.

Figure 4.4 shows the flow diagram of how this model is used during program deployment. During *compilation*, program information is extracted in the form of static code features. It links the compiled version of the program with a runtime library that consists of an automatically learned heuristic. For each parallel section, the compiler inserts a routine to runtime where the static code features of that parallel section are passed as a parameter. During *execution*, the runtime library combines these program features with external workload information as inputs to an automatically

| Static features | Dynamic features |
|---|---|
| $f^1$: Load/Store count *(lscount)* | $f^4$: Number of workload threads *(wthreads)* |
| $f^2$: Branch count *(branches)* | $f^5$: Number of available processors *(numproc)* |
| $f^3$: Instruction count *(instcount)* | $f^6$: Runtime queue size *(runq)* |
| | $f^7, f^8$: Load *(ldavg-1, ldavg-5)* |

Table 4.1: List of features.

learned heuristic that returns optimal number of threads for this parallel section. The parallel loop then executes with this predicted number of threads.

### 4.3.1 Automatic Heuristic Generation

The heuristic used is generated automatically using *supervised learning* to enable portability across different hardware platforms. This model is built using the standard three step supervised learning mechanism which is described in Section 2.3.1.

#### Features

Capturing essential characteristics using features is of crucial importance when building a machine learning model. A set of numerical values is used to form a *feature vector* to represent the static program features and dynamic runtime workload. The selected features are listed in Table 4.1. During the training phase, all possible set of features are collected. These 134 features comprised of many code and runtime workload parameters. From these, eight features were chosen based on the feature-selection mechanism (Section 2.3.3) i.e. on the quality of information gain attributed to the prediction accuracy of the model.

**Program Features:** The set of features comprises of program code features that include static instruction, memory and branch summary information. The code features at every loop were normalized to the total number of instructions in the program. This information is sufficient to differentiate across programs for thread selection. Code features characterize a program's inherent compute, memory or I/O intensive properties. These features are obtained by an LLVM-based compiler [Lattner and Adve 2004].

**Workload Features:** A set of profiling tools is used to collect dynamic workload features. These are extracted from the kernel. To characterize the runtime environment, three features from */proc* filesystem were used. These are *run queue size*, *ldavg-1*,

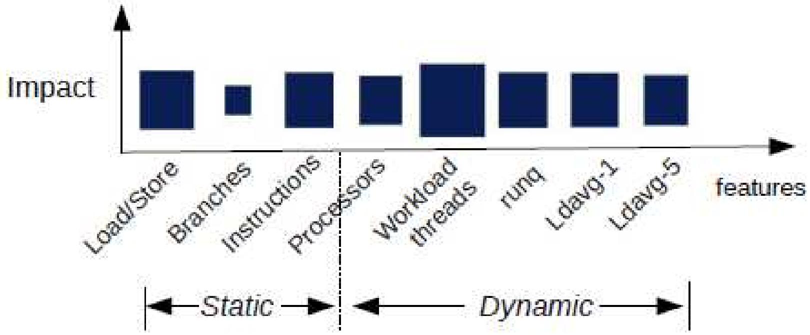


Figure 4.5: Hinton diagram showing impact of selected features on the model.

*ldavg-5*. The *run queue size* denotes the number of processes waiting to be scheduled in the Linux kernel. The *ldavg-n* is system load average that is computed as the average number of runnable or running tasks and the number of tasks in uninterrupted sleep over an interval of $n$ $(n = 1,5)$ minutes. Therefore the runtime features capture the current system load and the impact of external workload. In addition to these metrics, the feature set includes the number of workload threads and the total number of processors. These features are organized as a feature vector that is the input to the machine learning heuristic.

### 4.3.2 Feature Impact

In any mechanism that uses a predictive model, it is crucial to understand how the features selected impact the prediction accuracy. A Hinton diagram (Figure 4.5) shows the feature impact (see Section 2.3.4) of the selected features on the prediction accuracy of the model. Here the area of a square is a direct measure of the impact created by that feature. It can be observed that chosen static and dynamic features are equally important for the model with load/store count and workload-threads having biggest impact among the selected features.

#### Generating Training Data

Unlike previous approaches where the machine learning model is trained for each target program [Moore and Childers 2012], this model is trained using the data collected from a synthetic workload setting. The training data is generated by exhaustively running each training program together with a workload program. During the training,

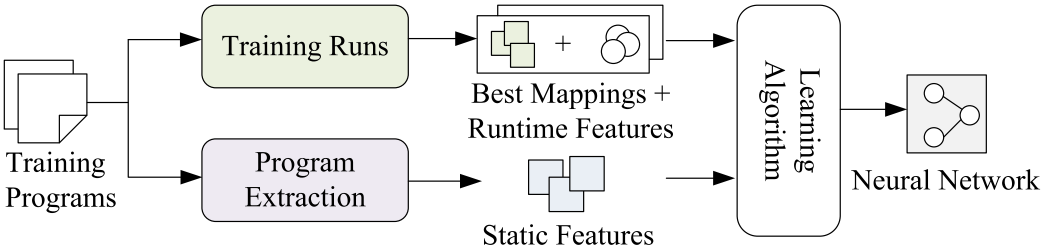


Figure 4.6: Building the predictor using training data.

the number of threads used for the target and the workload programs is varied. The best thread mappings and the execution time are explicitly recorded. During the generation of training data the set of *features* are collected to characterize the program along with runtime features that are then used to train a heuristic which is later evaluated in an unseen setting. Those runtime features and the best performing thread number, $t_{best}$, are put together to form the training data set, $\{v_i;t_{best,i}\}, i = 1,...,N$. Although producing training data takes time, it is only an one off cost incurred by this approach. The model is trained only once offline and frozen. No further learning takes place during the program execution.

Figure 4.6 describes how to train a heuristic from the training data. The training algorithm is run with training data collected offline. Each such data item includes the code, runtime features along with the best mapping. The training algorithm tries to find a function $f$ which, takes in a feature set, $v$, and gives a prediction, $\overrightarrow{t}$, that closely matches actual best mapping, $t_{best}$ in the training data set.

### 4.3.3 Building the Heuristic

This heuristic is based on an *Artificial Neural Network* [Bishop 2006] that employs Multilayer Perceptron approach with a single hidden layer. This model is described in detail in 2.3.1. This network learns by *back propagation* that is a generalized form of linear mean squares algorithm. The predictive model is automatically constructed from the training data as discussed in the previous section. Figures 4.7 and 4.8 show examples of how the neural network is used. The probability distribution of the number of threads to use is calculated based on the feature set for the target program in Section 4.2. The output is shown for two samples during the execution with workloads $W_1$ in 4.7 and $W_2$ 4.8. With $W_1$, 5 threads are selected as it has the highest probability and corresponds to the oracle value in Figure 5.2 . With $W_2$, 6 threads are selected.

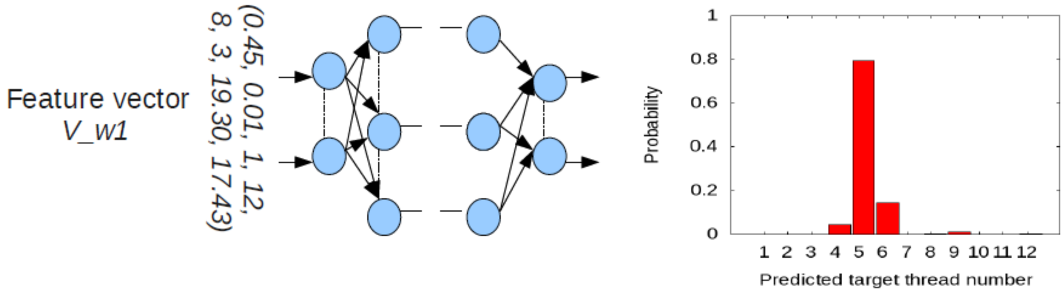


Figure 4.7: Predictive model employing ANN with input feature vector and probability distribution of predicted thread number (5) at workload $W_1$.

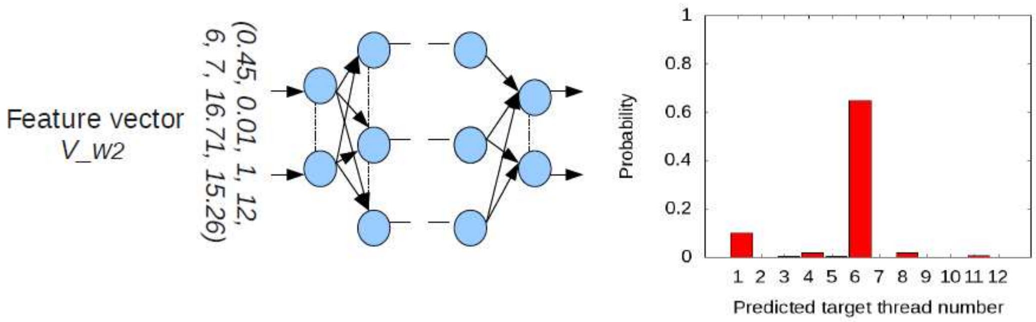


Figure 4.8: Predictive model employing ANN with input feature vector and probability distribution of predicted thread number (6) at workload $W_2$.

### Comparison with other learning algorithms

It is worthwhile to decide which machine learning model to deploy from the existing algorithms. Many factors influence this decision including the training data size, tradeoff between performance and accuracy and size of input feature to the model. Since the determined thread number greatly affects the programs performance, predicting this value needs to be highly accurate. This implies that the chosen predictive model should have maximum prediction accuracy value along with being extremely lightweight.

Based on the collected training data, different classification algorithms such as artificial neural networks (ANN), decision trees (C4.5) [Quinlan 1993], [Quinlan 1986] and support vector machines (SVM) [Boser et al. 1992], [Cortes and Vapnik 1995] were evaluated. Prediction accuracies for models learnt using ANN, decision tree, SVM were 81.58, 74.73, 78.65 respectively. High prediction accuracy suggests that the predicted values are highly likely to be optimal. Training time was the lowes for decision tree model and highest for support vector machines. Depending on the prediction accuracy values and time taken to train the model, ANN was employed to build a predictive model.

### 4.3.4 Runtime Deployment

Once the training data is generated and the model is learnt as described above, it can be deployed in any *unseen, new* program. At runtime, the runtime library is invoked that looks for any co-executing workloads. If there is no workload, the target program runs with the default configuration using all available physical threads. However if it detects any co-executing workload, the runtime features are collected from */proc*. The code features along with these runtime features are passed as input to the neural network that predicts optimal number of threads. The overhead in calling the runtime library, collection of features and determination of the thread number is negligible (a few microseconds) which is included in the measured execution time.

## 4.4 Experimental Set-up

This section summarises the experimental methodology along with a description of the hardware platforms and the benchmark programs that are evaluated. It further describes the state-of-the-art techniques which this approach is compared against.

### 4.4.1 Hardware and Software Configurations

All the experiments were carried on an Intel Xeon platform with two 2.4 GHz six-core processors (12 threads in total) and 16 GB RAM, which is a shared memory, homogeneous machine. The operating system was Red Hat 4.1.2-50 running Linux kernel 2.6.18. The programs were compiled using *gcc* 4.6 with parameters “-O3 -fopenmp”.

### 4.4.2 Benchmarks

In total, 11 benchmark programs were chosen to evaluate this approach. These include all OpenMP-based C programs from NAS, SpecOMP benchmark suites as listed in Appendix A. Smallest input data set was used for feature collection. However, for evaluation largest data sets were used for each program.

| Category | Number of programs | Total workload threads |
|---|---|---|
| Light | <2 | <6 |
| Medium | [2-5] | [6-12] |
| Heavy | >5 | >12 |

Table 4.2: Workload Settings.

### 4.4.3 Workloads

To evaluate this approach on different degrees of contention each target program is co-executed with a different set of workload programs. For this purpose the workloads are classified into three categories: *light*, *medium* and *heavy* depending on the number of external programs and the total number of threads used by them. Table 4.2 lists the setting of each category. As mentioned in Section 4.1 in any realistic computing environment, workloads *enter* and *exit* the system at varying rates. To characterize this behaviour, two workload arrival patterns were introduced. These are (a) low-frequency and (b) high-frequency where workload programs arrive at 2, 10 second intervals respectively. Thus the combination of three types of workloads and two types of arrival patterns results in six workload scenarios.

Overloaded systems are modelled to be systems with heavy workloads arriving at high frequencies. Here the arrival rate of workload programs is higher than the finishing rate. In this scenario, the system has to allocate resources to the target program to minimize the resource contention and hence stabilize the system quickly. Ideally, the optimal number of threads assigned to the target program should be minimal as the external workloads already oversubscribe the system. Here the competitive techniques take long time to reach to a stable state by determining thread numbers which are greater than required. However, the exact set of workloads (programs and arrival frequencies) are used to evaluate all compared approaches for a fair evaluation.

### 4.4.4 Comparison

The proposed approach is compared against with the OpenMP default scheme and state-of-art online technique.

- **Default**: The OpenMP runtime selects the number of threads to be equal to the number of hardware threads by default. It uses such undifferentiated policy and assigns same number of threads to all executing programs.

- **Online**: A robust online adaptation approach that uses hill climbing technique is proposed by [Raman et al. 2012]. Here at every parallel section, thread numbers are increased or decreased in unit steps based on loop execution times of previous instances until the speedup of the parallel section saturates. It helps in adapting to the dynamic system.

### 4.4.5 Methodology

For each target program, each experimental run is repeated for 10 times and the measured execution time is averaged across these runs. Each program ran with different workload programs and the average speedups across different workload program sets is reported with a statistical min-max bar. (where the top and the bottom of the statistical bar represents the maximum and the minimum speedups respectively).

To ensure that the machine learning model does not memorize the training data, it is evaluated using standard leave-one-out *cross-validation* technique. This technique ensures that the program to be evaluated is not included in the training set. This model is trained on a static pair-wise workload setting and is evaluated on completely *new, dynamic* workload settings that have not been seen in the training stage. This process ensures the model always predicts on an unseen program.

## 4.5 Experimental Results

This section first summarizes the performance of this approach against existing comparative scheme mentioned in the earlier section. The experimental results are averaged across all workload settings. Next, for each workload setting a detailed analysis is described. Later this approach is evaluated on a real-world case study to highlight the efficiency in a live workload pattern. Finally, the prediction accuracy of the model is evaluated by comparing its prediction to the oracle scheme.

### 4.5.1 Overall Results

Figure 4.9 shows the performance results on six different workload scenarios averaged across all benchmark programs. In a given workload setting, the speedup improvement

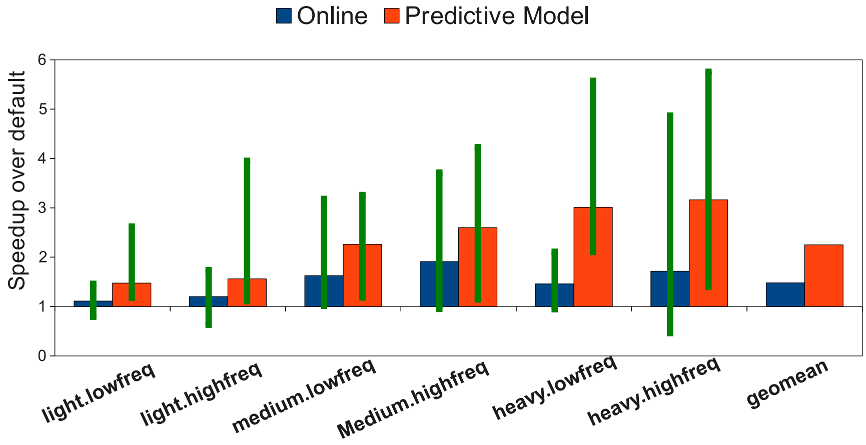
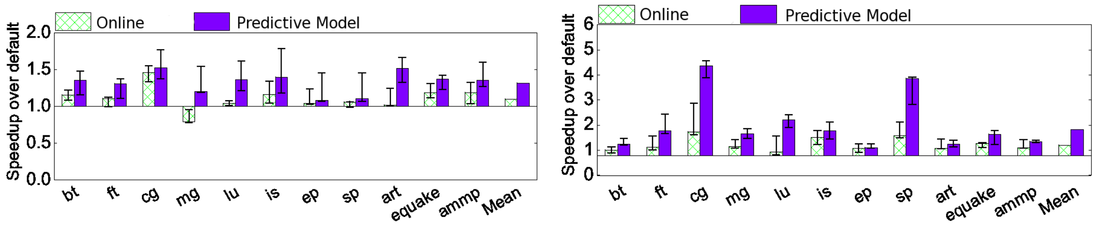


Figure 4.9: Comparison of speedups of online scheme and this approach over the OpenMP default under different workload scenarios averaged across all target programs. The min-max bars show the ***ranges*** across different target programs. On average the heuristic-based approach outperforms the online scheme (2.25x vs. 1.48x) and does not slow down the target program in any case.

varies for different programs. Hence, the min-max bars in this graph show the *range* of speedups achieved across various target programs. The baseline is the OpenMP default scheme.

**Online:** This scheme achieves a mean speedup improvement of 1.47x. For workload settings consisting of light programs, its improvement is small (less than 1.3x). For the medium and heavy workload settings, it is able to greatly improve the OpenMP default scheme with speedups over 1.5x. However, by looking closely to the min-max bars, it can be observed that this approach may actually slowdown some target programs by giving a speedup below 1. This is in particular for the high frequency workload programs owing to slow reaction to this environment.

**Predictive Model:** The heuristic-based approach not only gives better performance when compare to OpenMP default but also significantly outperforms the online adaptation scheme across all workload scenarios. For light workload settings, it is not surprising that the OpenMP default scheme performs reasonably well. Under such a setting, this approach gives the least improvement with a speedup of 1.5x. When considering medium and heavy workload settings, the predictive model approach has a clear advantage with speedups above 2.4x (up to 3.2x) over the OpenMP default scheme. This translates to a speedup over 1.36 (up to 2.3x) when comparing to the online adapta-



(a) Low frequency workload arrival rate

(b) High frequency workload arrival rate

Figure 4.10: Speedups of target programs with *standard deviation* for *light* workloads. This approach outperforms both existing schemes for all benchmarks and achieves a mean speedup of 1.31 and 1.82 over OpenMP default for (a) low and (b) high frequency arrival rates respectively.

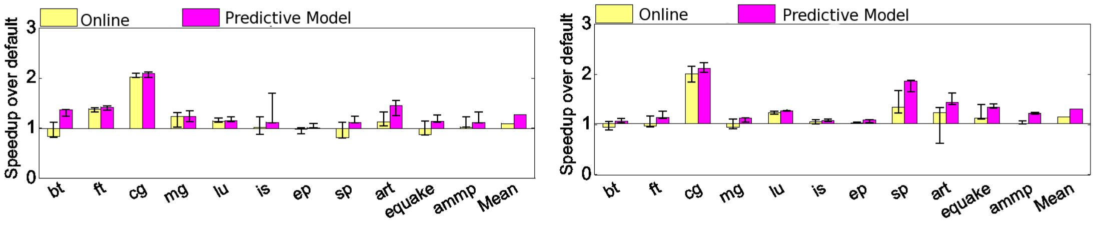
tion approach. Overall, the automatic approach achieves a geometric mean speedup of 2.3x. This translates to a 1.5 times improvement over the 1.47x speedup achieved by online.

### 4.5.2 Detailed Comparison

Here a detailed comparison of results is presented for each workload setting. The results are averaged across all the experiment runs and are presented on a per-benchmark basis. Although the goal is to maximize the performance of the target program, it is also important to know the impact of the mapping decision on the external workload programs. Ideally any mapping scheme should cause minimal disruption to the external workloads. Hence, the performance of workload programs is also included in this section.

#### Light workload

**Target:** Figure 4.10 shows the performance of the target program for low and high frequency workload arrival rates. For the *low frequency* arrival rate setting (Figure 4.10 (a)) predictive modelling based approach achieves a geometric mean speedup of 1.32, which outperforms online adaptation by a factor of 1.2. However, the range of improved speedup varies with the nature of the target program. For `ep` that scales very well on a multi-core, default scheme performs well and performance improvement obtained by both approaches is small. For benchmark `mg`, the online approach performs poorly by slowing down the program to 79% of the default performance. This is be-



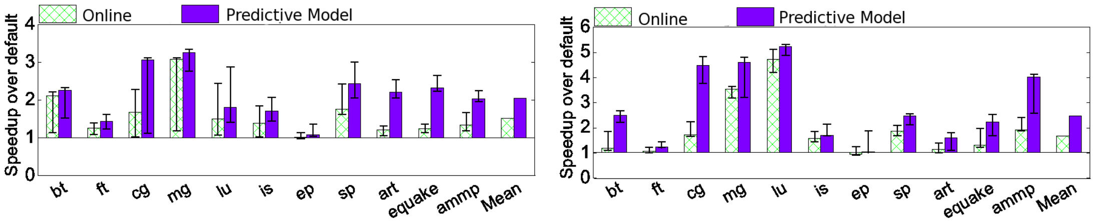
(a) Low frequency workload arrival rate

(b) High frequency workload arrival rate

Figure 4.11: Speedups of associated workloads with *standard deviation* in *light* workload settings. This approach achieves a mean speedup of 1.27 and 1.31 over OpenMP default for (a) low and (b) high frequency arrival rates respectively.

cause mg consists of many short-run parallel sections which do not provide enough time allowing the online approach to find the optimal number from its starting point. For the *high frequency* arrival rate setting (Figure 4.10 (b)), this approach achieves a geometric mean speedup of 1.82, leading to 1.5 times improvement over online technique. For benchmarks cg and sp, significant room for improvement was observed over OpenMP default and this approach achieves a speedup over 3.9. For other programs, such as bt, ep and art the room for improvement is small.

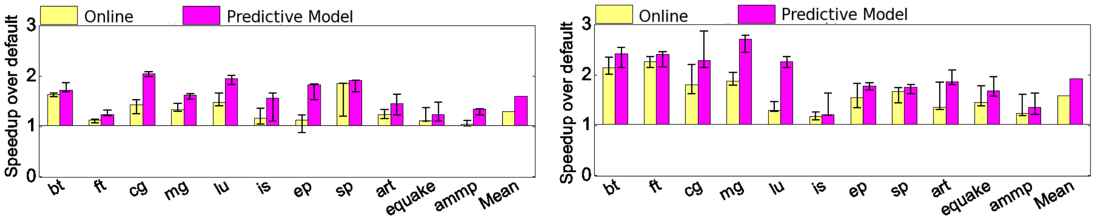
**Workload:** Figure 4.11 shows the performance of the workload programs when they run along with the target program on the x-axis. For example, the first bar in this figure shows the performance of combined workloads co-executing with target bt. In the *low frequency* arrival rate setting (Figure 4.11 (a)), the online scheme and the heuristic-based approach achieve a mean speedup of 1.42 and 1.65 respectively. Both approaches are able to improve benchmark cg as well as its associated workloads with workload speedups above 2.0. For other target programs, such as ep, is and ammp, there is little room for improvement. This approach does not slow down any workload programs when optimizing the target program. This is contrast to the online approach which causes disruption to the associated workloads by *over-saturating* the system. In the *high frequency* arrival rate setting (Figure 4.11 (b)), similar performance improvement (as the low frequency setting) was observed. It is important to not slow down the workload programs during this remapping. This approach is able to do so while online gives poor performance for the workloads associated with those benchmarks by over saturating the system. In this particular setting, heuristic-based approach delivers a mean speedup of 1.31 for workload programs, leading to 114% of improvement over online.



(a) Low frequency workload arrival rate

(b) High frequency workload arrival rate

Figure 4.12: Speedups of target programs with *standard deviation* for *medium* workloads. The heuristic approach achieves a mean speedup of 2.05 (vs. 1.52 of online) and 2.47 (vs. 1.68 of online) over OpenMP default for (a) low and (b) high frequency arrival rates respectively.



(a) Low frequency workload arrival rate

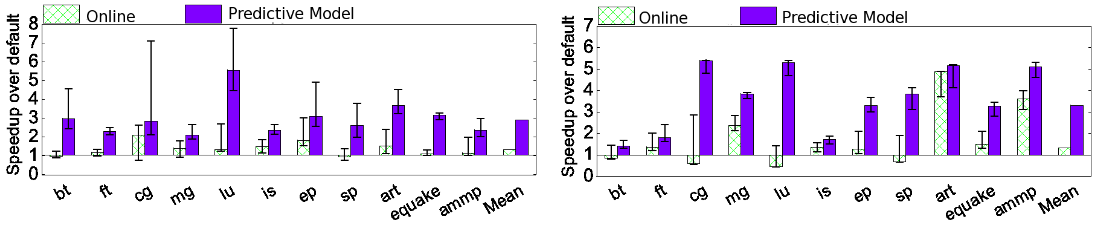
(b) High frequency workload arrival rate

Figure 4.13: Speedups of associated workloads with *standard deviation* in *medium* workload settings. This approach surprisingly improves the performance of the workload programs and obtains a mean speedup of 1.65 and 1.31 for (a) low and (b) high frequency arrival rates respectively.

### Medium workload

Under a medium workload scenario, workload programs create moderate contention for resources. The performance of the default scheme can be further improved for both the target program and the associated workloads, as shown in Figures 4.12 and 4.13.

**Target:** Significant performance improvement over the OpenMP default was observed under medium workload scenarios. The online achieves a mean speedup of 1.52 and 1.68 for the low and high frequency arrival rate settings respectively. Once again, heuristic-based approach outperforms the hill climbing technique with a mean speedup of 2.05 and 2.47 for the low and high frequency arrival rate settings respectively. For benchmark `mg`, hill climbing performs better here than it does for the light workload setting, because the optimal number of threads for `mg` for this particular set-



(a) Low frequency workload arrival rate

(b) High frequency workload arrival rate

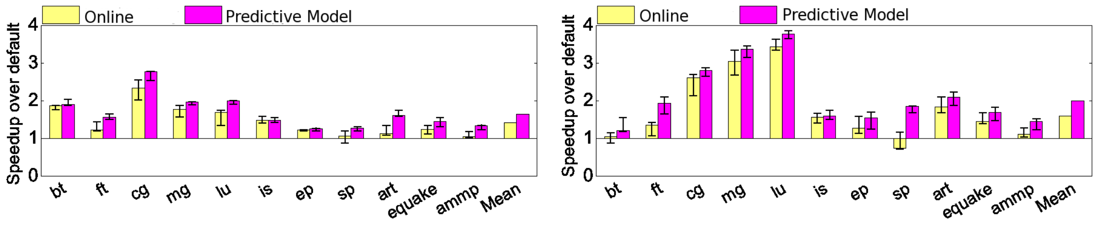
Figure 4.14: Speedups of target programs with *standard deviation* for *heavy* workload. This approach achieves a mean speedup of 2.9 (vs. 1.33 of online) and 3.2 (vs. 1.59 of online) over the OpenMP default for (a) low and (b) high frequency arrival rates respectively.

ting is close to the starting thread count picked by online. Hence, it can reach the optimal thread configuration quickly. Once again, heuristic-based approach outperforms the online approach across all target programs. This is particular for benchmark cg where the performance improvement is 1.4x and 2.6x for the low and high frequency workload arrival rates respectively. This result is due to the fact that the automatic approach directly predicts the optimal thread number instead of profiling the target program with different thread configurations.

**Workload:** As observed from Figure 4.13, performance of the associated workloads can also be improved along with the target program which is more significant in a setting with a high frequency workload arrival rate (Figure 4.13 (b)). This is due to the fact that the default scheme tends to cause harmful program contention by over subscribing hardware resources. Dynamic runtime schemes, on the other hand, achieve a better performance by eliminating the contention. Overall, online scheme is able to improve the performance of workload programs over the default scheme with a mean speedup of 1.30 and 1.58 for the low and high frequency arrival rate settings respectively. The heuristic approach outperforms online with a mean speedup of 1.60 and 1.92 for the low and high frequency arrival rate settings respectively.

### Heavy workload

A heavy workload scenario depicts a realistic picture of many-core systems, data centers and more where multiple programs execute together creating extreme competition for resources. There is enormous scope for the dynamic scheme to improve over the



(a) Low frequency workload arrival rate

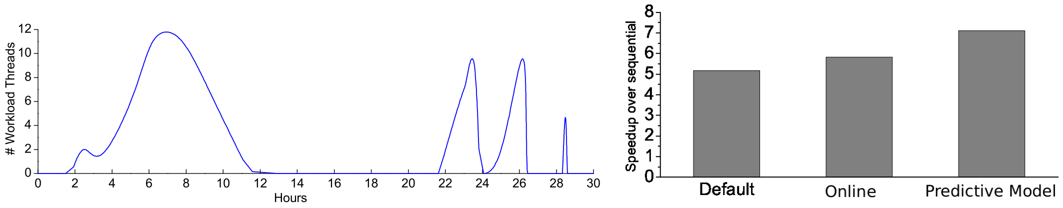
(b) High frequency workload arrival rate

Figure 4.15: Speedups of associated workloads with *standard deviation* in *heavy* workload settings. The heuristic approach improves the performance of the workload programs and achieves a mean speedup of 1.61 and 1.92 over the OpenMP default for (a) low and (b) high frequency arrival rates respectively.

static approach.

**Target:** As observed the graphs in Figures 4.14 (a) and (b) heuristic-based approach achieves a geometric mean speedup of 2.9x and 3.2x for low and high frequency workloads respectively. For heavy workloads, the default scheme over-saturates the system adding to the contention caused by the workloads. This approach eliminates this contention and achieves improvement as it selects the optimal thread number carefullysuited for this heavy workload environment. The online method achieves a mean speedup of 1.33x and 1.59x which suffers mostly due to the time spent to reach optimal level. This approach obtains high speedup improvement for `lu, equake` where the online approach only manages to obtain a marginal speedup. In the high frequency workload settings, the online approach fails to adapt to the workloads and worsens performance of programs like `bt, sp, lu,` whereas the ML-based approach achieves significant improvement for these programs.

**Workload:** The heuristic approach achieves an improvement in workload performance as well due to remapping strategy leading to reduced contention. From Figure 4.15, the heuristic-based approach improves workload performance by 1.69x, 2.12x times in low and high frequency settings. Hill climbing approach improves workload performance by 1.46x and 1.76x times in low, high frequency workloads. This translates to 1.15x, 1.20x times improvement over online approach. Both approaches provide fair allocation of processors to workloads by which they execute faster by reducing the congestion in the system specially for programs `cg, lu` and `mg`.



(a) Real world workload pattern

(b) Evaluation on real world workload pattern

Figure 4.16: (a) Workload distribution from a real world system. (b) Performance of the predictive model with live system workload.

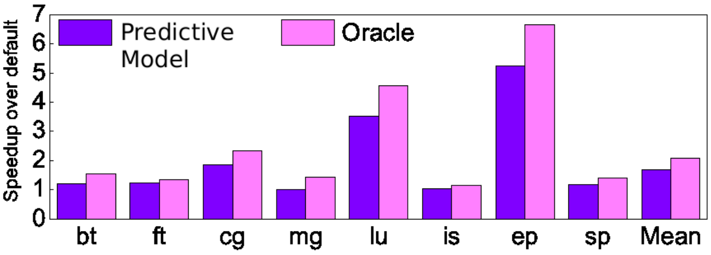


Figure 4.17: Speedups of the heuristic approach and the oracle on the training data. This model has a high accuracy by achieving 81% of the oracle performance.

### 4.5.3 Case Study

To validate the the proposed approach in a real world setting, a workload environment is selected based on a sample of a high performance cluster of computing systems as shown in Figure 1.1. The distribution of the arrival of jobs and the number of requested processors over a period of 30 hours. This pattern replicated on a small scale experiment on 12-core system can be observed in Figure 4.16(a).

Over this workload scenario, Figure 4.16(b) shows the speedup averaged across all programs with different schemes. It can be observed that the predictive model fares better than the default and the online technique by 1.37 and 1.22 times performance improvement. This clearly shows that this model adapts well with the dynamic external workload programs in any computing environment.

### 4.5.4 Oracle Study

The efficiency of any predictive model depends on its prediction accuracy. Hence it is desired to have maximum prediction accuracy to predict the outcome close to the oracle. As it is impossible to collect the oracle performance in a live system, the prediction

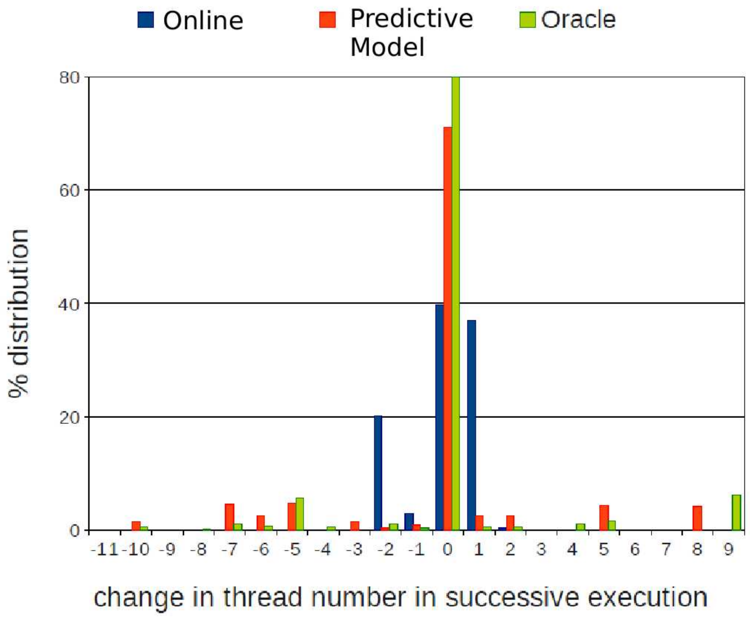


Figure 4.18: Distribution of thread number change in successive calls to runtime library.

accuracy of this model is evaluated using the collected training data where the oracle performance is known. Figure 4.17 compares this model against the oracle. The baseline is the default scheme. As can be seen from this diagram, the resulted performance of this model is very close to the oracle. For benchmarks `ep` and `lu`, there is room for improvement. This can be enhanced by additional training benchmarks. On average, the predictive model achieves a speedup of 1.66, which translates to 81% of the oracle speedup of 2.06. This confirms that this model has a highly accurate prediction result.

## 4.6 Analysis

From the previous sections, it can be observed that the predictive modelling-based approach outperforms state-of-the-art technique across various workload scenarios. This section gives a detailed insight into the source of performance improvement.

### 4.6.1 Insight into Performance Improvement

The performance gain from this approach can be accounted for

(a) **changing the thread number directly when needed and without delay**: In the presence of workloads, a direct change in the number of threads of target program may be necessary in order to reduce the contention. Ideally an approach has to react

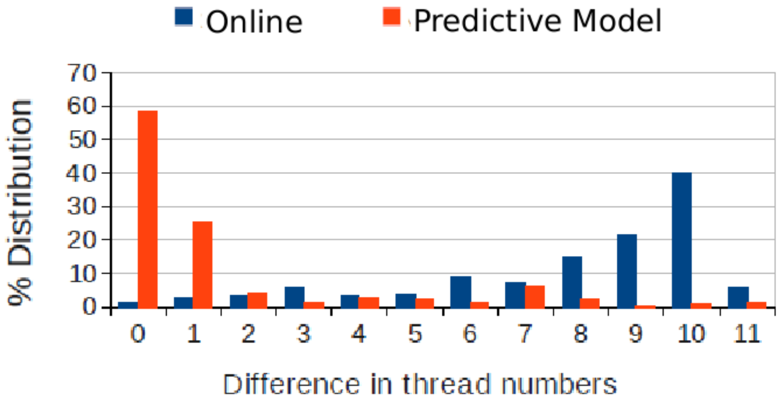


Figure 4.19: Distribution of difference in thread numbers between the online scheme and the predictive model approach relative to oracle.

quickly to this dynamic environment. The online technique changes thread numbers in incremental steps wasting much of the critical execution time trying to in reaching the optimal thread number. The heuristic approach reacts quickly to the workload and changes directly to the optimal thread number as and when required.

Figure 4.18 shows the difference in thread numbers between successive calls to the OpenMP runtime library. The oracle stays at the same thread number 80% of the time and there are several instances where sharp change in thread number (-5, +9) is essential for optimal performance. Closely following the oracle, the heuristic approach retains the same thread number at around 70% of the time and there are significant instances where there is a direct thread number change (-7,-5,+5,+8). The online scheme stays at the same thread number just 40% of the time and confines only to incremental changes in thread number. However the frequency for no change in the thread number is high owing to highly regular structure of the chosen programs.

(b) **Determining accurately a near optimal thread number**: An ideal approach would essentially assign thread numbers close to that of the oracle. The smaller the difference, the better will be the performance. Figure 4.19 shows the difference in thread numbers between (i) oracle and the heuristic approach (ii) oracle and the online technique. It can be observed that the heuristic model predicts thread numbers exactly as the oracle around 60% of the time, whereas the online technique rarely assigns thread numbers the same as that of the oracle, just around 1.3% of the time.

### 4.6.2 OS support for thread placement

In the Linux kernel version used, threads are assigned to a core based on the existing load on that core. When processes are spread across multiple sockets, the scheduler assigns a ready thread to a core with the lowest load. This default approach can be changed by setting the thread affinity parameter. However, the recent kernel versions handle this affinity based scheduling more efficiently. Evaluating this work on the latest kernel is a part of future work.

## 4.7 Summary

This chapter has presented a predictive modelling-based solution to determine the best mapping of an application in the presence of dynamic workloads. The proposed thread-predictor model brings together static compiler knowledge of the program and dynamic runtime information to reconfigure and optimize an application in a dynamic environment. It aims to maximize performance of the target program with minimum impact on the external workloads. This approach provides a significant performance improvement over existing techniques. On a 12-core Intel Xeon platform, this approach provides a significant performance improvement over the state-of-the-art online adaptation approach by a factor of 1.5x while having no impact on the external workloads.

The next chapter discusses what happens to this mapping policy when the execution environment changes drastically during program execution. It proposes a novel approach to exploit the proposed thread-predictor and adapt it online based on the changes in the execution environment.

# Chapter 5

# CDMapp: Change Detection-based Parallelism Mapping

This chapter presents a novel technique to improve parallelism mapping when the execution environment changes drastically at runtime. Here, the environment constitutes the dynamic parameters of the execution platform that may influence a program's execution. It includes any co-executing workloads, hardware, input data, software, etc.

Such a change invalidates offline models as they are not trained in the new environment. Here a target program executes with varying workloads when the hardware changes drastically. The proposed approach uses the offline thread-predictor model described in Chapter 4. A new machine learning model '*environment-predictor*' is developed which is used to evaluate the thread-predictor online and also detect any changes in the system. This enables the thread predictor to adjust its outcomes, if required, thus adapting to changes in the execution environment.

The chapter begins with an introduction in Section 5.1. Section 5.2 highlights the motivation showing how existing approaches lag in reaching the optimal state, along with an example. The proposed approach is described in Section 5.3 followed by the evaluation methodology in Section 5.5. Section 5.6 discusses experimental results and subsequent analysis in Section 5.7. This chapter concludes in Section 5.8.

## 5.1 Introduction

Modern day computing environments typically co-schedule multiple jobs to improve system utilization. Minimizing program interference and effective exploitation of resources are crucial to improving program performance. Traditional approaches for dynamic allocation of resources to programs or the mapping of parallel work to underlying hardware can roughly be categorized as ***offline*** or ***online*** that are described in Chapter 1. The key problem using these approaches is lack of mechanisms to detect a change in the environment and monitor the quality of mapping decisions at runtime. There is no obvious “system utilisation gauge” to refer to.

To overcome this problem, an approach is required that can exploit prior offline knowledge and discover online when this model needs to be updated. This chapter develops such an approach **CDMapp**, for efficient parallelism mapping based on *online change detection* [Basseville and Nikiforov 1993]. In Chapter 4 a model is learnt offline that considers both program characteristics and runtime workload to determine the best runtime mapping. However, such an approach depends critically on the coverage of its training set. If the online environment radically departs from the training assumptions, the model is no longer applicable and will make wrong decisions. Furthermore, it is unable to detect that it is wrong and update its policy.

The proposed solution aims to exploit best of both worlds, using the knowledge gained offline encapsulated in a model and reacting to changes in the environment that invalidate the model. Moreover, runtime scheduling is described formally as a *Markov Decision Process (MDP)*. The main contribution of this solution is to build a second model *environment-predictor* that predicts what the environment should look like if an ideal mapping decision had been made. In this work, the environment is formalized as the norm of the dynamic feature vector. (See Table 5.1). Intuitively, a large value of the environment indicates higher system load due to increased resource contention and vice versa. At the next decision point the accuracy of this prediction is measured. If there is a large discrepancy, this means that the assumptions used offline no longer hold true and the mapping should be adjusted accordingly.

This approach is shown in Figure 5.1 where at every loop $L_i$, both the best thread number is predicted as well as what the environment should look like. At the next loop, if the difference between actual and predicted environments is smaller than a threshold ‘δ’, it can be assumed the environment is behaving as expected and the outcome of thread number prediction is unchanged. If the environment is substantially different from expected ($>\delta$), it might be that the environment has now changed drastically and the number of threads are adjusted accordingly. The value of the threshold estimated experimentally is described later.

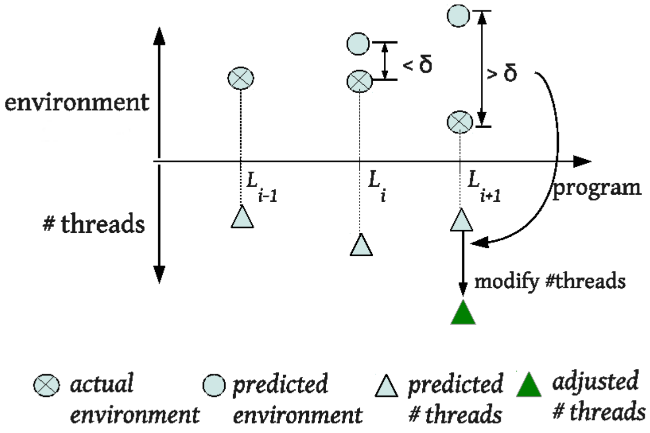


Figure 5.1: Using environment-predictor to adjust the outcome of the thread-predictor at every decision point.

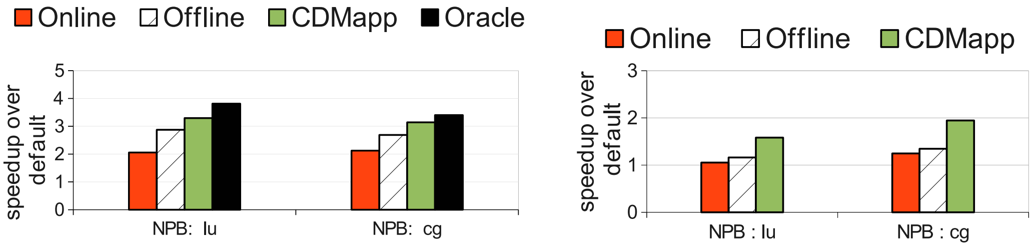
## 5.2 Motivation

This section provides an example to illustrate that determining the best thread number for a target program in a rapid dynamic environment is non-trivial. It also highlights that any mapping technique needs to adjust rapidly and accurately to the changes in the environment.

### 5.2.1 Scope for Improvement

To show there is scope for performance improvement over existing state-of-the-art schemes, two experiments are run on a 12-core system as described in Section 4.4 with `lu`, `cg` as targets from NAS benchmark suite. In each experiment, each target is co-executed with just one external workload `mg` from same benchmark suite that employs OpenMP default scheme. The first experiment is in a controlled environment with a static workload and hardware. The second is in a more realistic dynamic environment with varying workload and hardware. This approach is compared against the OpenMP default, online, and an offline trained approach (thread-predictor) as described in the previous chapter and an idealised oracle.

Since the offline model is trained in a similar execution scenario to the test experiment, it outperforms the default and online approaches. Here an idealised oracle scheme is also evaluated to determine the best achievable speedup. As seen from Figure 5.2(a), CDMapp performs better than all existing techniques, however, there is still



(a) controlled environment

(b) dynamic environment

Figure 5.2: Room for improvement in (a) static controlled system and (b) realistic dynamic system.

a scope for an average 13% improvement to attain the best speedup.

In a dynamic system, it is not possible to determine the best mapping at runtime owing to a large number of thread numbers to evaluate at runtime. As observed from Figure 5.2(b), CDMapp improves up to 40% over the offline scheme. This shows that there is a significant room for speedup improvement in a dynamic system even when the best scheme is unknown. This experiment also shows that an offline trained model is limited by the training environment. If the execution environment changes drastically from the one in which it was trained, then it is unable to determine the optimal thread number.

Figure 5.3 illustrates this improvement. It shows a timeline view of the thread numbers selected by each scheme in the second experiment where `lu` executes with `mg` on a 16-core system. At the $20^{th}$ second, the number of processors drops to 4. The online approach takes time to adjust as observed in the large number of peaks after time t=20sec. Both offline and CDMapp approaches predict nearly the same thread numbers initially. After a change in the system, the offline approach is unable to accurately predict the right thread number as this model in now out-of-date. This pattern can be observed where more peaks are observed for offline scheme after t=20sec. CDMapp adapts quickly to this new system as explained in Section 5.4.4 and can predict the right thread number with a fewer thread changes as shown in the timeline figure.

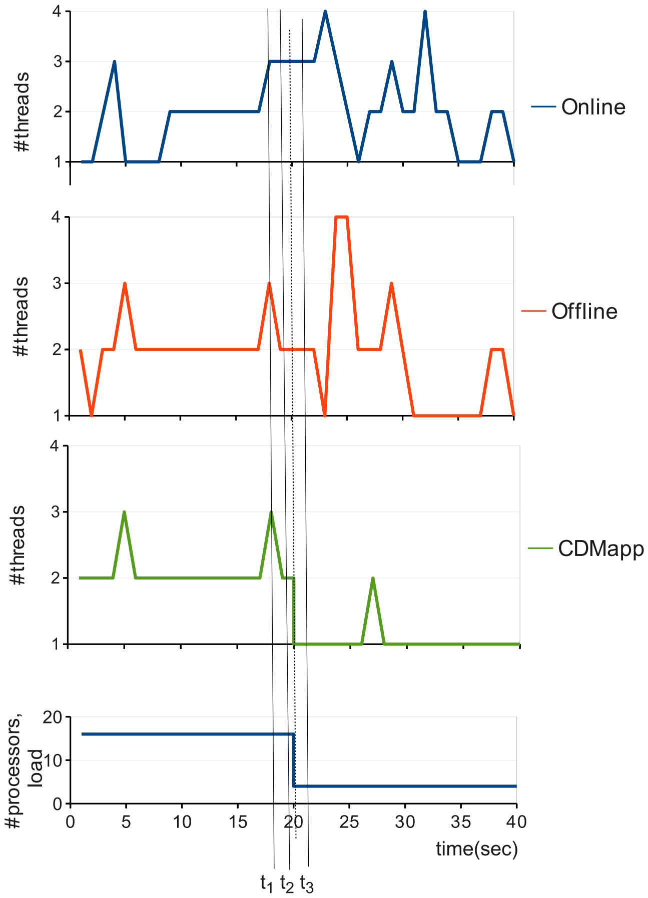


Figure 5.3: A timeline view of the number of threads selected by various schemes in a dynamic system when the hardware changes at time=20sec. Analysis of CDMapp at $t_1$ to $t_3$ is presented in Section 5.4.4.

## 5.3 Scheduling as a Markov Decision Process

As discussed in 2.4, a process is *Markovian* if the next state depends only on the current state and current action; it is independent of previous actions as these are captured by the current state. Runtime scheduling of parallel code readily fits this description where each scheduling decision point corresponds to a time step. State $S$ corresponds to the current state in the program execution. A scheduling decision corresponds to an action $A$, the result of which will change the current state. It is a probabilistic change as the exact impact of a scheduling decision on the state of the program or environment is not known. Runtime scheduling is *Markovian* as deciding the best action just depends on the current parallel loop and the environment. Previous actions will affect the current state; however there is no need to record the previous actions explicitly as they are completely captured by the current state. All scheduling policies want to minimise the execution time of the program which corresponds to the reward $R$ and how they attempt this forms their policy $\pi$.

**Finding best policy:** Current scheduling actions have different values $V$ and will affect the future state; which in turn will affect future actions, and so forth. Policy depends on reward and reward depends on policy. This recursive relation between reward and policy is normally solved using an iterative method to give a policy $\pi$, which defines the best action given any state. However, such mapping approach is not feasible for runtime scheduling. Firstly, the exhaustive state space is large corresponding to all possible states a parallel machine could be in, given any mix of target and workload programs. Furthermore, the possible value of an action varies from state to state. As the state space is enormous, direct solution to an MDP is not feasible. So the two main hurdles are massive state space and reward being highly dependent on state. All runtime scheduling policies, therefore, make simplifying assumptions to deliver a policy.

The next section formulates each evaluated mapping policy as a Markov Decision Process. It highlights how state transition probabilities can be used for environment change detection and improve scheduling.

**Mapping Policies as MDPs:** Here each evaluated policy is described in terms of an MDP. Figure 5.4 summarises these policies. Let $\underline{c_i}$ denote the program code of $i^{th}$ parallel loop, $\underline{e_i}$, the corresponding runtime environment, $\underline{f_i} = \underline{c_i}||\underline{e_i}$, the features combining code and environment information and $n_i$, the number of threads at loop $i$. The notation '$\wedge$' is used for a prediction and $\pi_{policy}$ denotes the rule of a scheduling policy. So $\underline{\hat{e}_{i+1}}$ is the predicted environment at the next parallel section and $\hat{n}_i$ is the predicted best number of threads. $\hat{n}_i$, $\underline{\hat{f}_i}$ represent predicted thread number and predicted environment at loop $i$ respectively. In this work, a time stamp corresponds to a decision point.

**Default:** OpenMP default policy assigns a thread number equal to the maximum number of available processors. In this approach, the only state $S$ modelled is the

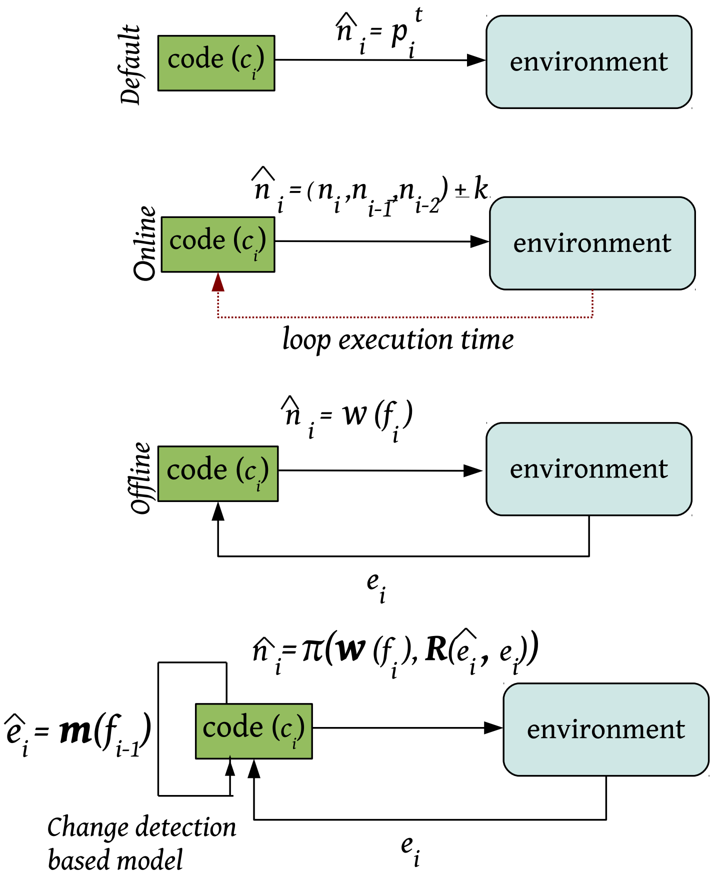


Figure 5.4: MDP formulation of various runtime scheduling approaches.

number of processors available at time $t$, $p^t$.

$$S = \{p^t\}$$

There is only one action $A$ available, to set the number of threads to the number of currently available processors.

$$A = \{n^t = p^t\}$$

The reward of an action $R$, the total reward $V$ and the state transition probability $P$ are ignored. This gives a simple policy to set the thread number to the current number of processors.

$$\pi_{default}(s^t) = a^t$$

**Online:** Here state is modelled as dynamic points in the code where decisions are made.

$$S = \{(s_i^t)\}$$

where $i$ refers to the parallel code section and $t$ refers to the of its dynamic instance of that parallel section. The actions are setting the thread number up to the current number of processors.

$$A = \{n_i^t = 1, \ldots, p_i^t\}$$

The reward of an action is explicitly recorded as $R(s_i^t)$. This is the time taken for a particular code section $i$ at dynamic instance $t$. The state transition probability $P$ is ignored. The policy explicitly considers the value of a particular code section $i$ at previous time stamps, $t-1$, $t-2$ before changing as:

$$\pi_{online}(s_i^t) : n_i^t = \begin{cases} j, n_i^t \% 3 = 0 \\ n_i^{t-1} + 1, n_i^t \% 3 = 1 \\ n_i^{t-2} - 2, n_i^t \% 3 = 2 \end{cases}$$

where $j = argmax_{j=0,1,2} R(s_{i-j}^t)$.

In other words, set the number of threads to the best time seen in the last two dynamic instances of this parallel section every three instances. Otherwise vary the number of threads by $+1$/-2.

**Offline:** The offline approach explicitly considers the state of the program and the environment in terms of features.

$$S = \{(\underline{f}_i^t = [\underline{c}_i^t, \underline{e}_i^t]\}$$

where $\underline{c}_i^t$ are the static code features of the loop $i$ to be scheduled, $\underline{e}_i^t$ are the current dynamic environment features, which are combined to given $\underline{f}_i^t$. The actions are the same as the online approach, setting the thread number up to the maximum number of current processors.

$$A = \{n_i^t = 1, \ldots, p_i^t\}$$

The reward $R$ for a particular action $a$ is the time taken for a parallel loop and is explicitly recorded during offline training as is the total reward $R$. The state transition probability $P$ is not explicitly considered but is assumed during training. Instead of using value iteration to determine a policy, a model $x$ is learnt that given an action $a$ and a state $s = \underline{f}$ returns an approximation of eventual value.

$$x(a, \underline{f}) = \hat{V}(a)$$

The policy then simply selects the action that maximizes this value:

$$\pi_{offline}(\underline{f}) : n = w(\underline{f} = (argmax_a(x(a, \underline{f}))))$$

The details of how such a model is learnt are described in Chapter 4. This policy is learnt offline and applied dynamically at runtime. There is no change or relearning of policy at runtime.

## 5.4 CDMapp

The proposed solution CDMapp is now presented. It exploits a model that has been learnt offline to dynamically map programs in the presence of external workload. It then improves the mapping quality by detecting any environment change, adjusting its behaviour over time.

**Predictive modelling:** At the heart of CDMapp is predictive modelling-based approach that relies on *two* models: (a) *thread-predictor* '*w*' that predicts the optimal thread number and (b) *environment-predictor* '*m*' that predicts the ensuing runtime environment after selecting the optimal number of threads, both based on the program characteristics and the current runtime environment.

The accuracy of the environment-predictor is easier to evaluate than the thread-predictor at runtime. This is due to the fact that accuracy of environment-predictor can be computed based on the difference between the actual and predicted environment (two scalar values). However, to evaluate the thread-predictor accuracy, the same parallel section needs to be run with all thread numbers and then compared with the predicted thread number. Such process is highly expensive to execute at runtime. The accuracy of the environment-predictor reflects the impact on system load by the predicted thread number. In other words, it acts a proxy to judge the decisions of the thread-predictor.

These models are trained and built offline using a three-step supervised learning algorithm as discussed in Section 2.3.1. This training process automatically generates the heuristic that is portable to other hardware platforms.

### 5.4.1 CDMapp as MDP

The proposed approach, CDMapp, based on '*change detection*' has the same state description $S$ as offline approach and also builds a function $w(a, \underline{f})$ to predict the reward of a particular action. It differs in that it explicitly considers the state transition function $P(s, s', a)$ and modifies its policy online based on the transition. It builds a *state predictor* '*m*' offline

$$m(\underline{f_t}) = \underline{\hat{e_{t+1}}}$$

that predicts the environment at the next time stamp. If this prediction is incorrect then $P(s, s', a)$ has changed and the offline policy $\pi_{offline}(a)$ is out-of-date. This is known as change detection. Now since the thread predictor model is invalidated, its policy is now updated with a proportion ($k$) of the observed difference, to reflect this change. Here the value of $k$ is observed to be 1, however, it can be application or platform dependent. The CDMapp policy can be stated as:

$$\pi_{CDMapp}(s) = \begin{cases} \pi_{offline} & \text{if } \|\underline{e}_t - \underline{\hat{e}}_t\| \leq \delta \\ \pi_{offline} \pm k\|\underline{e}_t - \underline{\hat{e}}_t\| & \text{otherwise} \end{cases}$$

### 5.4.2 Machine Learning Model Generation

Here, a description of the predictive models is presented starting with training data generation, feature collection and building the models. The set of features collected in Chapter 4 are reused here.

**Training data:** The training data is obtained by running each training program in the presence of external load with varying number of threads as described in Section 4.3. The code structure is captured by the static features $\underline{c}$ and environment behaviour $\underline{e}$ in a set of features $\underline{f}$. These features characterize the program structure and the environment state. The best number of threads that gave best overall performance are recorded along with retaining the environment features that resulted from selecting this best thread number.

**Features:** The set of features used for building the thread-predictor is reused for learning the new models ( the feature set is the same as the one in Section 4.3.1). The collected features $f^1, .., f^8$ are listed in Table 5.1. At loop $i$, the feature vector $\underline{f_i}$

| Static features | Dynamic features |
|---|---|
| $f^1$: Load/Store count *(lscount)* | $f^4$: Number of workload threads *(wthreads)* |
| $f^2$: Branch count *(branches)* | $f^5$: Number of available processors *(numproc)* |
| $f^3$: Instruction count *(instcount)* | $f^6$: Runtime queue size *(runq)* |
| | $f^7, f^8$: Load *(ldavg-1, ldavg-5)* |

Table 5.1: List of features.

is formed by these features. The parameters in this feature vector are plain numbers by default (for example, number of instructions, number of processors, load values). If additional feature values are associated with units of measurement, then they could be normalized to a fixed unit and can be included in the feature-vector.

Figure 5.5 shows the feature impact (see Section 2.3.4) across both models. It can be observed that static and dynamic features are equally important in their contribution to the quality of trained model predictions.

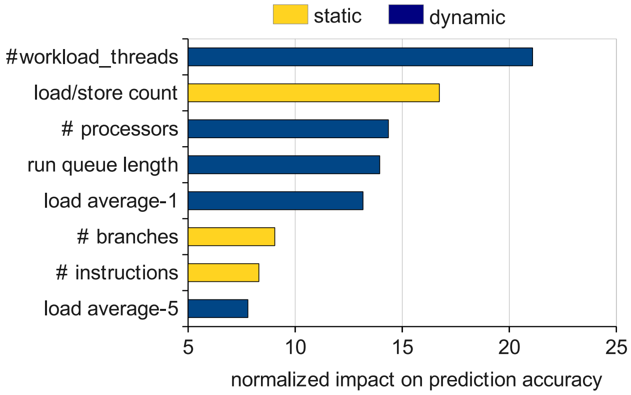


Figure 5.5: Impact of selected features on the predictive models. The static and dynamic features are equally important for the accuracy of the models.

**Building the model:** Both predictive models are built using the standard linear regression technique employing least squares method as described in Section 2.3. These two models are: $n_i = \underline{w}\underline{f}_i$ and $\|\hat{e_{i+1}}\| = \underline{m}\underline{f}_i$ where the first model (thread-predictor) is used to predict the thread number, the second model (environment-predictor) is used to determine a future environment (norm of the feature vector). Learning a model for the collected training data is simply finding the best linear fit to the data. This learning results in two simple 8-dimensional linear models $w$ and $m$ with their coefficients listed

| Features | $w$ | $m$ | Features | $w$ | $m$ |
|---|---|---|---|---|---|
| $f^1$: *lscount* | 0.104 | 0.213 | $f^6$: *runq* | 0.351 | 1.190 |
| $f^2$: *branches* | 0.612 | -0.420 | $f^7$: *load1* | -0.166 | -0.049 |
| $f^3$: *instcount* | -1.323 | 1.091 | $f^8$: *load5* | 0.158 | 0.313 |
| $f^4$: *wthreads* | -1.198 | -0.124 | *constant* | -1.443 | 0.69 |
| $f^5$: *num_proc* | 0.956 | 0.872 | | | |

Table 5.2: Regression coefficients.

in Table 5.2. Unlike an ANN used to learn a model in Chapter 4, regression technique was used here due to higher prediction accuracies of both models.

### 5.4.3 Deployment

The built predictive models are combined and deployed at runtime as shown in Figure 5.6. Once the compiler has extracted static code features, it links this compiled version to a runtime library. At execution time, the runtime system features are collected and fed as input to both the models. The parallel section executes with the predicted thread number $\hat{n_i}$ and the predicted environment $\widehat{e_{i+1}}$ is used to detect change. The absolute difference between the actual and predicted environment normalized to number of available processors is computed. If this value is below a predetermined threshold δ, it implies that the prediction of the model $w$ was accurate. The value of threshold δ is determined to be 0.5 obtained from a sensitivity analysis study as shown in Figure 5.17(c). If there is a large difference, then these models are inaccurate and need updating. In the thread-predictor the numeric error component is adjusted with a proportion of the difference between observed and predicted environments, $||e_i - \hat{e}_i||$ (here $k$=*1*).

### 5.4.4 Example

To demonstrate how CDMapp approach works, consider the timeline in Figure 5.3 at three different decision points at time-stamps $t_1$, $t_2$, $t_3$. At $t_1$, the feature-vector is $\underline{f}$ =[0.045, 0.013, 0.1, 16, 16, 6, 2.73, 2.17], the number of threads selected is $n_1 = \underline{w} \cdot \underline{f}$ = [0.045, 0.013, 0.1, 16, 16, 6, 2.73, 2.17] · [0.104, 0.612, -1.323, -1.198, 0.956, 0.351, -0.166, 0.158]-1.443 = 3. The environment at the next time step is pre-

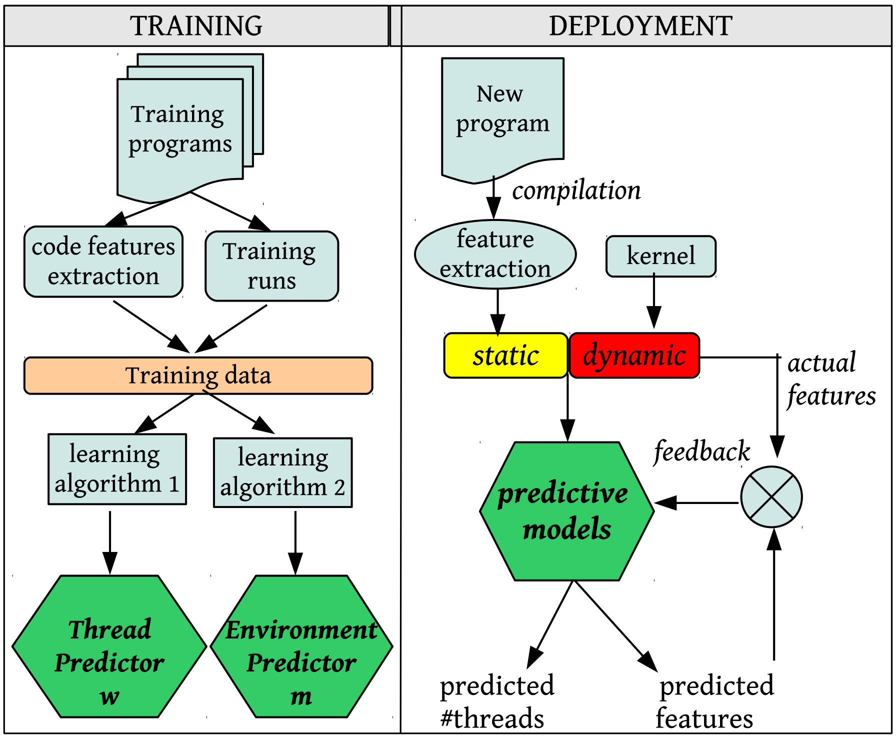


Figure 5.6: Training and deployment phases in CDMapp.

dicted as $\|\hat{e_2}\| = \underline{m} \cdot \underline{f}$ = [0.045, 0.013, 0.1, 16, 16, 6, 2.73, 2.17] · [0.213, -0.42, 1.091, -0.124, 0.872, 1.19, -0.049, 0.313] + 0.69 = 20.72. At the next decision point, $t_2$, the normalized difference between observed environment $\|e_2\|$=25.86 and predicted environment $\|\hat{e_2}\|$=20.72 is $|25.86-20.72|$/16=0.32 which is less than the threshold and hence the predicted thread number $n_2$=2 is unaltered. The environment at the next point is predicted again. When the environment has changed at $t_3$, the feature-vector $\underline{f}$ is now [0.032, 0.026, 0.2, 4, 4, 16, 4.76, 2.17] and the normalized difference between the observed environment $\|e_3\|$=17.75 and the predicted environment $\|\hat{e_3}\|$=22.05 is $|17.75-22.05|$/4=1.07 which is greater than the threshold. This signifies a drastic change in the system. Hence the predicted thread number $n_3$=5 is out-of-date and is now adjusted to result in new thread number $n_{3\_ad\hat{j}usted} = 5 - \lfloor 4.3 \rfloor = 1$.

## 5.5 Experimental Set-up

This section describes the set-up and evaluation methodologies used to perform the experiments.

| Workload type | Benchmark programs |
|---|---|
| light | (i) `is, cg` |
| | (ii) `ammp, ft` |
| | (iii) `ep, art` |
| moderate | (i) `ft, lu, ammp` |
| | (ii) `bscholes, cg, equake, ep` |
| | (iii) `is, mg, ep, art, btrack` |
| massive | (i) `sp, bt, equake, is, cg, art` |
| | (ii) `bscholes, lu, bt, sp, fmine, art, mg` |
| | (iii) `lu, sp, btrack, mg, ep, equake, ft` |

Table 5.3: Workload configuration.

### 5.5.1 Hardware and Software Configurations

All experiments were performed on a dual socket 4-core Intel Xeon E5530 2.40 GHz system with hyper-threading enabled (total 16 threads) running kernel 2.6.18. All programs were compiled using *gcc* 4.6 with -O3 optimization level. To allow direct comparison to prior work, the experimental set-up described in Section 4.4 is replicated for a small scale oracle study, used to evaluate the accuracy of this approach. This machine is a dual socket 12-core Xeon E5620 2.40GHz running kernel 2.6.18.

### 5.5.2 Benchmarks

For the evaluation purpose, 14 OpenMP-based C programs from NAS, SpecOMP and Parsec benchmarks are chosen. Each program runs with largest input dataset. These programs are described in detail in Appendix A.

### 5.5.3 Experimental Scenarios

The proposed approach is evaluated in two types of execution environments: *controlled static* and *dynamic runtime* environments. In a controlled static environment, the external workloads continue running till the target finishes executing. Moreover, there is no change in the hardware where the target and workloads are executing. In a dynamic execution environment, each target program co-executes with varying workload programs and potentially changing number of processors during program execution.

However, the effect of other external system issues such as network contention, software version upgrade etc., are inherent in the set of selected runtime features.

### External Workload

The external workload consists of a number of parallel programs selected from the above 14 programs. The number of workload programs and their number of threads are varied during program execution at runtime. Also, the workload is changed every 2 seconds to have a high variation in the nature of programs. As there are potentially many workload settings, for the purpose of this paper, three types of workloads are identified similar to the classification in Section 4.4. These are *light, moderate* and *massive* classified based on the amount of contention they create in the system.

Assume a workload can be represented as: $W_{type}$ = (p,t) where *p* is the total number of programs in a workload and *t* denotes the total number of threads. If '*P*' is the maximum number of *available* processors, the workloads are classified as (i) $W_{light}$: (<2, *P*/4), (ii) $W_{moderate}$: ([2-5], *P*/2) and (iii) $W_{massive}$: (>5, *P*)

For each workload type, three different sets of benchmarks are considered as shown in Table 5.3. Each set consists of programs chosen from aforementioned programs. All results are averaged over the three different benchmark sets. In order to minimize the amount of noise in the results, each run was repeated ten times. To ensure a fair comparison against different schemes, the same external workload is reproduced when the target is evaluated in all cases.

### Changing Hardware resources

There can be many hardware parameters that can vary during program execution. For this work, the *number of processors* are changed at runtime. This change ensures that the amount of available computing resources varies drastically. To reflect a change in hardware resources at runtime, two settings are defined and evaluated. These are (i) ***proc_drop:*** where each experiment starts with the maximum number of available processors in this evaluation, 16 and drop to 4 to reflect a sudden drop in computing resources and (ii) ***proc_inc:*** where the processor count is increased from 4 to 16 to reflect the expansion in computing resources.

This hardware change is achieved by modifying the processor count by switching '*online*' values (1 = enable, 0 = disable) for each CPU in */proc* filesystem. In the dual-socket machine used for these experiments, all 4 cores (8 processors) on one physical CPU and 2 cores (4 processors) on second physical CPU are disabled. In a realistic system, it is known that the change in the number of processors is less frequent than workload change. Hence, the CPUs are turned on/off only once during each experiment.

### 5.5.4 Methodology

This model is evaluated using standard leave-one-out cross-validation technique described in Section 2.6. From the training set, the program to be evaluated is removed and the heuristic is built based on remaining programs. Thus the model always predicts on an unseen program. This approach is compared against the OpenMP *default*, *Online* and *Offline* policies which are described in Section 5.3.

## 5.6 Experimental Results

This section discusses the experimental results of the evaluation of CDMapp against other schemes. Initially, the approach is evaluated in a dynamic setting and how a dynamic environment affects program performance is examined. The impact of each scheme on the workload is studied. This is followed by a real-world case study and an evaluation of an alternative competitively scheduled approach. Finally, results of a controlled experiment is presented where the best or “oracle” schedule can be determined, that provides a limit study of this approach.

### 5.6.1 Overall Results

Figure 5.7 shows the performance of each scheme relative to the OpenMP default. These summary results are averaged across all benchmark programs and shown for different workload and hardware settings. Here each box and whiskers represents the distribution of speedups across all the target programs with a central line represents the average speedup. The box represents the middle 50% while the whiskers represent the outliers. Light, moderate and massive refer to the workload type. For example, the first box represents the performance achieved by the online scheme in light workload environment. Overall, on average, online scheme improves performance by 1.35x, offline technique achieves a speedup of 1.62x while CDMapp achieves 2.14x improvement over OpenMP default.

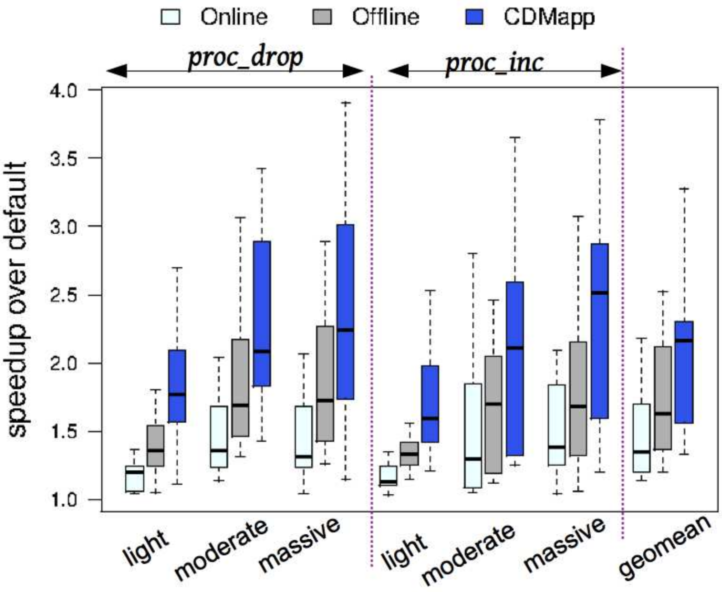


Figure 5.7: Speedup comparison of each scheme per workload and hardware resource scenario averaged across target benchmarks. Overall, on average, online improves performance by 1.35x, offline achieves a speedup of 1.62x while CDMapp achieves 2.14x improvement.

The default policy assigns a number of threads equal to the number of processors, resulting in significant contention due to resource over-provision. The adaptive online technique improves on this by changing the thread number based on the execution time but wastes time trying to find the best thread number. The offline model does better still frequently predicting a good thread number initially but when the hardware changes at runtime, it is unable to adapt. CDMapp adjusts the optimal thread number whenever there is a misprediction or change in hardware. In a *proc_drop* setting, the system contention increases owing to reduced available computation resources and vice-versa. Results presented in subsequent sections are averaged across the evaluations in these two hardware changes. All approaches give a slightly better average performance when the number of processors drops yet as the workload environment becomes more intense, only CDMapp continues to improve. Both the online and offline approaches however do not further improve from moderate to massive workload.

## 5.6.2 Detailed Comparison

Here the performance of CDMapp is examined on a per-benchmark basis across all workload and hardware scenarios. Figure 5.8 shows the distribution of speedups ob-

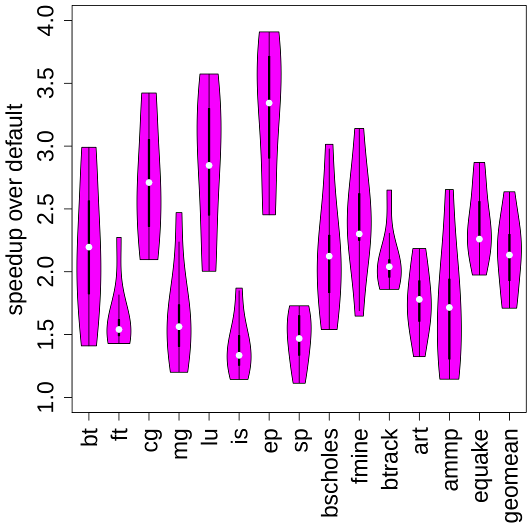


Figure 5.8: CDMapp: Distribution of achieved speedups for all benchmarks across all experimental scenarios.

tained for each target programs across different experiment settings. The central white dot represents the average performance, the black line represents the middle 50% of the distribution. For programs such as mg and sp this improvement factor is relatively low as they spend most of the time in data-access and related memory operations rather than on computation and are strongly affected by external workload. Programs such as cg, lu achieve much greater speedup. Program is is uniformly poor achieving an average speedup of 1.3x due to lack of parallelism, while other benchmarks such as bt have great variation in performance depending on the workload.

Next, these experimental results are preseneted on a per-benchmark basis averaged over three workloads per workload setting and two changing hardware scenarios.

### Light workload

Here the resource contention is initially minimal as shown in Figure 5.9. For certain programs such as cg the online approach is competitive with the offline scheme while in other cases it occasionally slows the program down relative to the default scheme as can be seen by the standard deviation bars on mg,is,sp,bscholes and fmine. The offline scheme approaches the performance of CDMapp on ammp and achieves significant improvement on ep, btrack, equake but in each case it is outperformed by CDMapp which improves programs performance by 1.73x on average.

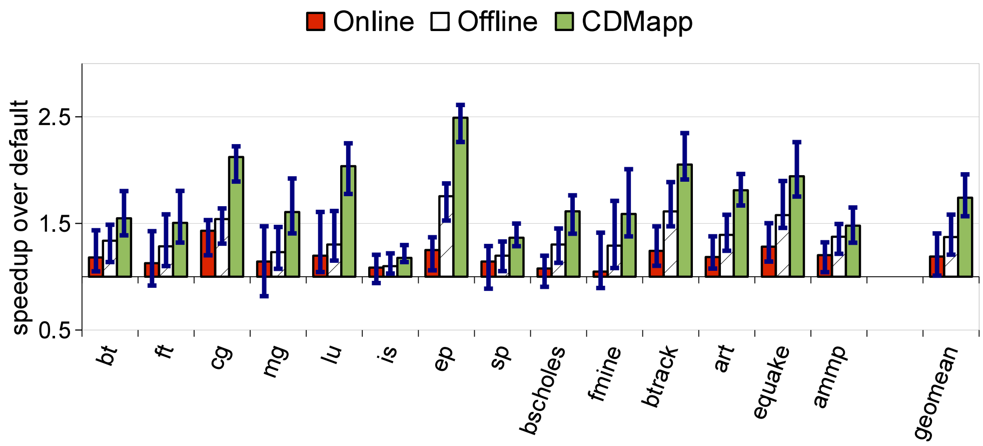


Figure 5.9: For light workloads CDMapp improves performances by 1.74x over OpenMP default, 1.4x over online approach and 1.21x over offline model.

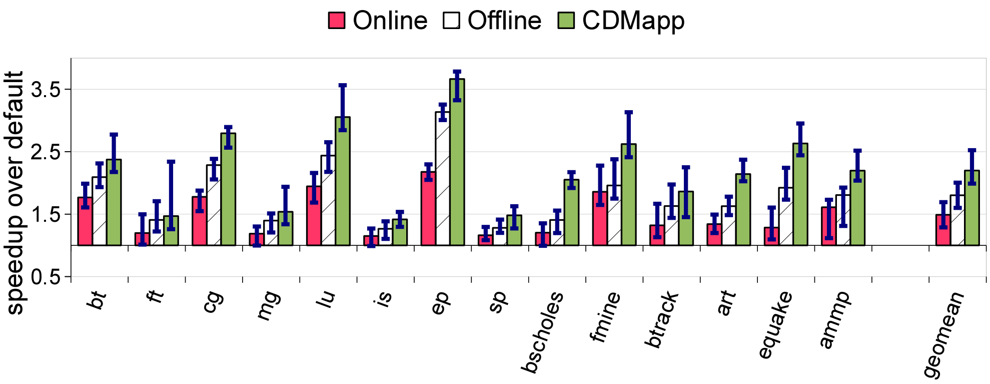


Figure 5.10: For moderate workloads CDMapp approach outperforms by 2.2x over OpenMP default, 1.47x over online approach and 1.22x over offline model.

### Moderate workload

When the system contention is moderate, on average, all the schemes increase program performance relative to the default as they were able to reduce the degree of contention. The online and offline schemes were able to find significant improvement on `bt`,`ep` and `fmine`, but not on `equake`. On average, CDMapp achieves speedup improvement of 2.22x over the default outperforming the other schemes as shown in Figure 5.10.

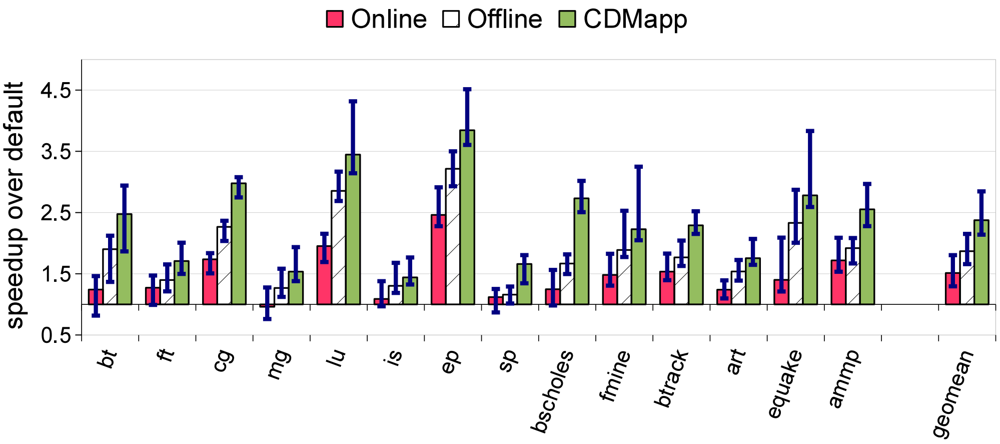


Figure 5.11: With massive workloads CDMapp improves performance 2.37x over OpenMP default, 1.57x over online approach and 1.27x over offline model.

#### Massive workload

Here the target programs are executed along with a massive workload. This high degree of load is to test the system with maximum contention. Once again the online and offline schemes provide good performance improvement on `ep` and the offline approach is also able to show significant improvement on `lu`. However, in certain cases the online scheme has slowdowns relative to the default as shown by the standard deviation bars for `bt, mg, is, sp`. The offline scheme is significantly poorer than CDMapp on `bscholes, ammp` that increases the overall improvement to a 2.37x speedup over the default as observed from Figure 5.11.

### 5.6.3 Impact on Workload

Any optimization technique aiming to improve the target program performance should ideally have minimal impact on external workload. Figure 5.12 plots the impact of various approaches on external co-executing programs. CDMapp reduces system-wide contention that leads to a marginal speedup improvement for external programs. The online approach critically affects the external programs for certain target programs `mg, is, sp` and `art` slowing them down and barely improves workload performance overall. By contrast, both offline and CDMapp improve workload performance by 1.20x and 1.25x respectively. This result - improving the target program improves workload too - is due to reducing overall contention for resources that helps all programs.

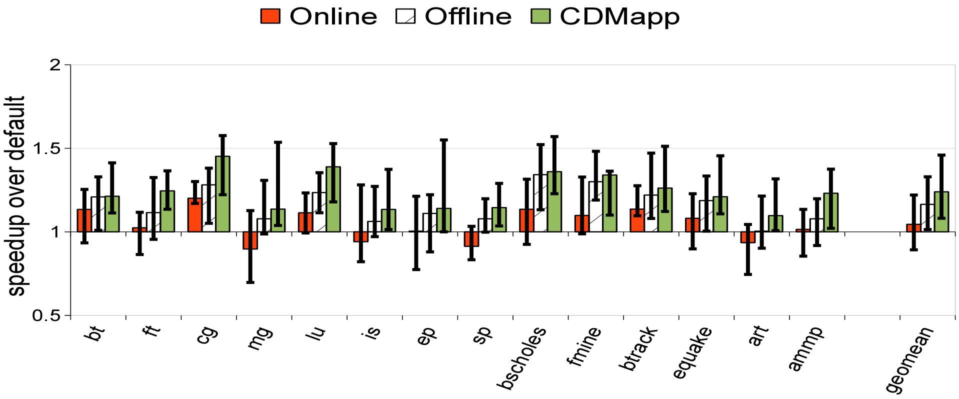
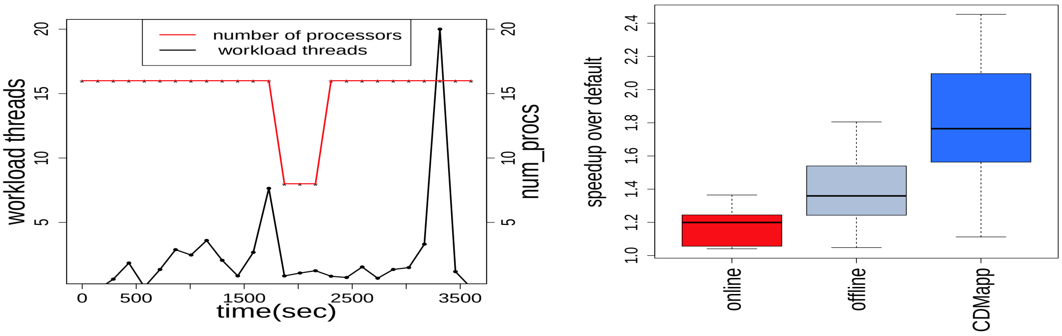


Figure 5.12: Impact of various approaches on the co-executing workload. Speedup greater than 1 implies no impact on the workload.



(a) System Activity

(b) Real-world case study

Figure 5.13: (a) Scaled down real-world system activity showing workload and hardware change patterns. (b) In this real-world case study, CDMapp improves 1.67x over the online and 1.48x over the offline models.

### 5.6.4 Case Study

To demonstrate how CDMapp works in a real live system, a small-scale study is performed similar to the methodology as discussed in 4.5.3. The workload pattern is derived from a log over a period of 50 hours as shown in Figure 1.1. During this period, there was a hardware failure due to which half of the processors were not available for 2 hours. Activities of both the workloads and hardware changes are scaled-down to a time window of 3600 seconds. The number of workload threads was scaled down in proportion to the maximum number of processors. A new workload program is randomly selected at each instance every 150 seconds corresponding to the sampling rate of the log. The resultant recorded activity is shown in Figure 5.13(a).

Here, CDMapp is clearly a superior policy achieving 1.67x improvement over the online and 1.48x over the offline models as observed from Figure 5.13(b) averaged across all benchmark programs. Significantly the average 1.8x improvement of this scheme is just short of the maximum range of improvement by the offline scheme. This result illustrates that CDMapp technique improves program performance in a highly dynamic unseen execution environment.

### 5.6.5 Runtime Evaluation Techniques

There has been some recent work trying to bridge the gap between the online and offline approaches. In [Ansel et al. 2012], different offline-tuned versions of the program are executed competitively at runtime and the best performing version is selected. This approach has the advantage of starting with good implementations and adapting to the online environment. However, executing multiple versions at runtime adds significant overhead and will have to be redone whenever the environment significantly changes. A small-scale study on representative NAS benchmarks co-executing with light workloads in the *proc_drop* setting resulted in average performance drop of 22% with a worst 54% drop compared to OpenMP default policy as seen from Figure 5.14.

### 5.6.6 Oracle Study

The CDMapp approach improves significantly over existing schemes. In addition, it is also important to know how close is this technique is to the oracle. This section describes a limited study to compare CDMapp with the known best scheme in a controlled environment. Since this is static environment, it is possible to evaluate the best achievable speedup. To determine the best oracle schedule, in a replicated experimental set-up as mentioned in Section 4.4, experiments are run with all possible thread numbers assigned to each parallel loop. Here the best execution time is recorded which serves as an upper bound on the achievable performance by any scheduling policy.

The results are shown in Figure 5.15 where on average CDMapp is within within 12% of the oracle. This translates into improving program speedup by 1.42x over an online technique, 1.17x over an offline model. This relative improvement is less than seen in the dynamic scenarios described earlier. As expected, the offline approach works well here as the deployment scenario is similar to the training environment. In more dynamic settings, such learning breaks down and CDMapp is more powerful.

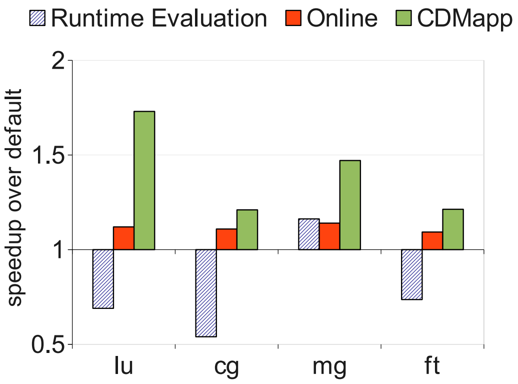


Figure 5.14: Runtime evaluation of different offline-tuned versions drastically degrades program performance.

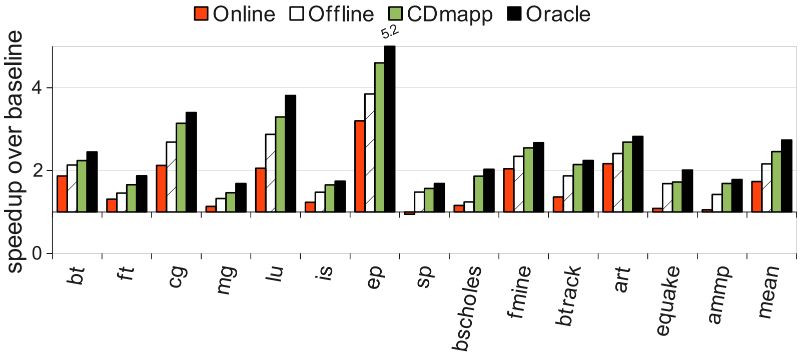
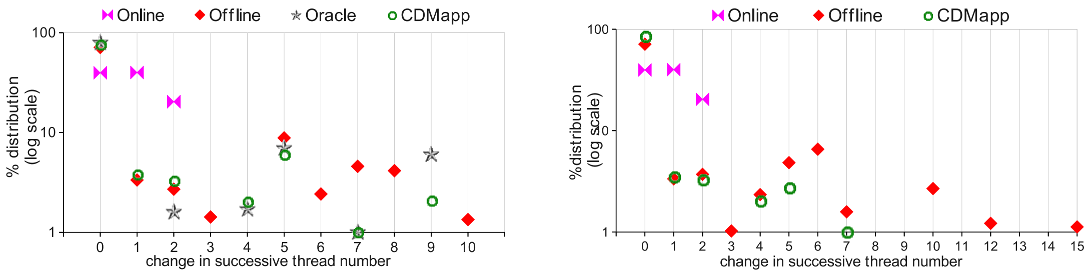


Figure 5.15: Comparison of CDMapp with existing techniques in a controlled static environment. It improves speedup by 11.7% over offline and is within 12% of oracle.

## 5.7 Analysis

This section analyses the source of performance of CDMapp. It first investigates how often each mapping approach changes the thread number in both controlled and dynamic environments. It is followed by an evaluation of the accuracy of the environment and thread number predictor in the controlled environment. Next, an evaluation of the threshold used by CDMapp scheme to modify thread prediction is discussed.



(a) Controlled static environment

(b) Realistic dynamic environment

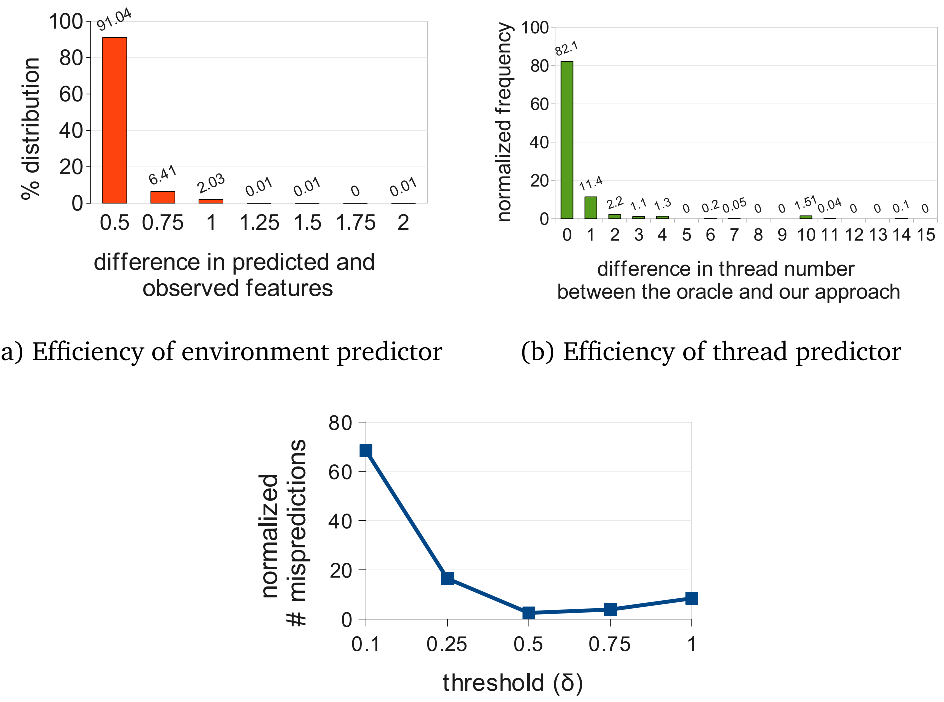
Figure 5.16: (a) In a controlled static environment with known oracle, CDMapp closely follows the oracle scheme. (b) In a dynamic environment with unknown oracle, CDMapp changes to the optimal thread number only as and when required. In both cases, online approach confines to fixed thread number change. Offline model is unable to adapt to new environment mispredicting thread number despite few thread number switches.

### 5.7.1 Thread Number Change

In dynamic environment, it is desirable to switch to the right number of threads as soon as possible, at any parallel loop of the target program to reduce the contention in the system. Such instant switching of thread numbers ensures a quick reaction to the external system.

Figure 5.16 shows the distribution of changes in thread number selected by each of the schemes. In addition, in the controlled static environment, this result can be compared with the known oracle. In both environments, the online approach changes the number of threads significantly but by a small amount. When compared to the oracle in the controlled environment, it clearly changes too frequently, and when it does need to change, it does not change by enough and is, therefore, slow to react to change. Its behaviour remains the same in both environments.

As expected the offline scheme performs well in the controlled environment as it is similar to the training set. It follows the oracle in changing only when needed. As the ideal number of thread changes increases, however, the offline approach diverges from the oracle mispredicting change in the 5 to 10 threads region. This large change in thread number is more apparent in the dynamic environment. CDMapp approach follows the oracle and, initially, like the offline scheme, in the controlled environment it changes infrequently. Unlike the offline scheme, it follows the oracle more closely when larger changes are needed only overestimating a change of 2 and underestimating a



(a) Efficiency of environment predictor

(b) Efficiency of thread predictor

(c) Threshold sensitivity analysis

Figure 5.17: (a) CDMapp achieves near-accurate (91%) environment prediction within the threshold. (b) Difference in thread numbers between the oracle and CDMapp. In 80% cases threads determined is same as the oracle. (c) Sensitivity of the threshold value (δ) to the number of mispredictions δ=0.5 has least mispredictions.

change of 9 threads. In the dynamic environment it is much less aggressive than offline in selecting large changes leading to significant performance improvement.

### 5.7.2 Environment Prediction Accuracy

CDMapp relies on an environment predictor to measure the reliability of the thread predictor. The accuracy in predicting near accurate future environment is crucial as it determines the efficiency of thread number prediction. In Figure 5.17(a) a plot of frequency distribution of the difference between predicted and actual environment in the limited controlled study described in Section 5.6.6. This difference when confined within the threshold of 0.5 is around 91% of the cases. This value shows that CDMapp yields accurate environment prediction based on current information and hence the predicted thread number is near-accurate.

### 5.7.3 Thread Prediction Accuracy

The oracle study observed in the limited study gives deep insight into the accuracy of the thread predictor. Figure 5.17(b) shows a plot of the difference in thread numbers between the oracle and CDMapp approach. The x-axis shows the thread number difference [0-16] and the y-axis shows the frequency distribution of this gap between thread numbers [0-100%]. It can be observed that this approach closely follows the oracle around 80% of the time which highlights the accuracy of the model.

### 5.7.4 Threshold Sensitivity Analysis

At the core of the CDMapp models: thread-predictor and environment-predictor lies the threshold factor (δ). It is thus crucial to determine the ideal value of the threshold that determines any changes in the environment. Graph 5.17(c) provides a sensitivity analysis of the threshold parameter used to decide mispredictions. The x-axis shows the range of threshold from [0-1], and y-axis shows normalized percentage of mispredictions [0-100%]. The threshold value that has least percent of mispredictions is chosen to judge the quality of prediction. It is clear that a threshold of 0.5 is the sweet-spot for determining mispredictions.

## 5.8 Summary

This chapter discussed an approach that to determine the best thread mapping for any parallel program that is adaptive to change in the system. A novel technique based on *online change detection* is proposed to improve predictions of offline trained models that have limited knowledge of the best scheme at runtime. This adaptivity is achieved by predicting future environment that acts as a feedback to improve mapping policy. This approach evaluated with varying loads and hardware resources, speedup improvement of 2.14x over OpenMP default, 1.58x over an online approach and 1.32x over an offline trained model.

The next chapter discusses an approach for mapping in varying execution scenarios with no single optimal policy. This solution uses different expert mapping policies and learns a model online, that selects the best expert at runtime that is optimal for that execution scenario.

# Chapter 6

# A Mixture of Experts Approach for Efficient Runtime Mapping

This chapter presents a novel parallelism mapping solution in dynamic environments based on mixture of experts idea. A one-size fits all approach may not suit all execution scenarios when the environment is diverse and dynamic. Moreover, it is not easy to adjust the policies to suit different evolving scenarios. The proposed solution based on mixture of experts approach selects the best mapping policy or expert from several offline experts. Each expert performs the best mapping in a sub-state space of all possible execution states. The mechanism is designed in such a way that it is flexible to adjust the number of experts on-the-fly.

The chapter begins with an introduction in Section 6.1. Next, Section 6.2 highlights the motivation showing how existing approaches lag in adjusting to dynamic system changes. The mixture of experts concept is described in Section 6.3 and further description of how the offline experts and online expert selector are built is described in Section 6.4. The evaluation methodology is described in Section 6.5. Later, Section 6.6 discusses experimental results and subsequent analysis in Section 6.7. This chapter concludes in Section 6.8.

## 6.1 Introduction

Modern day hardware platforms are parallel and diverse. In the past, parallelism was restricted to HPC environments running a single application at a time on an isolated system with fixed, known resources. This is no longer the case; mainstream applications have to share dynamically varying resources.

Matching program parallelism to platform parallelism is a real challenge for compilers when the environment is shared, dynamic and unknown at compile time. When the environments have dynamically varying external workloads and hardware resources, this problem becomes more complex. Runtime systems try to adapt parallelism to system changes. However, these solutions are program agnostic and slow to react. Such approaches are characterised by one-size fits all assumption. They have a single monolithic model or policy that matches a program to its parallel environment. There is little ability to examine whether the policy fits the current setting or whether another would perform better. No matter how parameterized the policy is, it is highly unlikely that a scheduling policy developed today will always be suited for tomorrow. One critical problem with current approaches is that they cannot be easily updated or extended. Adding additional expertise requires rewriting (or retraining) the policy. Furthermore, improving one of the policy heuristics may adversely affect others.

This chapter presents a new approach based on predictive modelling that considers a number of thread selection policies at runtime and selects the one that it believes will perform best in a particular scenario. As the execution environment changes, different policies will be dynamically selected at runtime. Such an approach is known as a Mixture of Experts (2.5). It avoids highly complex heuristics and over-fitting training data by allowing different experts to be selected based on their worth. This work focuses on one area of parallelism mapping, selecting the best number of threads for a parallel program. It is the key decision when reconciling program parallelism with available resources.

The central issue is to evaluate if a policy is good at runtime. Trying all policies online is expensive. Furthermore, once a policy is selected, there is no monitor to evaluate its performance as the environment keeps on changing. This deficit is overcome by developing models that not only predict what the right number of threads should be for a program, but also what the *environment* will look like based on this decision. The models are constructed such that if the environment was predicted correctly, then so was the number of threads. Given this ability to determine whether a policy is accurate, the efficiency of each expert is monitored dynamically. The expert that is observed to be most accurate is then selected.

The following sections provide sources of motivation to highlight how existing techinques fail in these scenarios. It further describes the proposed solution and the

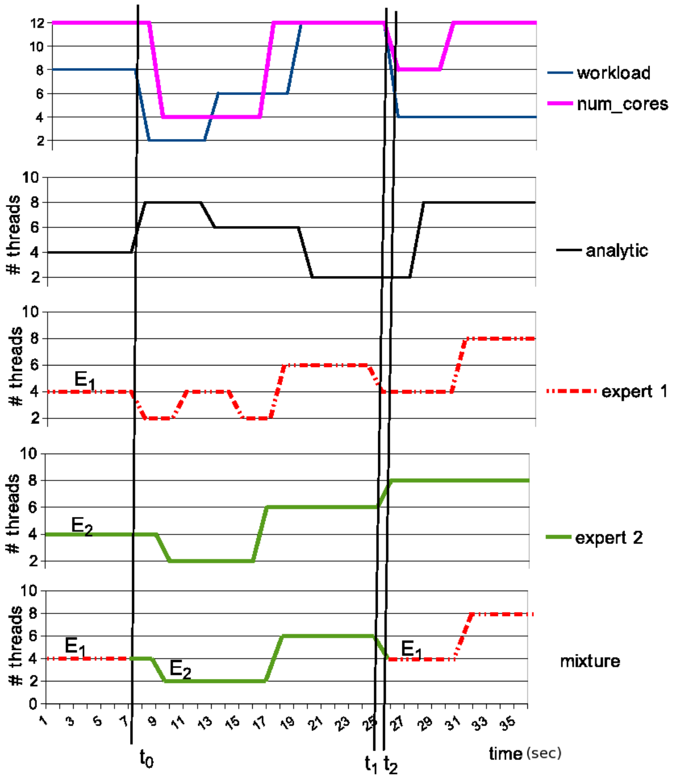


Figure 6.1: A snapshot of the dynamic system. Top graph shows the number of external workload threads and the number of cores available vs time. Remaining graphs shows the number of threads selected for target program `lu` by each policy; analytic, expert 1, expert 2 and mixture, over time. Change points at $t_0$, $t_1$, $t_2$ are highlighted. Analytic is delayed in reacting to change. The mixture approach selects expert 1 until $t_0$, expert 2 until $t_2$ and expert 1 thereafter.

methods of selecting and building the offline experts and online learning of the expert selector.

## 6.2 Motivation

The workload pattern in a highly dynamic realistic system (refer Chapter 1) is replicated on a scaled-down experiment on a 12-core machine as described in Section 4.4. As shown in the top graph of Figure 6.1, the number of workload threads and available processors varies over time. Here target `lu` co-executing with `mg` that uses OpenMP default policy, is optimized by trying to select right number of threads. The remaining graphs in this figure shows the number of threads selected by different policies over

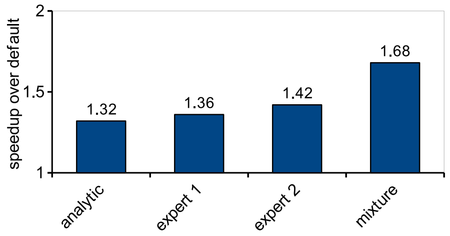


Figure 6.2: Selecting an optimal policy at runtime improves program performance.

time, reacting to changes within the target program and the external environment.

The graph labelled *analytic* depicts the behaviour of a scheme recently suggested in [Sridharan et al. 2014]. It dynamically changes the number of threads selected in response to the changes in the environment. However, to do this, it first runs a parallel section with varying number of threads to measure the speedup before settling on a preferred number resulting in a delay as seen at time step $t_0$. Just when the number of resources drops, this scheme increases the number of threads based on out-of-date data collected before $t_0$. It eventually settles down, but this delay has cost.

The next two graphs show the behaviour of two *experts*. Each expert ($E^1$, $E^2$) uses an offline trained model that predicts the best thread number. They differ in the state space that they are trained for, $E^1$ is more sensitive to changes in the number of processors than $E^2$, and consequently select different thread numbers.

The final graph shows the number of threads selected by the *mixture* approach. It chooses the expert which is best-suited for that current execution environment. For example, consider the timeline figure at two time stamps $t_1$, $t_2$. Initially, expert $E^2$ is the best model at $t_1$. When the execution environment changes at $t_2$, the selector switches to expert $E^1$ as it is more appropriate than $E^2$. As shown in Section 6.4.3, $E^1$ predicts the environment more accurately at $t_2$ than $E^2$, so this was the right decision.

The program performance using these techniques is seen in Figure 6.2. The analytic approach improves over the OpenMP default but is outperformed by either expert due to the delay in reacting to change. Having the ability to switch between experts dynamically greatly improves performance further still.

## 6.3 Mixture of Experts (ME): Overview

The mixture of experts approach selects the best offline policy or expert at runtime to predict the best number of threads. Given a number of mapping policies, this approach learns online which expert is best suited to each dynamic decision. Central to this formulation is the concept of reward i.e. determining how good a mapping is.

### 6.3.1 Offline Experts

Every expert is associated with two predictive models which are described in Section 5.4. These are (1) *thread-predictor* '*w*' that predicts the optimal thread number and (2) *environment-predictor* '*m*' that predicts the future environment after selecting the optimal number of threads, both based on the program characteristics and the current runtime environment.

**Thread Predictor:** A model $x$ is learnt that given a thread number $n$ and the current state at time stamp $t$ encoded as a feature vector, $\underline{f_t}$ returns an approximation of eventual speedup

$$x(n, \underline{f}) = \hat{V}(n)$$

whereˆis used to denote an approximation and $\hat{V}(n)$ is the predicted program speedup with $n$ threads. A *thread-predictor* '*w*' that selects the thread number that is predicted to maximize speedup:

$$w(\underline{f}) = n|(argmax_n(x(n, \underline{f})))]$$

The details of how such a predictor $w$ is learnt are described in Chapter 4.

**Environment Predictor:** At time step $t$, this predictor determines the environment at the next timestamp $t+1$.

$$m(\underline{f_t}) = \underline{\hat{e_{t+1}}}$$

If this prediction is incorrect then, the thread prediction $w$ will be incorrect. While it is highly unlikely to determine the accuracy of $w$; it is easy to judge the accuracy of $m$ at the next timestamp. As $m$ and $w$ are built from the same training data, if $m$ is accurate, there is a high chance that even $w$ is accurate, as observed from the evaluation results.

Environment prediction is the key to monitoring the accuracy of the experts. Existing experts that are generated using machine learning can be retrofitted by retraining them, using the same original training data, to predict the environment as well. It is more challenging for hand-crafted or ad-hoc experts as a new environment predictor would need to be created. Alternatively, an expert can be selected online periodically (with no environment predictor) and see how it affects the environment and record the result, slowly building an environment predictor automatically over time.

### 6.3.2 Expert Selector

Unlike standard ME, exhaustive evaluation of all experts cannot be performed online owing to extensive time consumed. There is also no way to monitor, online, how an expert performed. In this work, the environment predictor is used as a proxy to the thread predictor's quality. It, therefore, allows for swift adaptation by reducing additional overhead. Assuming that there are a number of different predictors or experts, each of which has an associated predictor pair $(m^k, w^k)$, the role of the mixture of experts model $M$ is to select the best expert '$k$' which is predicted to give the best performance as:

$$M(\underline{f_t}) = k|argmax_k(argmax_n(x^k(n, \underline{f_t})))]$$

in other words, select the expert that is expected to predict the number of threads that will lead to the maximum speedup. This selection is to be performed *online*. Learning the best gating function $M$ in this way is not feasible, as it is not possible to evaluate the performance of $x^k$ at runtime. Instead, the environment predictor is used here as:

$$M(\underline{f_t}) = k|argmin_k||\hat{e_t^k} - e_t||$$

in other words, select the expert that is most accurate in predicting the environment. As this can be evaluated at each time step, it can be used to build, online, the mixture of experts model. The code and environment features, $f$, are input to the online expert selector $M$ which determines which expert to select. The predicted thread number of the selected expert is then assigned to the parallel section. This model adapts over time based on the accuracy of each expert's prediction of the future environment. Figure 6.3 shows the mixture of experts idea.

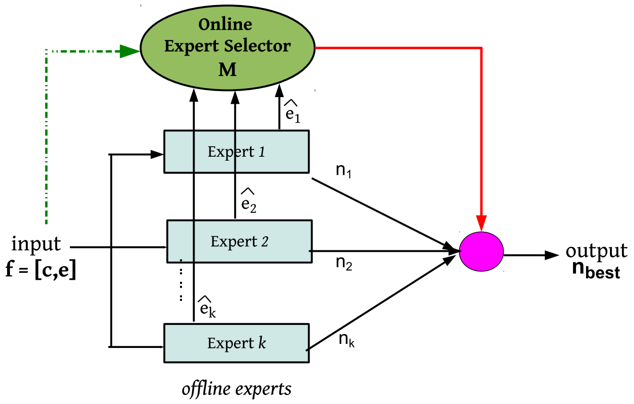


Figure 6.3: Mixture of experts: Depending on the input state $f$ consisting of code $c$ and environment $e$, the online model $M$ chooses an expert most likely to select the best number of threads $n$. Based on the environment prediction accuracy of each expert, $\hat{e}$, it updates its choice over time.

## 6.4 Approach

This section describes how individual experts are learnt. It then discusses how a model that selects the best expert online is built.

### 6.4.1 Individual Experts

Each expert is an offline trained mapping policy with two models as mentioned above. Any (potentially external) expert that determines these two parameters, via whatever means, can be included in the existing mixture.

There are numerous ways of selecting the training data for each expert. This work uses four experts chosen arbitrarily. How the experts are selected is shown in Figure 6.4. An initial analysis of the number of threads vs program performance is discussed in Section 6.7. The training programs are classified into two sets: scale and do not scale, based on their scaling behaviour. Then an expert is trained on two sets of hardware; 12-core and 32-core machines. This classification results in four experts. Determining the best number of experts in advance is a complex process. This open question needs further experiments and analysis. For this work, a program is said to scale up when it achieves a minimum speedup of $P/4$ when it executes with maximum hardware contexts on an isolated machine with $P$ processors. All program combinations are executed to collect the training data that hopefully covers the potential deployment scenarios.

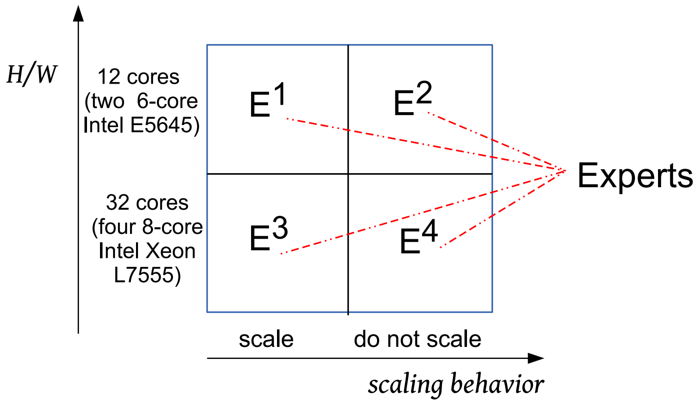

H/W
12 cores (two 6-core Intel E5645)
32 cores (four 8-core Intel Xeon L7555)
E1
E2
E3
E4
Experts
scale
do not scale
scaling behavior

Figure 6.4: Diagram showing how four experts are selected.

**Generating the Experts:** Experts are created using predictive modelling techniques. Training is performed once in an offline setting. No further training is allowed for new programs. Supervised learning process, described in Section 2.3.1, is employed for training the experts. As every machine learning model is built using a rich training set, how such data is generated is described next along with the set of features used.

**Generating the Training data:** Similar to the training process discussed in Chapter 4, the training data is generated by executing programs from NAS parallel benchmark suite an a 12-core system. Each target program co-executes with a single workload with varying number of threads for both programs. Apart from this, training experiments were also performed on another platform (32-core system) which generated new training data. Features are captured as $\underline{f} = [\underline{c}, \underline{e}]$ where $\underline{c}$ are static code features and $\underline{e}$ environment features. The number of threads '$n$' that led to best performance and the norm of corresponding environment vector are recorded.

**Features:** In addition to the eight features (see Section 4.3.1), two additional features are found to be essential for this training set. These features are listed in Table 6.1. At loop $i$, the feature vector $\underline{f_i} = (f^1{}_i, .., f^{10}{}_i)$ is formed by these ten features.

Although all experts use the same features, they vary in importance across each expert. Figure 6.5 shows the importance of selected features across different experts. The resultant normalized values for all four experts form the pie-charts. Each slice of the pie-chart corresponds to how crucial is the feature for that expert. These features are ranked in the order of relative importance to the experts. The $\pi$ values below every pie-chart represent the feature impact values (see Section 2.3.4). For example, run

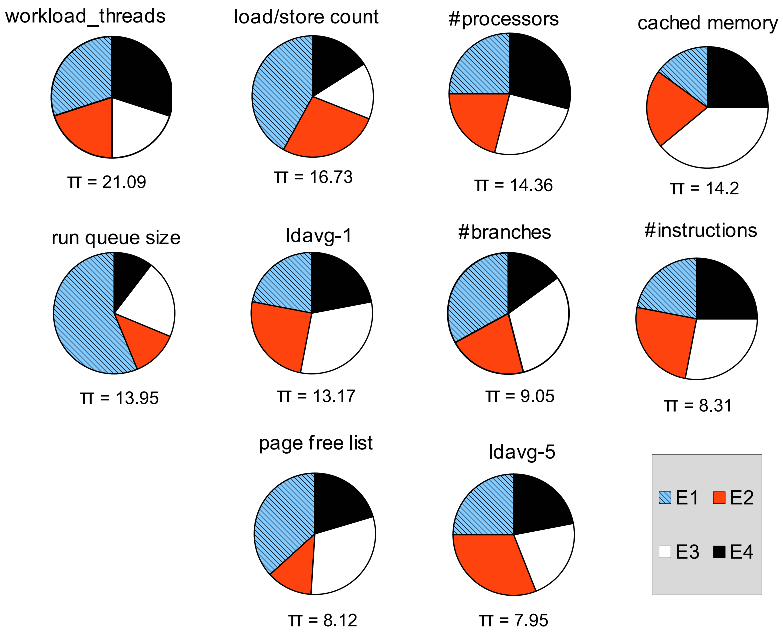


Figure 6.5: Impact of selected features on the experts. A slice in each pie-chart corresponds to how crucial is the feature for that expert. Feature impact ($\pi$) value below a pie-chart is averaged across all experts.

queue size is more critical to expert $E^1$ and less important to other experts. Certain features such as *#processors* are nearly the same for all experts.

**Building the models:** Similar to the process in Section 5.4, linear regression technique employing standard least squares is used to build two models that fit the training data. Alternate machine learning techniques could equally be used. Learning a model for this data is simply finding the best linear fit to the data i.e. determining weights for each selected feature ($w_1 f_1 + .. + w_n f_n + \beta$). This results in simple 10-dimensional linear models $n = \underline{w} \cdot \underline{f}$ and $\hat{e} = \underline{m} \cdot \underline{f}$ where the weights (coefficients) $w$ and $m$ are listed in Table 6.1. Each of the four experts has respective weights.

### 6.4.2 Expert Selector

A mixture of experts model $M$ consists of a series of hyperplanes $\underline{S}$ in the 10-dimensional feature space $\underline{f}$. These hyperplanes define the regions in the feature space where one expert is more accurate than the others. The environment prediction error, $a$ for expert $k$ is defined as

$$a^k = \|\underline{e}^{\hat{k}}_t\| - \|\underline{e}_t\|$$

| Features | | | $E_1$ | | $E_2$ | | $E_3$ | | $E_4$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Description** | **type** | $w$ | $m$ | $w$ | $m$ | $w$ | $m$ | $w$ | $m$ |
| $f_1$ | load/store count | compiler | 1.05 | -0.47 | -0.84 | 1.02 | 0.14 | 1.1 | 0.05 | 0.74 |
| $f_2$ | instructions | “ | -1.52 | 0.35 | 1.12 | -0.78 | 0.95 | 1.10 | 0.03 | 1.03 |
| $f_3$ | branches | “ | 0.87 | 1.15 | 0.84 | 0.05 | -0.87 | 0.54 | -0.57 | 1.12 |
| $f_4$ | workload threads | Linux | -0.62 | 0.39 | 0.05 | 0.44 | -0.48 | 0.44 | 0.004 | 0.39 |
| $f_5$ | processors | “ | 0.98 | 0.46 | 0.98 | 0.002 | 0.99 | 0.142 | 0.92 | 0.74 |
| $f_6$ | run queue length | “ | 0.003 | 0.29 | 0.02 | 0.23 | -0.15 | 0.25 | 0.22 | 0.28 |
| $f_7$ | cpu load-1 | “ | 0.002 | 0.17 | 0.03 | 0.09 | 0.473 | 0.07 | 0.01 | 0.09 |
| $f_8$ | cpu load-2 | “ | -0.013 | 0.64 | 0.227 | 0.6 | -1.07 | 0.15 | -0.62 | 0.59 |
| $f_9$ | cached memory | “ | -0.07 | 0.01 | 0.002 | 0.05 | 0.007 | 0.06 | 0.03 | 0.12 |
| $f_{10}$ | pages free list | “ | 0.004 | 0.002 | -0.08 | -0.04 | 0.01 | 0.14 | -0.14 | 0.003 |
| β | constant | | -1.21 | 0.25 | -6.8 | 0.28 | -3.03 | 0.33 | -2.5 | -0.05 |

Table 6.1: List of features, regression coefficients.

The hyperplanes $\underline{S}$ are learnt online such that the error of a predictor $k$ in this region is less than the average error of other predictors.

$$argmin_{\underline{S}^k}(a^k - \frac{\sum_{i\neq k} a^i}{K-1})$$

where,

$$S^{k-1} < \underline{f} \leq S^k$$

The feature space $\underline{f}$ is evenly partitioned initially and then adjusted online based on the accuracy of each expert. To update the model, the data from the last time-step is used to minimize runtime overhead.

### 6.4.3 Example

To demonstrate how our approach works consider the workload timeline shown in Figure 6.1. At timestamp $t_1$, the feature-vector $\underline{f}_1$ is

$$\underline{f}_1 = [0.032, 0.026, 0.2, 4, 8, 16, 4.76, 2.17, 1.11, 1.65]$$

Expert $E^1$ predicts thread number $n_1^1$ and environment $\hat{e_1^1}$ by the product of weights $w_1, m_1$ from Table 6.1 with $\underline{f}_1$ as

$$n_1^1 = \underline{w^1} \cdot \underline{f}_1 = 4; \quad ||\hat{e_1^1}|| = \underline{m^1} \cdot \underline{f}_1 = 12.56$$

Similarly expert $E^2$ predicts

$$n_1^2 = \underline{w^2} \cdot \underline{f}_1 = 6; \quad ||\hat{e_1^2}|| = \underline{m^2} \cdot \underline{f}_1 = 8.54$$

The Mixture of Expert selection $\underline{S}^1$ hyperplane is = [0.04,0,02, 0.2, 6, 10, 14, 4.00, 2.00, 1,1.5] and as $\underline{f}_1 < \underline{S}^1$, it selects $E^2$ as its expert and chooses 6 threads. This, in fact, turns out to be the correct decision as the actual measured environment is $||\underline{e}_1|| = 8.713$ which is closer to $E^2$'s prediction of 8.54 rather than $E^1$'s of 12.56. Later at timestamp $t_2$, the feature-vector $\underline{f}_2$ is

$$\underline{f}_2 = [0.045, 0.013, 0.1, 12, 12, 6, 2.73, 2.17, 0.01, 1.21]$$

Here, expert $E^1$ predicts $n_2^1$ and $\hat{e_2^1}$ as

$$n_2^1 = \underline{w}^1 \cdot \underline{f}_2 = 4; \quad ||\hat{e_2^1}|| = \underline{m}^1 \cdot \underline{f}_2 = 13.94$$

Similarly expert $E^2$ predicts

$$n_2^2 = \underline{w}^2 \cdot \underline{f}_2 = 8; \quad ||\hat{e_2^2}|| = \underline{m}^2 \cdot \underline{f}_2 = 8.504$$

Here, the mixture of experts selects expert $E^1$ as $\underline{S}^1 < \underline{f}_2$ and chooses 4 threads. This is the correct decision as the actual measured environment is $||\underline{e}_2|| = 11.763$ which is closer to $E^1$'s prediction of 13.94 rather than $E^2$'s of 8.504. If there was a misprediction, the hyperplane $S$ would be updated to reclassify this feature point.

## 6.5 Experimental Set-up

This section describes the hardware platform and benchmarks used and outlining the dynamic environment. It then lists the compared adaptive techniques.

### 6.5.1 Hardware and Software Configurations

All experiments are evaluated on the platform listed in Table 6.2. The target and workload programs start executing at the same time and continue till the other finishes. Each experiment was repeated three times and the mean value of program execution time is reported for all evaluation results.

### 6.5.2 Benchmarks

Several multi-threaded programs from various benchmark suites are chosen for evaluating this approach. These include all OpenMP-based C programs from NAS, SpecOMP and Parsec benchmarks as listed in Appendix A.

| | |
|---|---|
| **Hardware** | 32-core Intel Xeon L7555 @1.87GHz |
| | 4 one-socket nodes, 8 cores/socket |
| **OS** | 64-bit openSUSE 12.3 version |
| | 3.7.10 kernel |
| **Compiler** | *gcc* 4.6 -O3 optimization |

Table 6.2: H/W and S/W configurations of the evaluation system.

### 6.5.3 Policies

The mixtures approach is evaluated against the following adaptive policies. A detailed description of these policies is discussed in Chapter 4. A brief mention of these policies is as follows:

**Default:** OpenMP default policy assigns a thread number equal to the maximum number of available processors.

**Online:** This is a robust adaptive scheme that employs hill-climbing technique which changes the thread count at runtime based on execution time, responding to change in system environment.

**Offline:** The offline techinque uses a machine learning heuristic predicts a thread number at runtime based on an offline-trained model to map programs in the presence of external workloads.

**Analytic:** In [Sridharan et al. 2014] an analytical model determines the degree of parallelism at runtime based on observed speedups at fixed time-intervals and estimated using regression techniques.

### 6.5.4 Experimental Scenarios

In a highly dynamic environment, the target co-executes with varying workloads and changing number of processors. This is to reflect drastic changes in the system which changes the resource availability and contention. How these parameters are varied is described below. The effect of other external system issues such as network contention are reflected in the set of runtime features used in this model.

**Workloads:** The external workload consists of multiple parallel programs selected from the above benchmark programs. The number of workload programs and the

| Workload type | Programs |
|---|---|
| small | (i) `is, cg` |
| | (ii) `ammp, ft` |
| large | (i) `sp, bt, equake, is, cg, art` |
| | (ii) `bscholes, lu, bt, sp, fmine, art, mg` |

Table 6.3: Workload configuration.

number of threads are varied in each setting. Based on this configuration the workloads are classified as *small* and *light*. For each workload type, consider different sets of programs are considered as shown in Table 6.3. The workload is dynamically changed at every 2 seconds. All results are averaged over these different benchmark sets. The same external workload is reproduced for all evaluated policies in all cases. This ensures a fair comparison across different mapping policies.

**Hardware:** To reflect any change in hardware, the number of available processors are varied during program execution. This change can be due to several factors including hardware failures, turning them off for saving power, assigning more/fewer cores for other high/low priority jobs. It is assumed that hardware changes less frequently (at least 5x slower) than workloads. Hence the number of available processors is varied in two different frequencies: *low* and *high*: every 20 seconds and 10 seconds in low frequency and high frequency settings respectively.

## 6.6 Experimental Results

Firstly, all policies are executed on an isolated and static system to evaluate the overhead incurred. Next these policies are run in a dynamic environment as described in the previous section. The performance results for these settings are presented later. In all experiments the OpenMP default policy is used as a baseline for all experiments.

### 6.6.1 Isolated and Static Environment

Figure 6.6 shows the results of the evaluated schemes in a system that is static (no changes in the environment) and isolated (no co-executing workloads). The evaluated techniques return the corresponding best thread number when invoked.

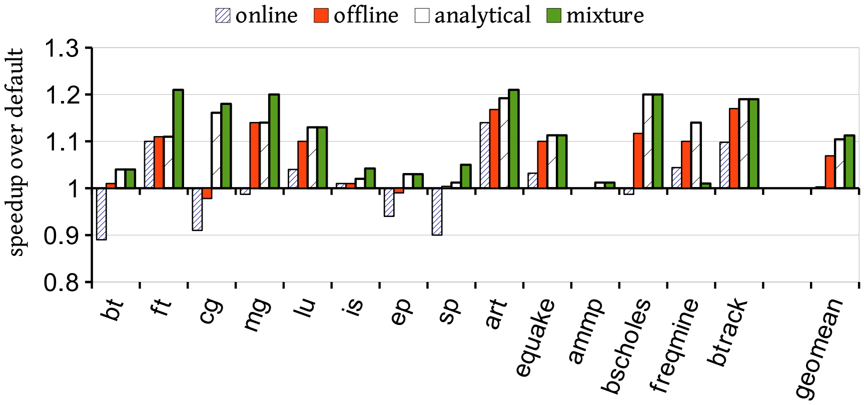


Figure 6.6: Evaluation of policies in an isolated static system. Mixtures approach adds no overhead.

Online spends too much time trying different thread numbers is unable to detect lack of workloads, yet employs the same optimization to find the best degree of parallelism. It, therefore, slows down a few programs. However, offline and analytic approaches adjust to find thread number leading to good speedup. The mixtures approach never slows down the target. Moreover, it improves certain programs like `mg`, `cg`, `art`. These involve irregular memory accesses and barriers and allocating many threads slows down the program. This approach analyzes this behaviour and determines the optimal thread number of the best expert which is most suited for a given program. On average, mixtures approach improves 1.12x over the default and by 8% over the analytic scheme. This highlights that, although the mixture approach is aimed for dynamic environment, it is promising to see no overhead in a static environment.

### 6.6.2 Dynamic Environment

Intially the results are summarized across all experimental scenarios for all the benchmark programs. This section is followed by a detailed study of the performance results on a per-scenario basis.

### 6.6.3 Overall Results

Figure 6.7 summarises the results averaged across all benchmark programs for different workload and hardware settings. On average, online, offline and analytic approaches improve performance by 1.22x, 1.34x and 1.45x respectively. The mixture of experts approach outperforms all these by achieving 1.69x mean (1.54x median)

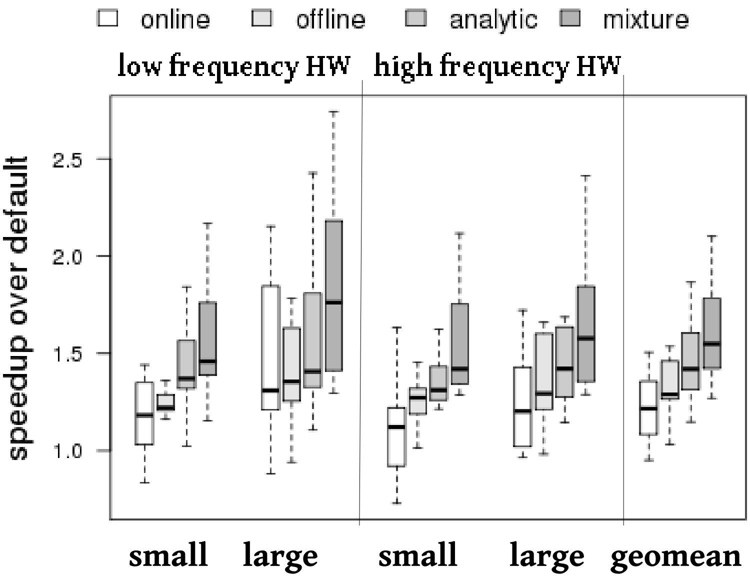


Figure 6.7: Speedup comparison of each scheme per workload and frequency of hardware change averaged across all benchmarks. Overall, on average, online, offline and analytic approaches improve performance by 1.22x, 1.34x and 1.45x respectively. This approach outperforms these by achieving 1.69x mean (1.54x median) improvement.

speedup improvement. In this figure, each box and whiskers represent the distribution of speedup relative to the default. The central line represents the median speedup; the box represents the middle 50% while the whiskers represent the outliers.

The default policy performs poorly due to increased resource contention. The online technique adapts by changing the thread number in response to the observed execution time. However, this reacts slowly to the changes and hence achieves marginal improvement. The offline technique improves over the online scheme, but it is limited by its workload training and cannot adapt to new environments. The analytic technique performs well with workload change but is unable to adjust to the changing hardware resources. The mixture of experts approach immediately detects these changes and selects the best expert that is more specialized in the observed system state. It achieves significant improvement over these existing techniques.

### 6.6.4 Detailed Comparison

Here the experimental results are detailed on a per-benchmark basis, averaged over all the workloads on each experimental setting.

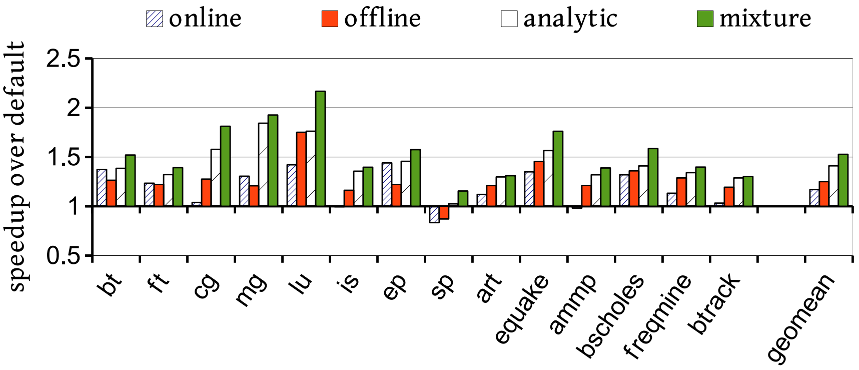


Figure 6.8: For targets executing with small workloads and low frequency hardware changes, the mixtures approach improves 1.52x over default, 1.3x over online, 1.22x over offline and 1.17x over analytic techniques.

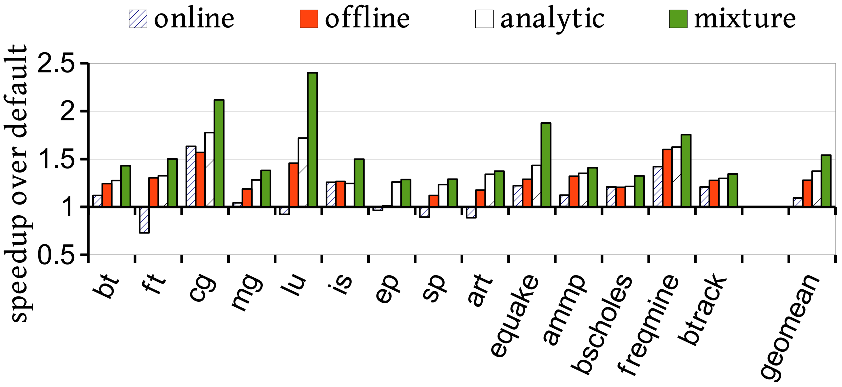


Figure 6.9: For targets executing with small workloads and high frequency hardware changes, the mixture of experts approach achieves performance of 1.55x over default, 1.42x over online, 1.21x over offline and 1.14x over analytic techniques.

**Small workload:** With small workloads, resource contention is minimal, however the changing number of processors limits the amount of computing resources.

**Low frequency hardware change:** Figure 6.8 shows the speedup for each policy averaged across all workload programs when the change in hardware is low. Here the mixture approach improves performance 1.52x over OpenMP default and outperforms all other schemes by 1.3x over online, 1.22x over offline and 1.17x over analytic techniques. Online improves over offline for `bt, ep` but it performs worse than the default for `sp`. Analytic approaches the performance of the mixture policy for `mg, cg, art, btrack` but in each case it is outperformed by the mixture of experts.

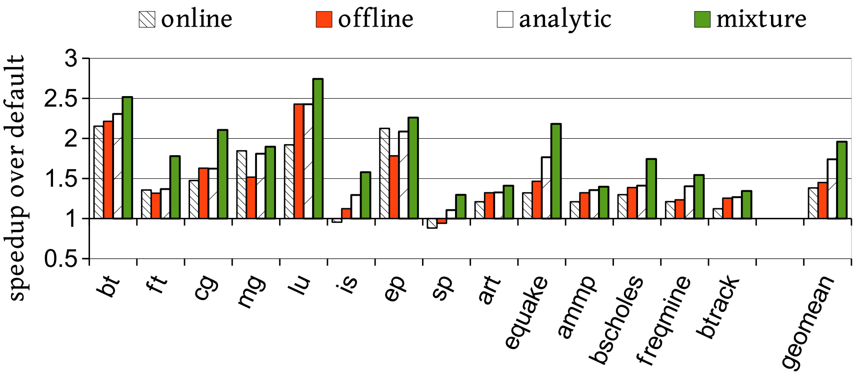


Figure 6.10: For targets executing with large workloads and low frequency hardware changes, the mixtures approach significantly improves 1.79x over default, 1.29x over online, 1.23x over offline and 1.19x over analytic techniques.

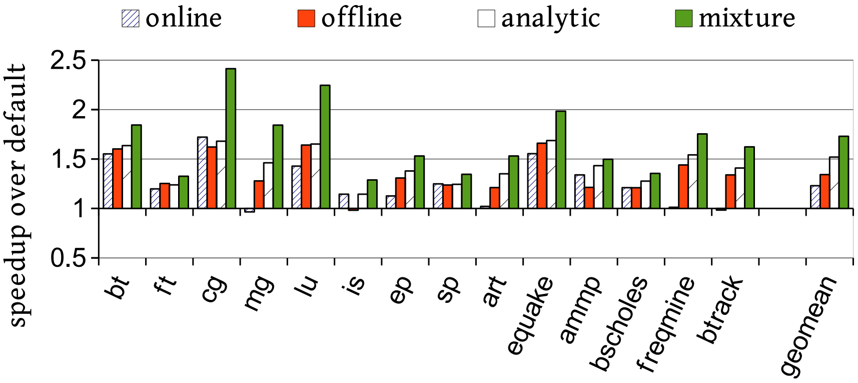


Figure 6.11: For targets executing with large workloads and low frequency hardware changes, the mixture of experts approach improves 1.65x over default, 1.34x over online, 1.26x over offline and 1.20x over analytic techniques.

**High frequency hardware change:** The results for this setting are shown in Figure 6.9 where the performance is improved by 1.55x over the OpenMP default using mixtures approach. It outperforms the other techniques, 1.42x over online, 1.21x over offline and 1.14x over analytic techniques. Online slows down certain programs, e.g., `ft, sp, art`. The offline approach never slows down any target program performing better than the online scheme.

**Large workload:** With large workloads, the contention for system resource is greater. In addition, variation in the available processors compounds this effect.

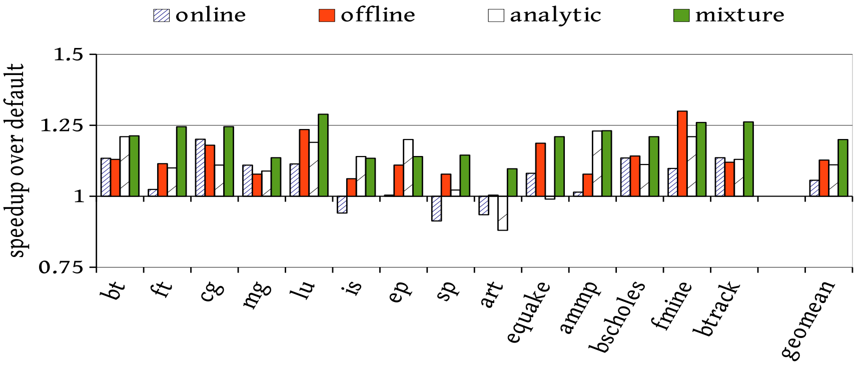


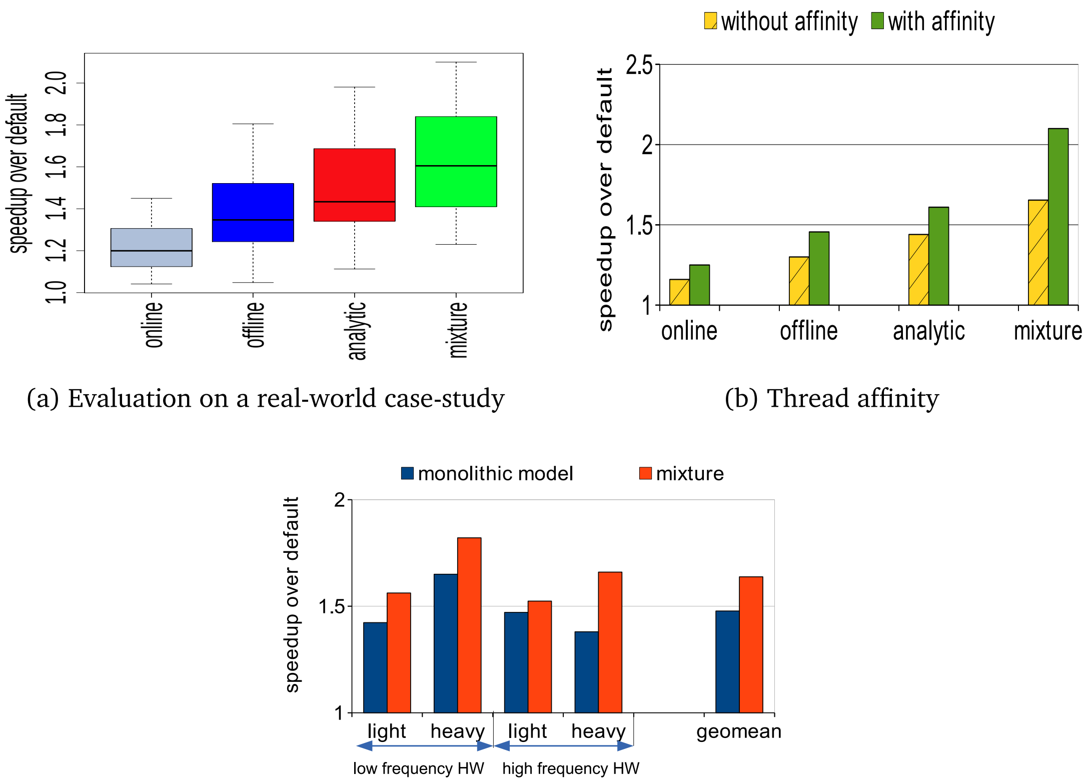
Figure 6.12: Effect of policies on external workloads. The mixture approach never degrades workloads, improving workload by 1.19x on average.

**Low frequency hardware change:** Figure 6.10 shows the results for this scenario. On average the mixtures approach achieves performance improvement of 1.79x over the default, 1.29x over online, 1.23x over offline and 1.19x over analytic techniques. Online improves bt, ep but slows down is, ep. Offline policy improves bt, lu, cg, ep but is ineffective for is, sp, freqmine, btrack. Analytic improves across all programs but is outperformed by mixture of experts technique in all cases. Certain programs like bt, lu, cg, equake benefit a lot from this approach.

**High frequency hardware change:** Here the mixture approach improves 1.65x over default, 1.34x over online, 1.23x over offline and 1.20x over analytic. Programs such as cg, lu, equake, freqmine benefit significantly from the mixture of experts. Offline and analytic improve over online across all programs except sp. Figure 6.11 shows the speedup results in detail.

### 6.6.5 Impact on Workloads

Any optimization scheme improving the target program performance should ideally exert minimal impact on the co-executing workloads. Figure 6.12 shows the impact of the evaluated schemes on the external workloads averaged across all experiment settings. All policies improve over the default on average, though online degrades the workload performance in certain cases. The offline and analytic models marginally improve over the online schemes. The mixture of experts approach outperforms these techniques by improving workloads performance by 1.19x. This result is primarily due to a reduction in system contention benefitting both target and workload.



(a) Evaluation on a real-world case-study

(b) Thread affinity

(c) Monolithic model vs. mixture of experts

Figure 6.13: (a) In a live system, the mixtures approach improves by, on average by 1.32x, 1.21x, 1.19x over the online, offline and analytic models. (b) Impact of affinity scheduling on thread selection policies averaged over all benchmarks and workloads for small workloads scenario. Here, the mixtures approach gives a 2.1x average speedup. (c) Evaluation of monolithic model vs mixture of experts. The latter improves 1.28x over one single model.

### 6.6.6 Case Study

As discussed in previous chapters on evaluating proposed solutions in a real live system (see Figure 1.1), even this technique is tested on the same setting. During this period, there was a hardware failure such that half of the processors were unavailable for 2 hours. This pattern was simulated on the experimental platform, mentioned in Section 6.5. Here the number of workload threads was scaled down in proportion with the maximum number of processors. All target benchmarks are evaluated in this scenario and summarized speedup results are shown in Figure 6.13(a). The average speedups are online: 1.19x, offline: 1.34x, analytic: 1.43x and mixture: 1.61x. Mixture of experts is clearly the superior policy, achieving improvement 1.32x, 1.21x, 1.19x over online, offline and analytic. This illustrates that the mixtures approach works well in totally unseen environments.

### 6.6.7 Thread Affinity

Associating threads to cores via affinity scheduling can improve performance as it may reduce memory traffic. An evaluation to verify this is discussed here. Here affinity scheduling is combined with each of the thread selection policies. All benchmarks are ran with multiple workloads in the small workload scenario described in Section 6.6.4, the scenario likely to benefit most from thread scheduling. The results averaged across the target benchmarks are shown in Figure 6.13(b). All schemes show improvement with affinity scheduling, but the mixture approach gives the largest improvement of 26%, giving an overall speedup improvement of 2.1x.

### 6.6.8 Generic vs. Experts

Here the performance of the mixture of experts policy is compared against a single aggregate model with the same total training data. Such a model is an ensemble of different experts generated by combining the corresponding training data of experts. Figure 6.13(c) shows the speedup comparison using a single model against the mixtures. The mixture of experts gives a 28% improvement over an aggregate model. This is basically due to the failure of the one size fits all approach of the aggregate model.

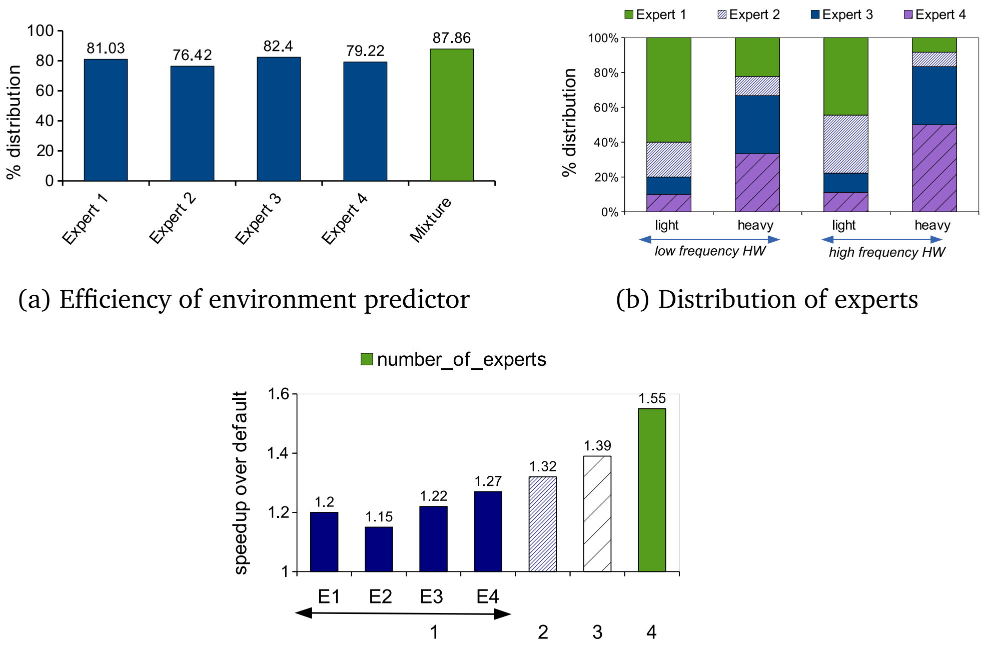
The next section presents a detailed analysis of this approach and results obtained. It highlights the accuracies of environment and thread predictors, followed by the effect of number of experts on performance.

## 6.7 Analysis

Here the accuracy of the experts are analyzed. An initial analysis on how the number of experts impacts performance is presented.

### 6.7.1 Environment Predictor Accuracy

The efficiency of the mixture of experts approach relies on the environment prediction capabilities of individual experts. Figure 6.14(a) shows how accurate these values are



(a) Efficiency of environment predictor

(b) Distribution of experts

(c) Effect of increasing number of experts

Figure 6.14: (a) Environment predictor accuracy of experts normalized to actual environment. (b) Distribution of the number of times an expert is chosen across each scenario. (c) Effect of increasing number of experts on program speedup averaged across all target programs. A mixture of four experts outperforms the best single expert by 1.22x.

for each expert. Y-axis shows the normalized difference between observed and predicted environment averaged across all experiments. It can be seen that all experts accurately predict the future environment between 79% and 82% of the time. So individually there are highly accurate. When combined in a mixture model, this accuracy increases to 87%.

### 6.7.2 Frequency of Expert Selection

If one expert were to dominate one or more scenarios, then having a mixture may be of little benefit. If, however, the frequency that an expert is selected is independent of scenario; then this undermines the need for online selection. Figure 6.14(b) shows the normalized frequency distribution of how many times each expert is selected in each of the four scenarios. As expected, one particular expert dominates each scenario: expert $E^1$ is used 60% of the time for lightweight, low-frequency scenario while expert

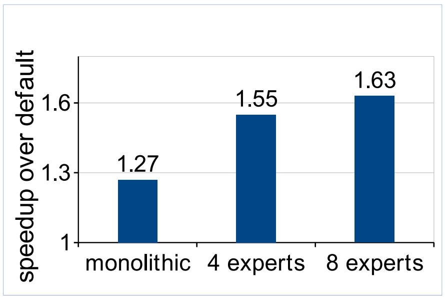


Figure 6.15: Increased granularity of number of experts

$E^4$ is preferred in the heavy-weight, high-frequency setting. Surprisingly, all experts are selected as the best at some point in each scenario. For instance, experts $E^1$,$E^2$ are almost evenly chosen in the lightweight high-frequency setting. This means that experts can be effectively used in scenarios they have not been specifically trained for.

### 6.7.3 Number of Experts

One of the central claims of the mixture approach is that experts can be added over time, helping improve performance. Here, the target speedup is measured with an increasing number of experts in the heavy-workload, low-frequency scenario. Figure 6.14(c) shows the average performance achieved across all benchmarks in this scenario using a varying number of experts. Individually, each expert gives low performance. As expected, from Figure 6.14(b) experts $E^3$,$E^4$ are most accurate here and give speedups of 1.22x and 1.27x. The mismatched experts $E^1$,$E^2$ give performance of only 1.2x and 1.15x. However, adding experts steadily improves performance. This shows that the slight additional cost to determine the environment prediction accuracy is more than compensated by the performance gains. The mixture approach gives a 22% improvement over a best single expert.

### 6.7.4 Experts of Finer Granularity

Here the analysis of the granularity of experts is discussed. Monolithic model is built as a single entity built using the entire training data. Section 6.4 described how four experts were classified and built. Moving towards a finer granularity, eight experts were built by further splitting the training programs. These are grouped into four groups based on speedup of at least $P$, $P/2$, $P/4$, $P/8$ on two systems (16-core, 32-core). The results averaged across all programs for the scenario in Section 6.6.4, are presented in Figure 6.15. It can be observed that an increased number of experts further benefits the programs with eight experts improving by 1.63x and 4 experts by 1.55x. This is probably due to more specialized experts, capturing environment changes more precisely.

## 6.8 Summary

This chapter has presented a technique based on a mixture of experts approach for efficient thread number selection. It determines at runtime, the best offline expert out of a collection of experts. It also provides a mechanism to add additional expertise knowledge gracefully. The main motivation is that there is no one-size fits all universal best mapping policy that is optimal in all possible execution scenarios. On evaluating with varying workloads and hardware resources on a 32-core platform, this approach improves over 1.69x over OpenMP default and outperforms other existing mapping policies. It has no impact on external workloads, rather, it improves them as well.

The next chapter discusses what happens to the system state if the workloads are *smart* and start to adapt using different optimization schemes. It also explores combinatorial executions of target and workload programs using competitive co-scheduling schemes.

# Chapter 7

# Competitive Co-scheduling: A Critique

This chapter presents a thorough analysis of existing state-of-the-art runtime scheduling policies of parallel programs. It presents a detailed evaluation of competitive co-scheduling of programs where programs can employ different policies. The chapter starts with an introduction in Section 7.1. Section 7.2 highlights the need for choosing an optimal scheduling policy and how this policy varies based on the nature of the program. A detailed description and formulation of the different scheduling policies is presented in Section 7.3. A methodology for analyzing program behaviour and classification is described in Section 7.4. Section 7.5 discusses the evaluation methodology followed by discussing analysis by per-policy and per-program basis in Section 7.6. This chapter concludes in Section 7.7.

## 7.1 Introduction

Co-scheduling applications is typical in modern computing scenario to improve system utilization. Minimizing cross-program interference and effective exploitation of resources are crucial in achieving the best possible performance. The problem of smart co-scheduling is near critical in modern day data centers. This problem is further complicated owing to the variety in the nature of the programs being scheduled on modern heterogeneous systems. Moreover, co-scheduling jobs is highly complex owing to different requirements, in terms of jobs preferences and complexity of execution architectures. Compute-intensive workloads exploit processing power to the extreme and scale well. Memory-intensive programs spend significant time in accessing the memory for data retrieval or transfer due to high cache misses. Limited bandwidth constitutes a bottleneck for I/O bound tasks that hinders applications' performance. Such programs do not scale well and they do not effectively exploit available computing resources.

The same nature of the variety in applications is observed in co-executing workloads. For example, `ep` from NAS is embarrassingly parallel that requires no data transfer tasks. On the other hand, `mg` performs long- and short-distance communication and is memory-intensive. When jobs of such different intrinsic behaviours try to co-execute with other programs, the nature and intensity of the contention due to interference varies significantly. Over-subscription or under-subscription of resources can lead to significant performance degradation. Runtime schedulers manage the co-location of these workloads to minimize interference and improve program performance and system utilization.

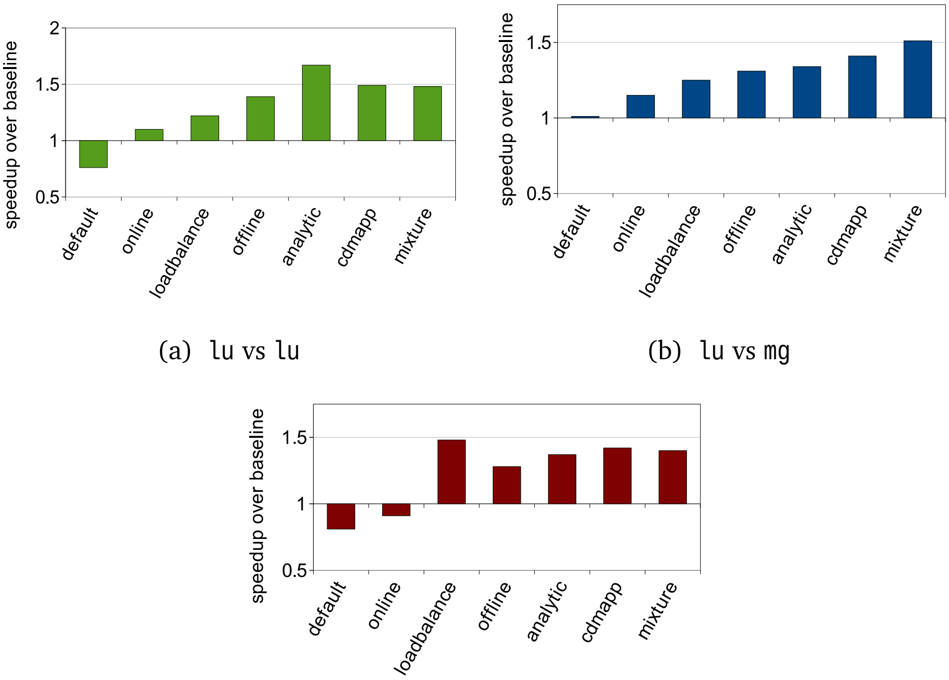
An optimal scheduling policy is the most efficient way to coordinate program execution on large scale settings like grids and clouds. Moreover, it is crucial to know if the system remains stable when each co-scheduled job uses a different scheduling policy. Hence, a proper scheduling mechanism needs to contemplate the program behaviour along with system runtime information to efficient co-scheduling of applications.

## 7.2 Overview

This section provides an example to illustrate that choosing an optimal co-scheduling policy is non-trivial, and a program's performance is dependent not only on the scheduling policies but also that the nature of the program. In addition, the impact on the total combined execution time when the jobs are co-scheduled using different scheduling policies is also discussed.

For this purpose, in an experimental set-up described in Section 7.5, a pair of programs from NAS parallel benchmark is co-scheduled. Both the programs start executing at the same time and continue running until the other program finishes executing. Different co-scheduling policies, described in previous chapters are evaluated when both programs use the same policy.

Initially, a pair `lu, lu` from NAS parallel benchmark is co-scheduled. The combined execution time of the co-scheduled programs compared over a static scheme used as baseline is shown in Figure 7.1(a). It can be observed that the default policy slows down the program due to oversubscription of threads. Rest of the techniques achieve different speedup improvements. Here, the analytic policy turns out to be



(a) `lu` vs `lu`

(b) `lu` vs `mg`

(c) `mg` vs `mg`

Figure 7.1: Comparison of policies of co-scheduling of (a) compute-intensive and compute-intensive (b) compute-intensive and memory-intensive, and (c) memory-intensive and memory-intensive workloads.

the best for both programs by improving over 1.52x. In the second experiment, a compute-intensive program `lu` is co-scheduled with a memory-intensive program `mg`. The performance of different policies in Figure 7.1(b). Here the default policy performs nearly as the baseline. The order of the speedups obtained by these policies is varied now. In this case, out of existing techniques, the mixture of experts approach outperforms the rest by improving 1.5x. Finally, when a memory-intensive program `mg` is co-executed with itself, the default scheme slows down the program. As observed from Figure 7.1(c), the order of best performing policies is yet again different from the previous results. In this case, loadbalance policy achieves best improvement over other policies by 1.48x.

From these performance results, it is quite evident that when a pair of programs is co-scheduled, different combination of mapping policies yields varied results and the impact of each policy changes based on the nature of the program. Moreover, choosing an optimal policy for each program is rather non-obvious. Hence, these factors need to be considered when deciding on the best co-scheduling policy to be employed.

## 7.3 Policies

This section describes the state-of-the-art runtime co-scheduling policies of parallel programs, as described in previous chapters.

There are numerous ways where programs can co-scheduled by different policies. These scheduling policies can be broadly classified as ***fixed*** and ***variable*** policies. A fixed policy determines an optimal resource allocation to a program at the beginning of its execution and never changes it during the program execution. A variable policy changes the allocation during program execution based on the runtime environment.

Fixed scheduling policies can further be classified into *Default* and *Static* schemes and variable policies into *Online, Offline, CDMapp, Loadbalance, Analytic* and *Mixture* schemes. These are described in detail in previous chapters. A quick run-through of these policies is as follows.

### 7.3.1 Fixed Policies

***OpenMP Default:*** (Chapter 4) OpenMP default policy assigns a thread number equal to the maximum number of available processors. It does so at every parallel loop irrespective of the co-executing workload. There is no predefined logic in OpenMP compiler to determine if the default number of threads is optimal for a given execution.
***Static:*** A static policy on a two socket machine with two programs restricts a program to execute on a single socket where each such program is assigned number of threads equal to exactly half of the number of processors. Such a scheme minimizes the cross-cache interference between a pair of co-executing programs. This allocation is static and is undifferentiated amongst different programs.

### 7.3.2 Variable Policies

***Online:*** (Chapter 4) Online scheme determines optimal scheduling based on their adaptation to the environment. The policy employs *hill climbing* optimization to changes thread count in unit steps at runtime based on execution time responding to change in system environment.
***Offline:*** (Chapter 4) This policy uses offline trained predictive model to determine the best thread number for programs. It explicitly considers the state of the program code and the environment.

***CDMapp:*** (Chapter 5) CDMapp uses the offline policy initially and adjusts its mapping decisions by monitoring its performance online. It uses an environment predictor as a proxy to adapt thread number prediction.

***Loadbalance:*** Callisto [Harris et al. 2014] tries to achieve a balance between the co-executing programs by sharing the computing resources by dynamic spatial scheduling to achieve a load balance in the system. When two programs are co-scheduled one on each socket, this dynamic spatial scheduling scheme determines if one of the programs does not benefit from the allocated resources, it reassigns the resources by allocating more cores to the other co-scheduled program.

***Analytic:*** (Chapter 6) This approach uses an analytical model that determines the degree of parallelism at runtime. This is determined based on observed speedups at fixed time-intervals and estimated using simple linear regression technique.

***Mixture:*** (Chapter 6) The mixture of experts approach selects a mapping policy from several offline experts. It learns the best expert based on current executing scenario online. It thus adapts better than previous techniques.

## 7.4 Program Analysis

During mapping of parallel programs, the scheduling mechanism needs to consider program-centric behaviour for efficient policies. The scheduling policy which is optimal for a certain type of program may be non-optimal for a program of contrasting behaviour. Hence capturing this program-centric behaviour is very much essential while making mapping decisions.

This chapter analyzes all the chosen OpenMP-based C programs from scientific HPC benchmarks from NAS (*bt, ft, cg, mg, lu, is, ep, sp*), SpecOMP 2001 (*equake, art, ammp*) and graph analytic workloads, from Green-Marl (*pagerank, triangle_counting, hop-dist*) that is specially designed for graph data analysis. The details of these programs are listed in Appendix A.

Figure 7.2 shows the scaling behaviour of chosen programs on an isolated 16-core machine as mentioned in Section 5.5. All compute-intensive programs utilize the increasing computing power to speedup the execution. On the contrary, memory-intensive programs spend most of their execution time on memory related operations that is a result of increased load/stores and increased last level ($L_2$) cache miss rates [Ebrahimi et al. 2011]. Hence, they do not have much impact of increased threads

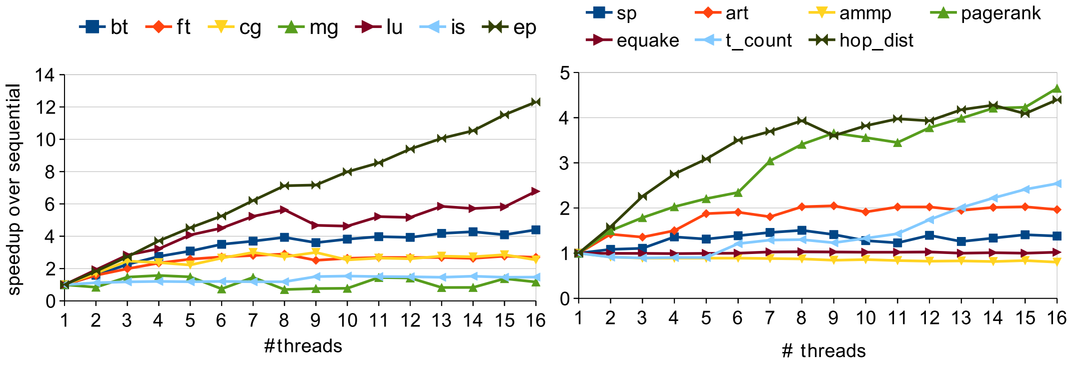


Figure 7.2: Scaling behaviour of benchmark programs on an isolated 16-core machine. Programs that scale well are compute-intensive that make most efficient use of increased processors for faster execution.

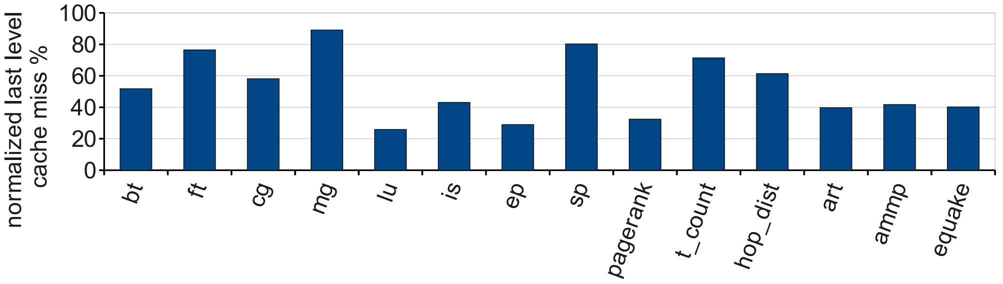


Figure 7.3: Plot showing last level cache misses of each program on an isolated 16-core machine. Higher the bar, more the time it spends on memory access and hence is memory- intensive.

on their execution times and they exhibit minute speedup or no speedup at all. In this work, programs that are disk-bound are included in the memory-intensive group. Memory access information of the programs is shown in Figure 7.3. This figure shows the percentage of last level cache $L_2$ misses normalized to the observed maximum value from these programs. This information is obtained from the hardware performance counters using *perf* tool on the same isolated 16-core system. Increased cache misses implies that the program spends most of the execution time in data fetch or store operations. Hence, it is unable to use the effective computing power to improve the performance.

Based on these characteristics, programs are classified as *compute-intensive* and *memory-intensive*. Programs `bt, lu, ep, pagerank, triangle counting, art` can be grouped as compute-intensive and `ft, cg, mg, is, sp, equake, ammp, hop dist` as memory-intensive.

## 7.5 Experimental Set-up

This section explains the hardware and software configuration employed for the experimental evaluation in this chapter. All experiments were run on a 16-core (2 sockets with 8 cores per socket) Intel Xeon E5530 2.40 GHz system with kernel 2.6.18. *gcc* 4.6 compiler with -O3 optimization level was used to compile all programs.

Both programs start executing at the same time and continue running till the other finishes. This set-up ensures that that the contention due to co-scheduling exists throughout. The experiments were repeated till all exhaustive combinations of program pairs using all combinations of policies are evaluated. Each experiment is run for 10 iterations for noise minimization.

### 7.5.1 Policy Evaluation

All combinations of the chosen scheduling policies were evaluated by co-executing a pair of programs. To observe how each scheduling policy affects a program from the perspective of the nature of the program three different sets of experiments were run based on behaviour of each program that is to be co-scheduled. Experiments in each of these categories are performed on all combinations of respective program pairs. The combinations of programs evaluated can be classified into:

- Compute-Intensive vs. Compute-Intensive
- Compute-Intensive vs. Memory-Intensive
- Memory-Intensive vs. Memory-Intensive

### 7.5.2 Pair-wise Program Evaluation

Here the impact of different scheduling policies is evaluated on the combined execution time when both jobs use the same *shared* policy. The performance results on per-program basis are presented in the next section. This study is to analyze how a program pair consisting of benchmark programs of different nature executes using the same policy. The *static* policy where each job uses one socket each to ensure minimum interference forms the baseline for this evaluation.

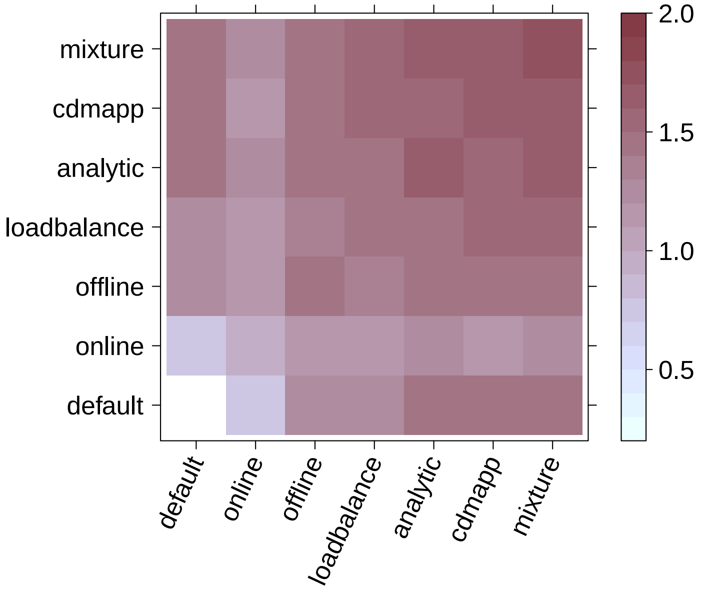


Figure 7.4: Speedup comparison of all policies averaged across all programs. Each box shows the improvement compared to a static policy. The overall speedup is improved over 1.68x when both jobs are co-scheduled with CDMapp and 1.71x by mixtures approach.

## 7.6 Analysis

This section provides a detailed analysis of competitive co-scheduling including comparison of policies. This classification is based on different combinations of program behaviour classified as described in Section 7.4. It further describes the observations from program-pair evaluations using a common shared policy.

### 7.6.1 Policy Comparison

Here the performance of different scheduling policies is analyzed when exactly two programs are co-scheduled. Every co-scheduled program uses a different policy based on the behaviour co-scheduled programs. Figure 7.4 shows a *heatmap* of the performance of exhaustive combinations of scheduling policies averaged across all pair of programs. Each box corresponds to either speedup improvement or slowdown of the combined program execution where one program employs a policy on the corresponding column and the other program employs a policy on the respective row. It can be observed that the default scheme worsens the combined performance of the programs leading to a slowdown up to 0.4x over baseline. When a program sticks to one policy there is a variation in the range of performance when another program changes its scheduling policy.

As observed from Figure 7.4 that shows the evaluation of all possible combinations of policies, the offline policy attains speedup improvement of 1.52x over baseline. Loadbalance scheme also performs nearly equal achieving 1.35x improvement. The analytic and CDMapp schemes perform still better, however the mixtures approach outperforms all by achieving 1.71x improvement. Now a detailed analysis of policy evaluation is presented when two programs of different behaviour are co-scheduled using all possible combination of policies.

**Compute-Intensive vs. Compute-Intensive:** Figure 7.5(a) shows the performance of policy combinations averaged across all pairs of compute-intensive programs. It can be observed that the default policy degrades the performance as in the baseline scheme, the programs make effective use of threads on individual sockets. Even though there is a potential for the programs to scale near-linear when they are assigned maximum threads, the effective number of threads that need to be scheduled are doubled which are queued until the resources are available. This process slowdowns the program execution. The online scheme tries to achieve an optimal allocation of resources by varying them in step-wise manner that leads to under-provisioning of resources to the processor-hungry programs. Hence, it achieves an improvement a little over the baseline. Loadbalance scheme improves over 1.4x as it achieves a near-optimal allocation of cores to co-scheduled programs. Offline policy achieves a further improvement attaining 1.54x as it determines the best allocation of threads to each program. The analytic scheme tries different thread configurations and selecting the best, achieves 1.58x. CDMapp and mixtures approach achieve nearly the same speedup of 1.61x and 1.6x respectively. A fine grain analysis of a program classified as compute-intensive can reveal if it achieves near-linear or sub-linear scale-up. This policy captures this *critical* program-centric behaviour along with the existing system contention and is hence able to achieve more efficient resource scheduling.

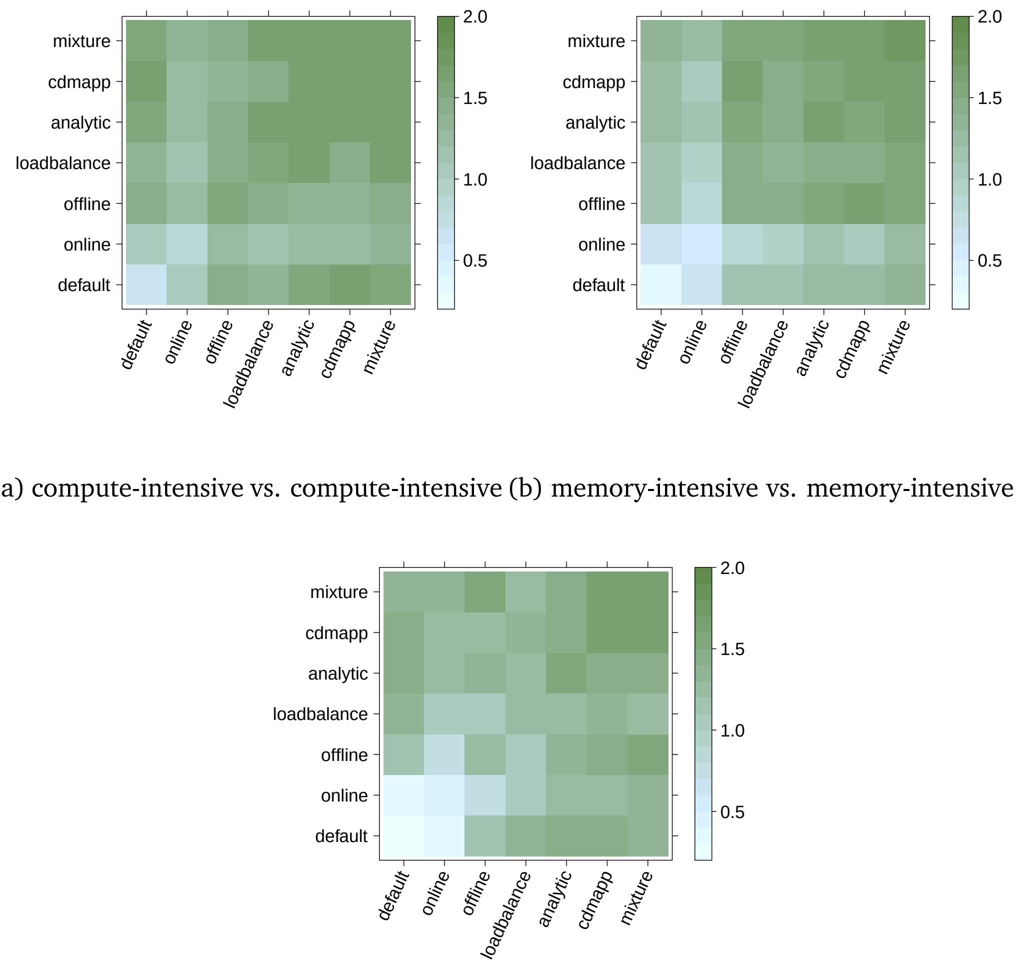
When each program uses a different policy, it is interesting to see how this combination of policies impact the overall performance. In Figure 7.5(a) the first column / last row of blocks (symmetric) shows the performance when the first program uses different scheduling policies while the second programs confines to the default scheme.

It can be observed that a combination of offline and default outperforms other combinations improving over 1.3x. Similarly when one program uses online policy only and the other is flexible with scheduling scheme, it can be observed that this combination improves over baseline barely owing to the lag to reach an optimal state by the online policy. Out of all possible policy combinations the CDMapp scheme drastically improves the performance by judiciously allocating optimal number of threads to each program and the default scheme significantly slows down both programs.

**Memory-Intensive vs. Memory-Intensive** When a memory-intensive program is co-executed with another similar program, it is crucial to know the optimal number of resources that each of the program needs. Since these programs do not scale well with increased processors, over-provisioning may harm the performance. Moreover, under-provisioning may lead to a drastic reduction in system utilization as more processors will remain idle. Programs of this behaviour usually scale up till a peak after which the performance drops. This tipping point varies with each program.

Figure 7.5(b) shows the average performance across all possible program pairs in this category. The default policy execution implies double the number of threads as the number of cores that are redundant as they do not speedup the program execution. Offline policy takes into consideration this specific program behaviour and tries to achieve a right balance of the amount of resources needed for each program. On average, this scheme improves 1.3x over baseline. When one program employs default scheme with the second program uses different schemes, the offline and loadbalance policies improve by 1.25x,1.3x on average. Loadbalance scheme marginally improves over the offline for these programs. However, the overall speedup is also increased when both programs use offline scheme attaining speedup over 1.32x. Surprisingly, when loadbalance is evaluated against the same scheme, it performs poorer than with loadbalance with default. Offline combined with analytic and CDMapp performs better than using the same policy. Policy combinations including analytic, CDMapp, mixture achieve outperform the rest. When both programs used same policy; CDMapp or mixture, it significantly the maximum speedup of over 1.62x.

**Compute-Intensive vs. Memory-Intensive** Figure 7.5(c) shows the performance of scheduling policies when a compute-intensive workload is co-scheduled with a memory-intensive program. A compute-intensive program can make effective use of



(a) compute-intensive vs. compute-intensive (b) memory-intensive vs. memory-intensive

(c) compute-intensive vs. memory-intensive

Figure 7.5: Performance comparison of mapping policies based on the different classes of program behaviour.

increased processors while the vast memory can be effectively used by the memory-intensive program. When offline policy is employed by both the programs, it improves over 1.42x over baseline. Surprisingly the analytic scheme achieves little improvement over offline, 1.49x. The mixtures shared policy outperforms all policy combinations by 1.63x.

Considering different policy combinations, offline and loadbalance achieve near-equal improvement when one program uses the default policy. Interestingly when both programs use online scheme, it performs worse than the shared default policy. As in previous cases, combinations of offline and loadbalance achieve great speedup

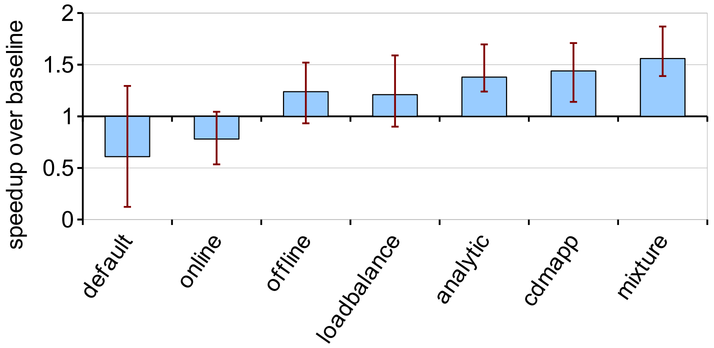


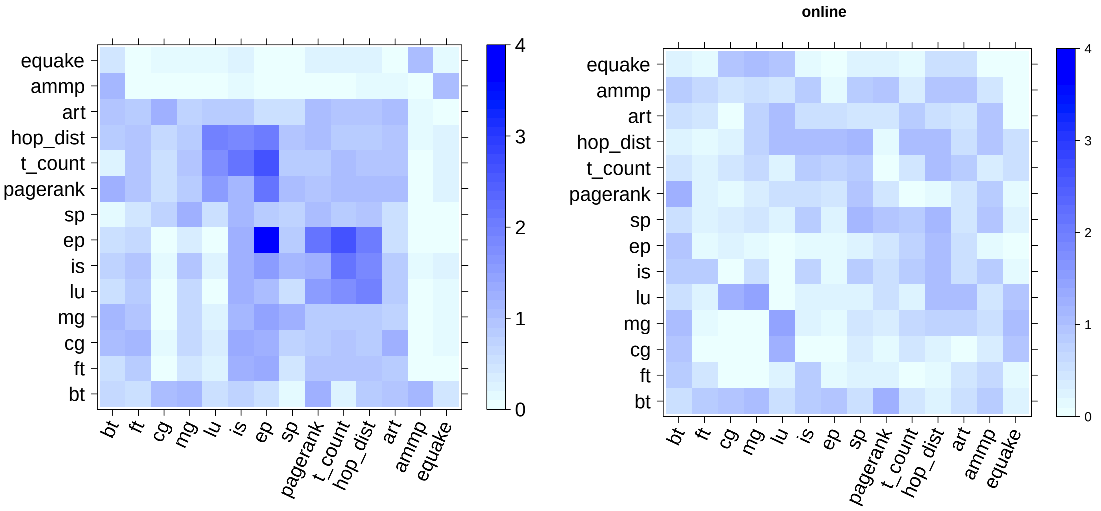
Figure 7.6: Speedup comparison of all policies averaged across all programs.

improvement of over 1.35x. In addition, this study also shows how different policies affect the stability of the system. An ideal policy reaches a stable execution point in minimal time by providing optimal allocation of resources to each of the co-scheduled program. CDMapp and mixture policies stabilize the system quicker than other schemes owing to the faster execution of the programs. The default and online policies pressurize the system for more resources resulting in an unstable execution state by delaying the program execution.

### 7.6.2 Pair-wise Program Comparison

Here the pair-wise performance of co-scheduled programs is evaluated when each program executes with same shared policy. This anaysis is to demonstrate how programs perform when they are co-executed with other programs when both use the same policy. Figure 7.6 shows the average performance of the pair of programs using same scheduling scheme averaged across all experiments.

By deploying the default scheme for both programs, the overall execution time is increased compared to a baseline static scheme and suffers a maximum degradation of 0.2x and barely improving for certain program pairs up to 1.2x. The online scheme performs worse and has a significant drop in speedup and rarely improving over the baseline. It slows down by 0.45x to 0.9x. Offline does not penalize the co-executing program. Hence, it improves significantly reaching up to 1.52x speedup. It slows down up to 0.92x for certain program pairs that are either aggressive or sensitive. The loadbalance policy also outperforms the baseline by achieving 1.35x improvement and occasionally slows down when it is unable to find the ideal number of threads required by either of the programs or both. The overall speedup is improved over 1.48x, 1.52x when both jobs are co-scheduled using CDMapp, mixture policies and next by analytic



(a) default vs. default

(b) online vs. online

Figure 7.7: Performance of a program pair using same policies. Darker shades highlight improved speedup (>1) and lighter shades depict slow downs (<1). (a) Overall the default policy slows down the programs barring certain combinations involving `ep` (b) When both programs use online policy the overall improvement is minimal.

by 1.44x. The speedup improvements using the CDMapp and mixture policies are coming from providing more appropriate numbers of threads to different processes.

Figures 7.7 to 7.10 show a detailed comparison of per-program performance when a pair of programs chosen from the benchmarks is co-executed using same shared policies. Each block represents the combined execution time of both target and workload and is normalized to the baseline static policy where both target and workload execute with on one socket each. A value greater than 1 implies speedup improvement and vice-versa. Darker shades highlight improved speedup (>1) and lighter shades depict slow downs (<1).

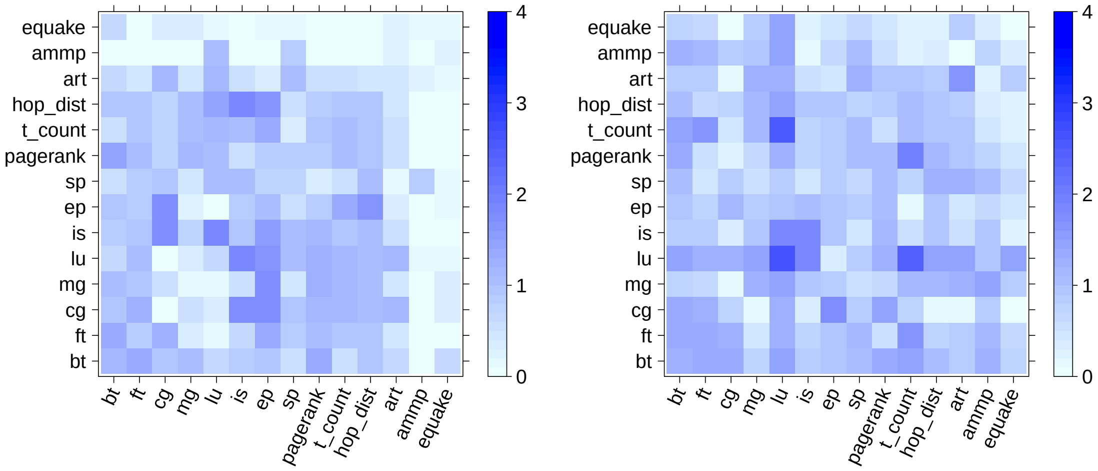
**OpenMP Default:** The OpenMP default policy assigns threads equal to the number of available processors. When the pair of co-scheduled jobs deploys this policy, there are more threads than the H/W contexts. This accumulation increases the overall resource contention in the system. As observed from Figure 7.7(a), compute-intensive programs like `ep`, `lu`, `pagerank`, `art` improve their performance with increased number of threads owing to more parallel computations in the program with minimal data transfers. Hence, they are able to exploit a large number of threads for achieving maximum speedup.

**Online:** In the online approach, the model determines thread number in response to the loop execution time and changes the value in unit steps if the previous decision improves the parallel section's performance. Due to frequent change in non-optimal thread number the program spends much of the crucial execution time trying to reach the optimal solution. This policy does not take the system contention into consideration for determining optimal thread number. From Figure 7.7(b) it can be observed that this scheme meagrely improves the speedup for certain pairs involving `lu, hop_dist, pagerank`. It drastically affects both the co-scheduled programs thus leading to a reduction in overall execution time by 22% compared to the baseline.

**Loadbalance:** The loadbalance policy finds to provide adequate resources to each of the co-scheduled jobs. Hence, it achieves good speedup improvement over the baseline scheme. However, for some pair of jobs, the exact loadbalance is non-optimal. When a compute-intensive job co-executes with a non-compute-intensive job some of these can be reassigned to a compute-intensive task which utilizes them more efficiently, as the latter does not benefit from the running on more processors. This policy improves speedup for most of the program pairs but slows down in few program pair combinations that include `ammp, equake,cg`. On average, the shared loadbalance policy improves over 1.35x over the baseline policy and is shown in Figure 7.8(a).

**Offline:** When both target and workload employ offline policy, the model predicts the optimal thread number for each co-executing program with neither under-provision nor over-provision of resources. It thus helps to reduce the combined execution time significantly with efficient resource utilization. For certain program combinations that include `bt, ft, lu, is` and `pagerank` in the program pair, this policy improves speedup over 1.3x but slows down in certain program pairs with `cg, equake` and `ep`. As seen from Figure 7.8(b), this shared policy improves over 1.5x over the baseline static scheme.

**Analytic:** The analytic policy adjusts thread numbers based on trying different thread configurations. It is able to reduce the system load as the resource contention is captured in the loop execution time captured at every decision point. For program pairs that consist `art, lu, t_count`, this policy achieves great improvement. It achieves barely for certain program configurations including `equake, ammp, cg`. On average,



(a) loadbalance vs. loadbalance

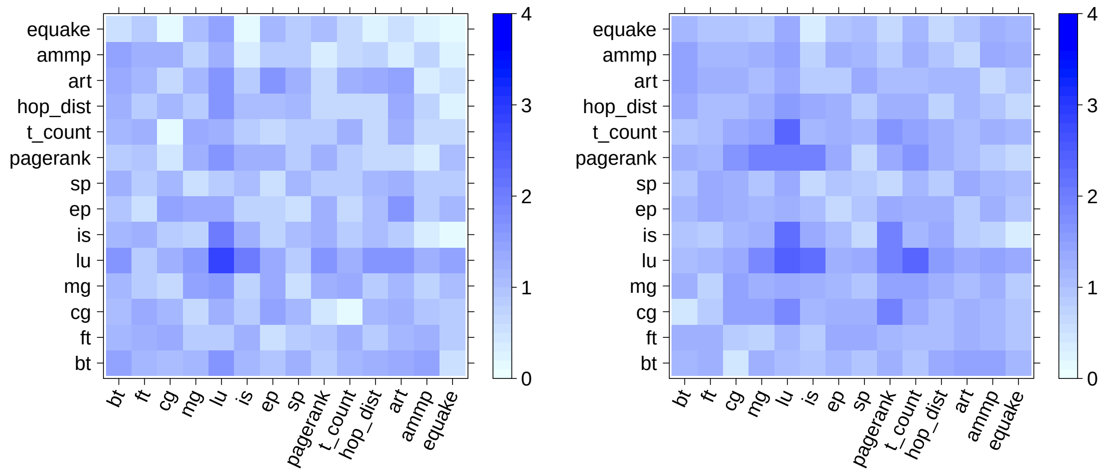
(b) offline vs. offline

Figure 7.8: (a) Loadbalance improves overall performance of both programs on average. However, it drastically degrades for programs with `ammp` (b) When programs execute using same share offline policy, the overall speedup is greatly improved, specifically with compute intensive programs involving `lu`. Overall most blocks have darker shades implying speedup improvement across wide range of benchmarks.

the analytic policy improves 1.53x over the baseline scheme. The results are shown in Figure 7.9(a).

**CDMapp:** CDMapp policy uses the Offline scheme for predicting ideal thread number. It uses an environment predictor as a feedback for monitoring this scheme and adjusts the thread number, if necessary. It can adjust to the changing circumstances and correcting the mispredictions of thread predictor. It improves significantly for programs that include `lu, t_count`. Overall the speedup is greatly improved for majority of program combinations. However, for programs `equake, cg` the range of increase in speedup is low. Figure 7.9(b) shows that this shared policy attains 1.65x improvement over the baseline scheme.

**Mixture:** This technique selects the best expert based on the current execution. Since the experts are built considering the scale-up behaviour of the programs, they are more optimized for specific programs. The online model selector learns and selects the optimal policy. For example, when a program-pair is a combination of a compute-



(a) analytic vs. analytic

(b) CDMapp vs. CDMapp

Figure 7.9: (a) On average, Analytic technique improves speedup for all program combinations. It achieves best improvement for certain program-pair containing `lu` and worst for `equake, ammp`. (b) CDMapp greatly improves speedup for all program combinations outperforming other policies.

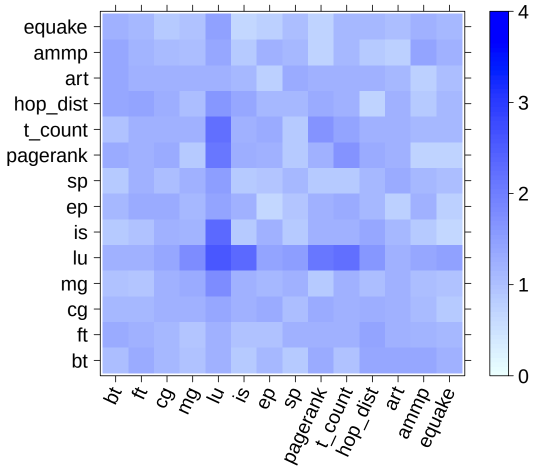
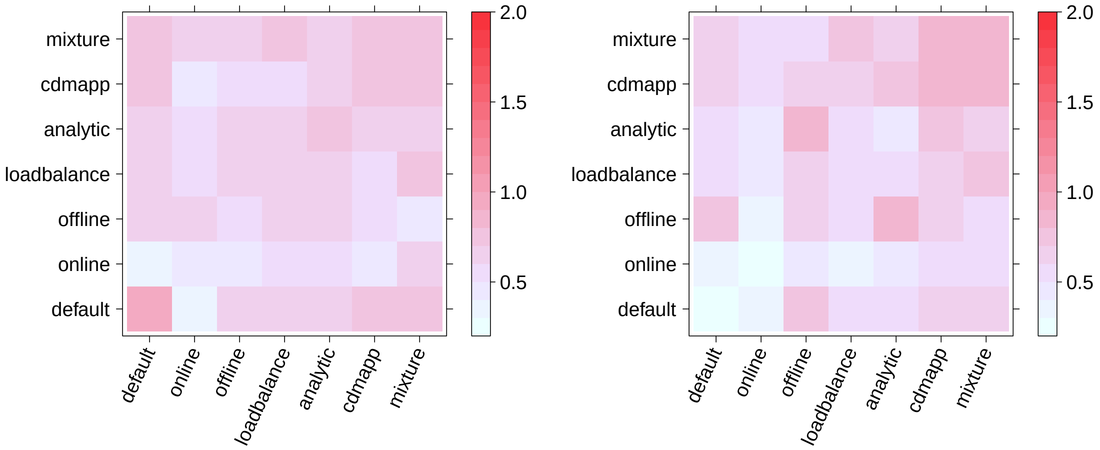


Figure 7.10: On average, the mixture of experts policy improves best speedup for almost all program combinations. It achieves best improvement for certain program-pair containing `lu, pagerank, t_count` and worst for `equake, ammp, ep`.

intensive and a memory-intensive programs, this approach determines best expert for each program. The expert policy which is fine-tuned for a compute intensive program is chosen for it and a different expert for the other program. It thus is able to satisfy both the programs by determining the best thread counts. The mixture policy achieves



(a) performance of ep averaged across all workloads

(b) performance of equake averaged across all workloads

Figure 7.11: (a) ep of NAS benefits from increased number of threads irrespective of co-executing workload. OpenMP default policy outsmarts all other existing policies. Sequential policy performs the worst. (b) equake performs worse with any policy.

best improvement for certain program-pair containing lu, pagerank, t_count and worst for equake, ammp, ep. Figure 7.10 highlights these results.

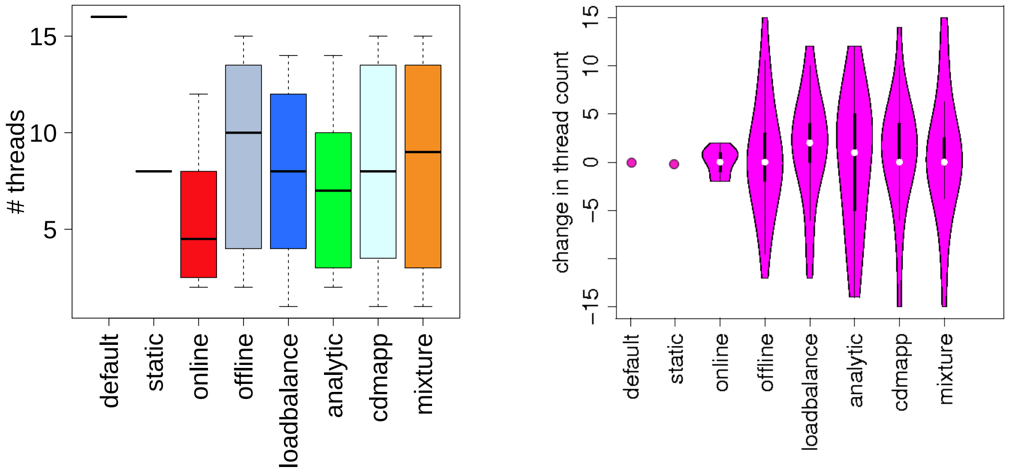
### 7.6.3 Outliers

There are certain *outlier* programs that aggressive and scale extremely well with processors irrespective of co-executing workloads or do no benefit at all from any thread configurations and do not scale with the number of processors. ep from NAS and equake from SpecOMP are programs that can be classified in this category.

As seen from Figure 7.11(a) ep slows down by 0.32x, 0.78x, 0.93x by online, loadbalance and offline policies. The default scheme improves it by 1.8x. Similarly, as seen in Figure 7.11(b), for equake, default and online tend to suffer most by degrading by 0.3x, 0.25x. None of the evaluated policies improve over static. Few combinations that include analytic and CDMapp try to match the static scheme performance.

### 7.6.4 Source of Improvement

This section gives insights into why the offline policy outperforms other scheduling policies. It determines the optimal number of threads for the pair of programs so that it benefits both the programs.



(a) number of threads determined by policies (b) distribution of how frequently thread numbers change

Figure 7.12: (a) Number of threads assigned by each policy to all exhaustive combinations of programs. (b) Distribution of the frequency of thread number changes.

Figure 7.12(a) shows the distribution of the number of threads assigned by the policy averaged across all programs. It shows the range of threads as well as the normalized distribution. It can be observed that default and static schemes determine the same thread number always. The online scheme restricts to a smaller range with a mean of 5 threads. The offline technique spans a wide range of thread numbers with a mean thread number 10. Loadbalance and analytic policies determine most of the thread numbers in smaller ranges. CDMapp and mixture approaches span a wide range of thread numbers. They differ in the mean thread numbers.

Next, the frequency in change in thread numbers assigned by different policies is plotted in Figure 7.12(b). A frequent change in thread number is not ideal as it involves frequent movement of data across caches where this delay could overshadow the chances of an increased speedup. Since the default and static schemes are always constant, the difference of thread numbers is 0. Online scheme varies thread numbers in small unit steps hence the thread number difference is restricted to a small range [-2, 2]. The CDMapp and mixture policies stay at the best thread number once found and does not change limiting to drastic thread number changes. There are instances when there is a direct thread number change as observed from the black line in the offline bar. The bulge at the center of the bar shows that this policy restricts to minimal

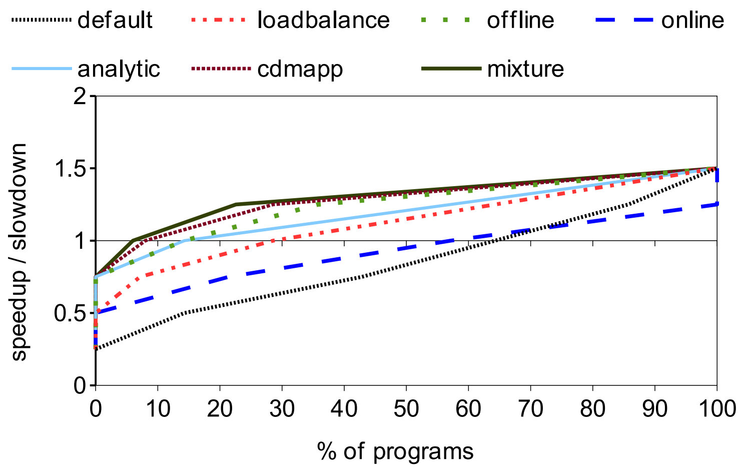


Figure 7.13: Plot showing the percentage of programs that gain speedup or slowdown due to various policies.

thread number change. Analytic policy also changes to ideal thread numbers; however, it does not change to the ideal thread numbers as frequently as the offline scheme.

Furthermore, a quantitative study of how many programs speedup or slowdown using the aforementioned scheduling policies is presented in the next section. Figure 7.13 shows a plot of the percentage distribution of programs on x-axis and their speedups on y-axis. Speedup value greater than 1 implies that the program benefits from the scheduling policy in reducing its execution time and vice-versa.

As observed from this figure, CDMapp and mixture approaches prove to be the best mapping policies for at least 90% and 91% of evaluated program combinations. Only a minute fraction do not benefit vastly employing these policies. These are followed by the analytic scheme that improves 88% of programs, slowing down 12% programs respectively. The offline policy slows down only 13% of the programs. Loadbalance policy improves performance of around 72% of programs and slows down around 28% of programs. The online scheme slows down more than half of the program-set, 52% owing to its weak scheduling mechanism. The default policy performs the worst of all by slowing down at least 65% of the programs.

## 7.7 Summary

This chapter has presented different state-of-the-art scheduling policies with a detailed analysis of each policy. Also, programs are analyzed based on their behaviour and the impact of different policies for competitive co-scheduling of jobs are studied. A detailed policy evaluation is described and per-program comparison to observe how these policies affect the programs performance based on their behaviour. From the results obtained, it can be observed that programs of different behaviour benefit mostly by using a common policy. However, there are cases where different policy combinations outperform a shared policy. Next it can be deduced that when both co-executing programs employ the same policy, the mixture of experts and CDMapp schemes outperform other policies. They improve the performance of the co-scheduled jobs over 1.71x and 1.68x respectively. Finally, there are few programs that do not benefit no matter what mapping policy they use.

The next chapter concludes this thesis. It provides a summary of the contributions followed by a critical analysis of the techniques used in this work and identified challenges and pitfalls. It then lists out few research areas where this work may be extended.

# Chapter 8

# Conclusions

This thesis focused on adaptive parallelism mapping, i.e., tuning thread numbers for a parallel program, when it co-executes with workloads in a highly dynamic system. Such an approach has no impact on external workloads, rather, it improves them as well. This chapter begins with a summary of main contributions in Section 8.1, followed by a review of few challenges and pitfalls in Section 8.2. Next, a detailed analysis of the employed techniques is presented in Section 8.3 and finally, this chapter concludes with potential future directions in Section 8.4.

## 8.1 Summary of Contributions

The primary contribution of this thesis is determining the best mapping for parallel programs in dynamic execution environments that minimizes execution time. It does so by using lightweight predictive models that are trained offline. The ideal thread numbers determined avoid either extremes: under-subscription and over-subscription of resources.

### 8.1.1 Adaptive Parallelism Mapping

Chapter 4 has presented a predictive modelling-based model *thread-predictor*, for predicting the optimal thread count. The underlying machine learning heuristic considers program behaviour in terms of static code features and runtime system information in terms of dynamic features as input. Based on the training algorithm, it then outputs the best thread number. Every parallel section first invokes the deployed model and resumes the execution with the predicted thread number. Such an automatic approach can easily be ported to new machines and deployed for new parallel programs.

This approach was evaluated against an online technique [Raman et al. 2012], that changes the thread count in response to the system load using a hill-climbing algorithm. Unlike this technique which is slow to react to system changes, the proposed model determines quickly the best thread number. On evaluating representative parallel benchmark programs on a 12-core Intel Xeon system, this model was able to improve over the online scheme by 1.5x speedup improvement.

### 8.1.2 Exploiting Offline Models and Online Adaptation

Machine learning-based models are, in general, trained in a limited state space of potential execution scenarios. Prior anticipation of all possible scenarios and corresponding training is highly infeasible. So, when such a model is deployed in scenarios for which it was not trained for, it is likely to make wrong decisions. Moreover, a lack of online monitoring mechanisms allow it to continue with mispredictions.

This work proposed a solution to adapt the mapping as and when required. It initially exploits the thread-predictor and uses an *environment-predictor* to monitor its performance. This innovative technique predicts what the future environment should look like if the current decision is optimal. By comparing the observed and predicted environments, change in the system can be detected. The observed difference is then used as a feedback to adjust mapping, thus adapting to changes in the execution environment. This novel approach is presented in Chapter 5. When evaluated on a 16-core Intel Xeon system with dynamic workloads and hardware resources, this technique achieved speedup improvement of 2.14x over OpenMP default, 1.58x over an online approach and 1.32x over the thread-predictor model.

### 8.1.3 Online Learning of Best Policy Selection

Parallelism mapping policies are usually characterised by a one-size fits all assumption. They have a single monolithic policy that matches a program to its parallel environment. There is little ability to determine if the policy fits the existing setting with lack of methods to easily extend to incorporate additional knowledge.

This chapter has presented a technique to tackle this scenario based on a mixture of experts approach. It determines at runtime the best offline mapping policy or expert out of a collection of experts. Every expert predicts a thread number and future environment at every decision point. A model to select the best expert is learnt online based on the accuracy of predictions of individual experts. It then determines that the thread number predicted by the selected expert is optimal for that scenario. This mechanism also allows for including additional expertise knowledge. Any policy that can predict future environment along with thread number can be included in the mixture.

Chapter 6 presented this work in detail. When evaluated on a 32-core Intel Xeon platform with highly dynamic workloads and hardware, this approach improved over 1.69x over OpenMP default and outperforms an analytic policy [Sridharan et al. 2014] by 1.21x speedup improvement.

## 8.2 Challenges and Pitfalls

Here we describe few shortcomings of this work and ways on how the proposed techniques could have been improved. It summarizes potential pitfalls and challenges.

All proposed techniques were evaluated on Intel Xeon systems. For robustness and sanity checking, similar experiments could have been performed on other architectures such as multi-socket Haswell, SPARC M7, IBM Cell. These experiments could have revealed interesting insights into the impact of thread count tuning on the performance. If the set of training runs included experiments on multiple architectures, other system parameters which are critical to the machine learning models could have been identified. The accuracy of the machine learning models was in the range of 80-90%. These values could have been improved further with richer input data from better training experiments.

Next, the experimental set-up could have been more diverse. For example, a scenario could include a program executing on a low power hand-held device with minimal resources ported to a massive cluster with enormous computing power.

As mentioned in Chapter 6, there are numerous ways of building experts. In this work four experts were selected arbitrarily, as the optimal number of required experts is unknown in advance. Currently as observed from Section 6.7, the average environment prediction accuracy of each expert is around 80%. An alternate design for classifying and building the experts may have further improved respective prediction accuracies.

## 8.3 Analysis

This section presents an analysis of the methods used in this work, with focus on machine learning techniques, training costs and alternate parallel programming models.

### 8.3.1 Machine learning Models

Throughout this thesis, all proposed mapping approaches are built using machine learning techniques where deciding what algorithm to use is a critical issue. There are many models using different algorithms to choose from, such as decision trees, support vector machines, regression techniques, neural networks, random forests etc. For a given training data set, the accuracy of each model is not the same. Also the time taken for training and the overhead in its implementation at runtime also varies.

Ideally, a machine learning model should have highest prediction accuracy, take least training time, and add minimal overhead. For example, *Boosting* is effective when the training data set is very large. *K-nearest neighbors* are often effective but require lot of memory and are slow. *Neural networks* are slow to train but very fast to run. *SVMs* are best with limited data but lose out to *boosting, random trees* when large data sets are available. This thesis used models learnt using Artificial neural networks and regression techniques that were selected based on highest prediction accuracies attained from the training data sets and lightweight execution.

### 8.3.2 Training Costs

The accuracy of machine learning models rely on the quality of the training data. The generation of training data is a one-off cost, usually done offline before deployment. The central issue often, is the generation of rich training data that can be used to build highly accurate models. The training data is a collection of static code features and runtime system features. The feature extraction of code features is relatively simpler. Identifying the optimal configuration that forms the *labelled* data involves some effort in the form of various training experiments.

Also, identifying representative programs that exhibit different characteristics is crucial. Programs of similar nature do not reveal diverse inherent characteristics as programs with varied features do. Next factor is the time taken for exploring multiple thread configurations and identifying the optimal one. The time consumed varies based on the number of benchmark programs and different thread combinations. Exhaustive evaluation of all possible thread number changes is highly expensive. Another challenge is the random sampling of experiments that contribute to rich training data. Techniques such as active learning can come handy in identifying the best training data set which is just sufficient for highly accurate models.

### 8.3.3 Alternate Parallel Programming Models

This thesis optimizes programs that use OpenMP. It tweaks the default OpenMP compiler to change the thread number at runtime. The choice of using OpenMP was (a) existence of libraries to change thread count (using *num_threads*) parameter and (b) presence of relatively easy techniques to dynamically change this parameter with minimal programmer effort. Achieving the same for other programming models such as MPI, Pthreads requires more effort. As such, any model that allows for adjusting thread number at runtime can use the models developed in this work.

## 8.4 Future Work

The work proposed in this thesis may be extended to tackle additional challenges. This section presents few potential future directions that could be explored based on the proposed solutions.

### 8.4.1 Multi-objective Optimization

Throughout this thesis, a program is optimized for minimizing its execution time. Modern day hardware demand energy-efficient solutions apart from program performance. Much of the existing work focuses on either performance or power. Optimizing both goals at the same time is a complex task. Optimizing one goal may interfere with the other and de-stabilize the system. Here, determining the pareto-optimal curve is required, where a mapping solution improves both performance and power.

### 8.4.2 Multi-configuration Tuning

This thesis optimized the thread count for parallel programs based on program characteristics and runtime information. There are other tunable parameters for faster execution of programs such as thread affinity-based scheduling, dynamically changing processor frequencies, enhancing the default schedule.

Here the challenge is deciding: (1) what to tune and (b) how much to tune. Identifying what tunable parameters have significant impact on program is essential as each parameter has different impact on the performance. Next issue is determining how tuning one or multiple parameters may prove beneficial in a given execution scenario. Here, what magnitude should each parameter be changed is also critical. Machine learning can be used in identifying the weights of each tunable parameter. For example, for a program with varying phase behaviour, once next phase is identified to not scale on current hardware, the processors can be re-allocated to co-executing programs and/or lowering the processor frequencies.

### 8.4.3 Reinforcement Learning-based Mapping

Reinforcement learning is the problem of enabling an agent to act in the world to maximize its rewards. It is an advanced machine learning method where a model *learns* on-the-fly, based on observed state parameters. In parallel scheduling domain, few attempts have been made to employ techniques using reinforcement learning. A scheduler, initially, can use a heuristic that is learnt offline. While the program is executing, training data that is collected, that is used to re-learn the offline heuristic. In regression terms, this is equivalent to adjusting weights of the input variables and finding the optimal change in those weights. This approach can adapt to multiple changing system parameters and can make more efficient mapping decisions.

### 8.4.4 Mapping with Heterogeneity

This thesis focussed on optimizing parallel programs on multi-core platforms. Heterogeneous hardware and corresponding massively parallel programs add more complexity in determining ideal mapping decisions. Most existing work focuses on task partitioning and porting that work on either CPU or GPU. However, this process is challenging when multiple programs co-execute on heterogeneous processors. Machine learning techniques can be employed here as well to determine ideal task partitioning, mapping and scheduling in dynamic environments.

## 8.5 Summary

This chapter concludes the thesis. It has summarized the main contributions to the area of adaptive parallelism mapping in dynamic environments. A critical analysis of the proposed techniques and shortcomings followed by ideas for future work have been discussed.

# Appendix A

# Benchmarks used for Evaluation

Tables A.1 and A.2 list the programs from different benchmarks suites used for evaluation in this thesis. These are: **NAS** [Bailey et al. 1991], [NAS], **SpecOMP** [SpecOMP], **Parsec** [Bienia 2011], [Parsec] and **Green-Marl** [Hong et al. 2012], [Green-Marl]. The representative parallel programs consisting of compute, disk and memory-bound programs are diverse with emerging workloads from various domains. For example, `blackscholes` from Parsec is a financial application which computes options pricing using Black-scholes partial differential equations. SpecOMP programs target high performance computing (HPC) domain. NAS programs are derived from computational fluid dynamics (CFD) applications. Such a selection of programs ensures that the proposed solutions are evaluated on a wide variety of programs.

| Benchmark suite | Program | Input size | Notes |
|---|---|---|---|
| NAS | IS | Class D:<br>no. of keys = $2^{31}$<br>key max. value =$2^{27}$ | Integer Sort<br>random memory access |
| NAS | EP | Class D:<br>no. of random-number pairs = $2^{36}$ | Embarrassingly Parallel |
| NAS | CG | Class D:<br>no. of rows = 1500000<br>no. of nonzeros = 21<br>no. of iterations = 100<br>eigenvalue shift = 500 | Conjugate Gradient<br>irregular memory access and communication |
| NAS | MG | Class D:<br>grid size = 1024 x 1024 x 1024<br>no. of iterations = 50 | Multi-Grid on a sequence of meshes, long and short-distance communication,<br>memory-intensive |
| NAS | FT | Class D:<br>grid size = 2048 x 1024 x 1024<br>no. of iterations = 25 | discrete 3D fast Fourier Transform,<br>all-to-all communication |
| NAS | BT | Class D:<br>grid size = 408 x 408 x 408<br>no. of iterations = 250<br>time step = 0.00002 | Block Tri-diagonal solver |
| NAS | SP | Class D:<br>grid size = 408 x 408 x 408<br>no. of iterations = 500<br>time step = 0.0003 | Scalar Penta-diagonal solver |
| NAS | LU | Class D:<br>grid size = 408 x 408 x 408<br>no. of iterations = 300<br>time step = 1.0 | Lower Upper-Gauss-Seidel solver |

Table A.1: NAS Parallel Benchmark programs

| Benchmark suite | Program | Input size | Notes |
|---|---|---|---|
| SpecOMP | `320.equake` | medium | Finite element simulation; earthquake modelling |
| SpecOMP | `330.art` | medium | Neural network simulation; adaptive resonance theory |
| SpecOMP | `332.ammp` | medium | Computational Chemistry |
| Parsec | `blackscholes` | simlarge:<br>65,536 options | Option pricing with Black-Scholes<br>Partial Differential Equation |
| Parsec | `bodytrack` | simlarge:<br>4 cameras, 4 frames<br>4,000 particles, 5 annealing layers | Body tracking of a person |
| Parsec | `freqmine` | simlarge:<br>Database with 990,000 click streams<br>minimum support 790 | Frequent itemset mining |
| GreenMarl | `Pagerank` | graph of 1m nodes, 8m edges | Algorithm computing pagerank of each node, i.e. its importance |
| GreenMarl | `triangle counting` | graph of 1m nodes, 8m edges | Computes the number of closed triangles |
| GreenMarl | `hop-dist` | graph of 1m nodes, 8m edges | Computes the distance of every node from a root node |

Table A.2: SpecOMP, Parsec, GreenMarl Benchmark programs

# Bibliography

[NAS] NAS parallel benchmarks 2.3, OpenMP C version. `http://www.nas.nasa.gov/publications/npb.html`.

[Agakov et al. 2006] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA. IEEE Computer Society. Available from: `http://dx.doi.org/10.1109/CGO.2006.37`.

[Ansel et al. 2009] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1542476.1542481`.

[Ansel et al. 2012] Ansel, J., Pacula, M., Wong, Y. L., Chan, C., Olszewski, M., O'Reilly, U.-M., and Amarasinghe, S. (2012). Siblingrivalry: Online Autotuning Through Local Competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 91–100, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2380403.2380425`.

[Arnold et al. 2005] Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. (2005). A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2). special issue on "Program Generation, Optimization, and Adaptation".

[Bailey et al. 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber,

R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS Parallel Benchmarks; Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91.

[Balasundaram et al. 1991] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. (1991). A static performance estimator to guide data partitioning decisions. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 213–223, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/109625.109647`.

[Basseville and Nikiforov 1993] Basseville, M. and Nikiforov, I. V. (1993). *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Bhadauria and McKee 2010] Bhadauria, M. and McKee, S. A. (2010). An Approach to Resource-aware Co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1810085.1810113`.

[Bhattacharyya and Hoefler 2014] Bhattacharyya, A. and Hoefler, T. (2014). PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime. In *Accepted at the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. ACM.

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

[Bishop 1995] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA.

[Bishop 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc.

[Blagodurov et al. 2011] Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A. (2011). A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA. USENIX Association. Available from: `http://dl.acm.org/citation.cfm?id=2002181.2002182`.

[Blumofe et al. 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/209936.209958`.

[Blumofe and Leiserson 1999] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling Multithreaded Computations by Work Stealing. *Jorunal of the ACM*, 46(5):720–748. Available from: `http://doi.acm.org/10.1145/324133.324234`.

[Boser et al. 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/130385.130401`.

[Boyer et al. 2013] Boyer, M., Skadron, K., Che, S., and Jayasena, N. (2013). Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2482767.2482794`.

[Breslow et al. 2013] Breslow, A. D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D. M., and Snavely, A. E. (2013). The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*.

[Carriero et al. 1995] Carriero, N., Freeman, E., Gelernter, D., and Kaminsky, D. (1995). Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49. Available from: `http://dblp.uni-trier.de/db/journals/computer/computer28.html#CarrieroF%GK95`.

[Chamberlain et al. 2007] Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312. Available from: `http://dx.doi.org/10.1177/1094342007078442`.

[Chandramohan and O'Boyle 2014] Chandramohan, K. and O'Boyle, M. F. (2014). Partitioning Data-parallel Programs for Heterogeneous MPSoCs: Time and Energy

Design Space Exploration. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 73–82, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2597809.2597822.

[Charles et al. 2005] 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 *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1094811.1094852.

[Chen et al. 2010] Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 448–459, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1806596.1806647.

[Coons et al. 2008] Coons, K. E., Robatmili, B., Taylor, M. E., Maher, B. A., Burger, D., and McKinley, K. S. (2008). Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008*, pages 32–42. Available from: http://doi.acm.org/10.1145/1454115.1454122.

[Corbalán et al. 2000] Corbalán, J., Martorell, X., and Labarta, J. (2000). Performance-driven processor allocation. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 5–5, Berkeley, CA, USA. USENIX Association. Available from: http://portal.acm.org/citation.cfm?id=1251229.1251234.

[Cortes and Vapnik 1995] Cortes, C. and Vapnik, V. (1995). Support-Vector Networks. *Journal of Machine Learning*, 20(3):273–297. Available from: http://dx.doi.org/10.1023/A:1022627411411.

[Ţăpuş et al. 2002] Ţăpuş, C., Chung, I.-H., and Hollingsworth, J. K. (2002). Active harmony: towards automated performance tuning. In *Proceedings of the 2002*

*ACM/IEEE conference on Supercomputing*, SC '02, Los Alamitos, CA, USA. IEEE Computer Society Press. Available from: http://dl.acm.org/citation.cfm?id=762761.762771.

[Curtis-Maury et al. 2006] Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D., and Nikolopoulos, D. S. (2006). Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1183401.1183426.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55. Available from: http://dx.doi.org/10.1109/99.660313.

[Dave and Eigenmann 2009] Dave, C. and Eigenmann, R. (2009). Automatically Tuning Parallel and Parallelized Programs. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 126–139.

[Delimitrou and Kozyrakis 2014] Delimitrou, C. and Kozyrakis, C. (2014). Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2541940.2541941.

[Dey et al. 2013] Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. (2013). Resense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4):41:1–41:25. Available from: http://doi.acm.org/10.1145/2555289.2555298.

[Diniz and Rinard 1997] Diniz, P. C. and Rinard, M. C. (1997). Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97.

[Draper and Smith 1981] Draper, N. R. and Smith, H. (1981). Applied regression analysis 2nd ed.

[Dubach et al. 2009] Dubach, C., Jones, T. M., Bonilla, E. V., Fursin, G., and O'Boyle, M. F. P. (2009). Portable Compiler Optimisation Across Embedded Programs and Microarchitectures Using Machine Learning. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 78–88, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1669112.1669124.

[Ebrahimi et al. 2011] Ebrahimi, E., Miftakhutdinov, R., Fallin, C., Lee, C. J., Joao, J. A., Mutlu, O., and Patt, Y. N. (2011). Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 362–373. ACM.

[ECDF] ECDF. Edinburgh compute and data facilities. http://www.ed.ac.uk/schools-departments/information-services/services/r%esearch-support/research-computing/ecdf/.

[Edakunni et al. 2011] Edakunni, N. U., Brown, G., and Kovacs, T. (2011). Boosting as a Product of Experts. In *27th Conference on Uncertainty in Artificial Intelligence, UAI 2011*.

[Emani and O'Boyle 2014] Emani, M. K. and O'Boyle, M. (2014). Change Detection based Parallelism Mapping: Exploiting Offline Models and Online Adaptation. *International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2014*.

[Emani and O'Boyle 2015] Emani, M. K. and O'Boyle, M. (2015). Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. *ACM SIGPLAN conference on Programming Language Design and Implementation, 2015*.

[Emani et al. 2013] Emani, M. K., Wang, Z., and O'Boyle, M. F. P. (2013). Smart, Adaptive Mapping of Parallelism in the Presence of External Workload. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 1–10. Available from: http://doi.ieeecomputersociety.org/10.1109/CGO.2013.6495010.

[Eyerman and Eeckhout 2010] Eyerman, S. and Eeckhout, L. (2010). Probabilistic job symbiosis modeling for smt processor scheduling. In *Proceedings of the Fifteenth Edi-*

*tion of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 91–102, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1736020.1736033`.

[Eyerman and Eeckhout 2014] Eyerman, S. and Eeckhout, L. (2014). The Benefit of SMT in the Multi-core Era: Flexibility Towards Degrees of Thread-level Parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 591–606, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2541940.2541954`.

[Fang et al. 1990] Fang, Z., Tang, P., Yew, P.-C., and Zhu, C.-Q. (1990). Dynamic Processor Self-Scheduling for General Parallel Nested Loops. *IEEE Transactions on Computers*, 39(7):919–929. Available from: `http://dx.doi.org/10.1109/12.55693`.

[Fursin et al. 2008] Fursin, G., Cupertino, Miranda, Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and O'Boyle, M. (2008). MILEPOST GCC : Machine Learning Based Research Compiler. In *GCC Developers Summit*.

[Ghahramani 2004] Ghahramani, Z. (2004). Unsupervised learning. In *Advanced Lectures on Machine Learning*, pages 72–112. Springer.

[Gordon et al. 2006] Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGARCH Computer Architecture News*, 34(5):151–162. Available from: `http://doi.acm.org/10.1145/1168919.1168877`.

[Gordon et al. 2002a] Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. A., Leger, C., Wong, J., Hoffmann, H., Maze, D., and Amarasinghe, S. (2002a). A stream compiler for communication-exposed architectures. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 291–303, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/605397.605428`.

[Gordon et al. 2002b] Gordon, M. I., Thies, W., Karczmarek, M., Wong, J., Hoffman, H., Maze, D. Z., and Amarasinghe, S. (2002b). A Stream Compiler for Communication-Exposed Architectures. Technical Report MIT/LCS Technical Memo

LCS-TM-627, Massachusetts Institute of Technology, Cambridge, MA. Available from: http://groups.csail.mit.edu/commit/papers/02/streamit-627-v2.pdf.

[Grama et al. 2003] Grama, A., Karypis, G., Kumar, V., and Gupta, A. (2003). *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition. Available from: http://www.worldcat.org/isbn/0201648652.

[Green-Marl] Green-Marl. A domain specific language for graph data analysis. https://github.com/stanford-ppl/Green-Marl.

[Grewe et al. 2011] Grewe, D., Wang, Z., and O'Boyle, M. F. (2011). A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 117–126. ACM.

[Grewe et al. 2013] Grewe, D., Wang, Z., and O'Boyle, M. F. P. (2013). Portable mapping of data parallel programs to OpenCL for heterogeneous systems. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 0:1–10.

[Guyon and Elisseeff 2003] Guyon, I. and Elisseeff, A. (2003). An Introduction to Variable and Feature Selection. *The Journal of Machine Learning Research*, 3:1157–1182. Available from: http://dl.acm.org/citation.cfm?id=944919.944968.

[Hall et al. 1996] Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. (1996). Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29:84–89. Available from: http://dx.doi.org/10.1109/2.546613.

[Hall and Martonosi 1998] Hall, M. W. and Martonosi, M. (1998). Adaptive parallelism in compiler-parallelized code. *Concurrency: Practice and Experience*, 10(14):1235–1250. Available from: http://dx.doi.org/10.1002/(SICI)1096-9128(19981210)10:14<1235::AID-CPE3%73>3.0.CO;2-Z.

[Harris et al. 2014] Harris, T., Maas, M., and Marathe, V. J. (2014). Callisto: co-scheduling parallel runtime systems. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, page 24. Available from: http://doi.acm.org/10.1145/2592798.2592807.

[Harris and Singh 2007] Harris, T. and Singh, S. (2007). Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 251–264, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1291151.1291192`.

[Hennessy and Patterson 2003] Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition.

[Hoffmann 2014] Hoffmann, H. (2014). Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *26th Euromicro Conference on Real-Time Systems (ECRTS), 2014*, pages 223–232.

[Hoffmann and Maggio 2014] Hoffmann, H. and Maggio, M. (2014). PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 241–247, Philadelphia, PA. USENIX Association. Available from: `http://blogs.usenix.org/conference/icac14/technical-sessions/presentati%on/hoffman`.

[Hong et al. 2012] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. (2012). Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2150976.2151013`.

[Hormati et al. 2009] Hormati, A. H., Choi, Y., Kudlur, M., Rabbah, R., Mudge, T., and Mahlke, S. (2009). Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, Washington, DC, USA. IEEE Computer Society. Available from: `http://portal.acm.org/citation.cfm?id=1636712.1637760`.

[Huh et al. 2013] Huh, W. T., Liu, N., and Truong, V.-A. (2013). Multiresource Allocation Scheduling in Dynamic Environments. *Manufacturing and Service Operations Management*, 15(2):280–291. Available from: `http://dx.doi.org/10.1287/msom.1120.0415`.

[Hummel et al. 1992] Hummel, S. F., Schonberg, E., and Flynn, L. E. (1992). Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101. Available from: http://doi.acm.org/10.1145/135226.135232.

[Intel] Intel. Intel Xeon Processors. http://www.intel.co.uk/content/www/uk/en/processors/xeon/xeon-processor%-e5-family.html.

[Ioannidis and Dwarkadas 1998] Ioannidis, S. and Dwarkadas, S. (1998). Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 107–122. Springer Berlin Heidelberg. Available from: http://dx.doi.org/10.1007/3-540-49530-4_8.

[Ipek et al. 2008] Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. (2008). Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *SIGARCH Computer Architecture News*, 36(3):39–50. Available from: http://doi.acm.org/10.1145/1394608.1382172.

[Jacobs et al. 1991] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87. Available from: http://dx.doi.org/10.1162/neco.1991.3.1.79.

[Jiménez et al. 2009] Jiménez, V. J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., and Navarro, N. (2009). Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC ’09, pages 19–33, Berlin, Heidelberg. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-540-92990-1_4.

[Jordan and Jacobs 1993] Jordan, M. and Jacobs, R. A. (1993). Hierarchical mixtures of experts and the EM algorithm. In *Neural Networks, 1993. IJCNN ’93-Nagoya. Proceedings of 1993 International Joint Conference on*, volume 2, pages 1339–1344 vol.2.

[Kaleem et al. 2014] Kaleem, R., Barik, R., Shpeisman, T., Lewis, B. T., Hu, C., and Pingali, K. (2014). Adaptive Heterogeneous Scheduling for Integrated GPUs. In

*Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2628071.2628088.

[Karsai et al. 2003] Karsai, G., Ledeczi, A., Sztipanovits, J., Peceli, G., Simon, G., and Kovacshazy, T. (2003). An approach to self-adaptive software based on supervisory control. In *Proceedings of the 2nd international conference on Self-adaptive software: applications*, IWSAS'01.

[Kohavi 1995] Kohavi, R. (1995). A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. Available from: http://dl.acm.org/citation.cfm?id=1643031.1643047.

[Kulkarni et al. 2007] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. (2007). Optimistic Parallelism Requires Abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1250734.1250759.

[Lattimore et al. 2014] Lattimore, T., Crammer, K., and Szepesvári, C. (2014). Optimal Resource Allocation with Semi-Bandit Feedback. *CoRR*, abs/1406.3840. Available from: http://arxiv.org/abs/1406.3840.

[Lattner and Adve 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society. Available from: http://dl.acm.org/citation.cfm?id=977395.977673.

[Lee et al. 2010] Lee, J., Wu, H., Ravichandran, M., and Clark, N. (2010). Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 270–279, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1815961.1815996.

[Leverich and Kozyrakis 2014] Leverich, J. and Kozyrakis, C. (2014). Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2592798.2592821`.

[Lewis and Berg 1998] Lewis, B. and Berg, D. J. (1998). *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Liao and Chapman 2007] Liao, C. and Chapman, B. (2007). Invited paper: A compile-time cost model for openmp. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8.

[Liu and Ulukus 2006] Liu, N. and Ulukus, S. (2006). Optimal Distortion-Power Tradeoffs in Sensor Networks: Gauss-Markov Random Processes. *CoRR*, abs/cs/0604040.

[Long et al. 2007] Long, S., Fursin, G., and Franke, B. (2007). A cost-aware parallel workload allocation approach based on machine learning techniques. In *Proceedings of the 2007 IFIP international conference on Network and parallel computing*, NPC'07, pages 506–515, Berlin, Heidelberg. Springer-Verlag. Available from: `http://portal.acm.org/citation.cfm?id=1789295.1789363`.

[Luo et al. 2009] Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. (2009). Dynamic performance tuning for speculative threads. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 462–473, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1555754.1555812`.

[Maggio et al. 2012] Maggio, M., Hoffmann, H., Papadopoulos, A. V., Panerati, J., Santambrogio, M. D., Agarwal, A., and Leva, A. (2012). Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):36:1–36:32. Available from: `http://doi.acm.org/10.1145/2382570.2382572`.

[Magni et al. 2014] Magni, A., Dubach, C., and O'Boyle, M. (2014). Automatic Optimization of Thread-coarsening for Graphics Processors. In *Proceedings of the 23rd*

*International Conference on Parallel Architectures and Compilation*, PACT ’14, pages 455–466, New York, NY, USA. ACM.

[Mars et al. 2011] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. (2011). Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM.

[Masoudnia and Ebrahimpour 2014] Masoudnia, S. and Ebrahimpour, R. (2014). Mixture of experts: a literature survey. *Artificial Intelligence Review*, 42(2):275–293. Available from: `http://dx.doi.org/10.1007/s10462-012-9338-y`.

[McCann et al. 1993] McCann, C., Vaswani, R., and Zahorjan, J. (1993). A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(2):146–178. Available from: `http://doi.acm.org/10.1145/151244.151246`.

[Merkel and Bellosa 2008] Merkel, A. and Bellosa, F. (2008). Memory-aware Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower’08, pages 1–1, Berkeley, CA, USA. USENIX Association. Available from: `http://dl.acm.org/citation.cfm?id=1855610.1855611`.

[Merkel et al. 2010] Merkel, A., Stoess, J., and Bellosa, F. (2010). Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys ’10, pages 153–166, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1755913.1755930`.

[Miceli et al. 2013] Miceli, R., Civario, G., Sikora, A., César, E., Gerndt, M., Haitof, H., Navarrete, C., Benkner, S., Sandrieser, M., Morin, L., and Bodin, F. (2013). AutoTune: A Plugin-driven Approach to the Automatic Tuning of Parallel Applications. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, PARA’12, pages 328–342, Berlin, Heidelberg. Springer-Verlag. Available from: `http://dx.doi.org/10.1007/978-3-642-36803-5_24`.

[Mitchell 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Moore and Childers 2012] Moore, R. W. and Childers, B. R. (2012). Using Utility Prediction Models to Dynamically Choose Program Thread Counts. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 135–144, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ISPASS.2012.6189220.

[Nathuji et al. 2007] Nathuji, R., Isci, C., and Gorbatov, E. (2007). Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 5–, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ICAC.2007.16.

[Nichols et al. 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Nickolls et al. 2008] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53. Available from: http://doi.acm.org/10.1145/1365490.1365500.

[Novaković et al. 2013] Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R. (2013). Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, Berkeley, CA, USA. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=2535461.2535489.

[Ogilvie et al. 2014] Ogilvie, W., Petoumenos, P., Wang, Z., and Leather, H. (2014). Fast automatic heuristic construction using active learning. *International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2014*.

[OpenMP] OpenMP. OpenMP API for parallel programming, version 2.0. http://openmp.org/wp/.

[Pacheco 1996] Pacheco, P. S. (1996). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Papadimitriou and Tsitsiklis 1987] Papadimitriou, C. and Tsitsiklis, J. N. (1987). The Complexity of Markov Decision Processes. *Mathematics of Operations Research*, 12(3):441–450. Available from: `http://dx.doi.org/10.1287/moor.12.3.441`.

[Parsec] Parsec. 2.1. `http://parsec.cs.princeton.edu/`.

[Penry et al. 2010] Penry, D. A., Richins, D. J., Harris, T. S., Greenland, D., and D., R. K. (2010). Exposing parallelism and locality in a runtime parallel optimization framework. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 117–118, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1787275.1787311`.

[Polychronopoulos and Kuck 1987] Polychronopoulos, C. D. and Kuck, D. J. (1987). Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439. Available from: `http://dx.doi.org/10.1109/TC.1987.5009495`.

[Pusukuri et al. 2014] Pusukuri, K. K., Gupta, R., and Bhuyan, L. N. (2014). Shuffling: A Framework for Lock Contention Aware Thread Scheduling for Multicore Multiprocessor Systems. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 289–300, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2628071.2628074`.

[Puterman 1994] Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

[Quinlan 1986] Quinlan, J. R. (1986). Induction of Decision Trees. *Journal: Machine Learning*, 1(1):81–106. Available from: `http://dx.doi.org/10.1023/A:1022643204877`.

[Quinlan 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Radojković et al. 2012] Radojković, P., Čakarević, V., Moretó, M., Verdú, J., Pajuelo, A., Cazorla, F. J., Nemirovsky, M., and Valero, M. (2012). Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. *SIGARCH Computer Architecture News*, 40(1):235–248. Available from: `http://doi.acm.org/10.1145/2189750.2151002`.

[Raman et al. 2011] Raman, A., Kim, H., Oh, T., Lee, J. W., and August, D. I. (2011). Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 26–37, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1993498.1993502`.

[Raman et al. 2012] Raman, A., Zaks, A., Lee, J. W., and August, D. I. (2012). Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2254064.2254082`.

[Rangan et al. 2011] Rangan, K. K., Powell, M. D., Wei, G.-Y., and Brooks, D. (2011). Achieving uniform performance and maximizing throughput in the presence of heterogeneity. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 3–14. IEEE.

[Reinders 2007] Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

[Ribic and Liu 2014] Ribic, H. and Liu, Y. D. (2014). Energy-efficient Work-stealing Language Runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 513–528, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2541940.2541971`.

[Russell and Norvig 2003] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

[Scherer 2001] Scherer, A. (2001). *Adaptive Parallelism in Distributed Shared Memory Environments*. Available from: `http://books.google.co.uk/books?id=mRydmgEACAAJ`.

[Snavely and Tullsen 2000] Snavely, A. and Tullsen, D. M. (2000). Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 234–244, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/378993.379244`.

[Somu Muthukaruppan et al. 2014] Somu Muthukaruppan, T., Pathania, A., and Mitra, T. (2014). Price Theory Based Power Management for Heterogeneous Multicores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 161–176, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2541940.2541974`.

[SpecOMP] SpecOMP. 3.0 Benchmark suite. `http://www.spec.org/omp/`.

[Sridharan et al. 2013] Sridharan, S., Gupta, G., and Sohi, G. S. (2013). Holistic Runtime Parallelism Management for Time and Energy Efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 337–348, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2464996.2465016`.

[Sridharan et al. 2014] Sridharan, S., Gupta, G., and Sohi, G. S. (2014). Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2594291.2594292`.

[Stone et al. 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering*, 12(3):66–73. Available from: `http://dx.doi.org/10.1109/MCSE.2010.69`.

[Streit et al. 2012] Streit, K., Hammacher, C., Zeller, A., and Hack, S. (2012). Sambamba: A Runtime System for Online Adaptive Parallelization. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 240–243, Berlin, Heidelberg. Springer-Verlag. Available from: `http://dx.doi.org/10.1007/978-3-642-28652-0_13`.

[Suleman et al. 2008] Suleman, M. A., Qureshi, M. K., and Patt, Y. N. (2008). Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 277–286, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1346281.1346317.

[Sussman 1992] Sussman, A. (1992). Model-driven Mapping Onto Distributed Memory Parallel Computers. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 818–829, Los Alamitos, CA, USA. IEEE Computer Society Press. Available from: http://dl.acm.org/citation.cfm?id=147877.148139.

[Sutton and Barto 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press.

[Tang et al. 2011] Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L. (2011). The impact of memory subsystem resource sharing on datacenter applications. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 283–294. IEEE.

[Tang et al. 2013] Tang, L., Mars, J., Wang, W., Dey, T., and Soffa, M. L. (2013). ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '13, pages 89–100, New York, NY, USA. ACM. Acceptance Rate: 23 Available from: http://doi.acm.org/10.1145/2451116.2451126.

[Thies et al. 2002] Thies, W., Karczmarek, M., and Amarasinghe, S. P. (2002). StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=647478.727935.

[Vengerov 2008] Vengerov, D. (2008). A Gradient-based Reinforcement Learning Approach to Dynamic Pricing in Partially-observable Environments. *Future Generation Computer Systems*, 24(7):687–693. Available from: http://dx.doi.org/10.1016/j.future.2008.02.012.

[Vengerov 2009] Vengerov, D. (2009). A reinforcement learning framework for utility-based scheduling in resource-constrained systems. *Future Generation Computer Systems*, 25(7).

[Voss and Eigenmann 2000] Voss, M. and Eigenmann, R. (2000). ADAPT: Automated De-coupled Adaptive Program Transformation. In *International Conference on Parallel Processing (ICPP)*, pages 163–170.

[Wang 2011] Wang, Z. (2011). *Machine Learning Based Mapping of Data and Streaming Parallelism to Multi-cores*. PhD thesis, University of Edinburgh.

[Wang and O'Boyle 2010] Wang, Z. and O'Boyle, M. F. (2010). Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 307–318, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1854273.1854313`.

[Weinberg 2006] Weinberg, J. (2006). *Job Scheduling on Parallel Systems*. PhD thesis, University of California, San Diego.

[Wen et al. 2014] Wen, Y., Wang, Z., and O'Boyle, M. (2014). *Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms*. High Performance Computing (HiPC).

[William et al. 1996] William, B., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. (1996). Parallel Programming with Polaris. *IEEE Computer*, 29:78–82. Available from: `http://dx.doi.org/10.1109/2.546612`.

[Yang et al. 1995] Yang, C.-T., Tseng, S.-S., Chuang, C.-D., and Shih, W.-C. (1995). Using knowledge-based techniques for parallelization on parallelizing compilers. In *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 415–428. Springer Berlin / Heidelberg. 10.1007/BFb0020482. Available from: `http://dx.doi.org/10.1007/BFb0020482`.

[Yang et al. 2013] Yang, H., Breslow, A., Mars, J., and Tang, L. (2013). Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/2485922.2485974`.

[Ye and Li 2010] Ye, X. and Li, P. (2010). On-the-fly runtime adaptation for efficient execution of parallel multi-algorithm circuit simulation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 298 –304.

[Yu et al. 2005] Yu, J., Buyya, R., and Tham, C. K. (2005). Cost-Based Scheduling of Scientific Workflow Application on Utility Grids. In *Proceedings of the First International Conference on e-Science and Grid Computing*.

[Zahedi and Lee 2014] Zahedi, S. M. and Lee, B. C. (2014). REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 145–160, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2541940.2541962.

[Zhai et al. 2013] Zhai, J. T., Bamakhrama, M. A., and Stefanov, T. (2013). Exploiting Just-enough Parallelism when Mapping Streaming Applications in Hard Real-time Systems. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 170:1–170:8, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2463209.2488944.

[Zhang and Voss 2005] Zhang, Y. and Voss, M. (2005). Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 44.2–, Washington, DC, USA. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/IPDPS.2005.386.

[Zhuravlev et al. 2010a] Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010a). Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1736020.1736036.

[Zhuravlev et al. 2010b] Zhuravlev, S., Blagodurov, S., and Fedorova, A. (2010b). AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 249–260, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1854273.1854307.