Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Πρόβλεψη Κλιμάκωσης Παράλληλων Περιοχών σε Πολυπύρηνες Αρχιτεκτονικές

Διπλωματική Εργασία

του

Γεωργίου Χατζόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Νοέμβριος 2012

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Πρόβλεψη Κλιμάκωσης Παράλληλων Περιοχών σε Πολυπύρηνες Αρχιτεκτονικές

Διπλωματική Εργασία

του

Γεωργίου Χατζόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1^{η} Νοεμβρίου, 2012.

....	....	....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Δημήτριος Φωτάκης
Αν. Καθηγητής Ε.Μ.Π.	Επίκ. Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Νοέμβριος 2012

....

Γεώργιος Χατζόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η παρούσα διπλωματική έγκειται στο ευρύτερο πεδίο των Παράλληλων Συστημάτων και εστιάζει στη μελέτη των χαρακτηριστικών των παράλληλων περιοχών και την χρησιμότητά τους σε μια στατική ανάλυση, η οποία σε συνδυασμό με πληροφορία χρόνου εκτέλεσης, μπορεί να προσφέρει μια μικρού κόστους πρόβλεψη της κλιμάκωσης της παράλληλης περιοχής, δεδομένων περισσότερων υπολογιστικών πυρήνων, με σκοπό την αποδοτικότερη χρονοδρομολόγηση των παράλληλων προγραμμάτων, δρομολογώντας παράλληλες περιοχές ξεχωριστά, παρά ολόκληρα παράλληλα προγράμματα. Εστιάζουμε σε προγράμματα που έχουν παραλληλοποιηθεί με την προγραμματιστική διεπαφή OpenMP και σε κανονικές δομές, συγκεκριμένα σε δομές parallel-for.

Αρχικά, υποστηρίζουμε την ιδέα της χρονοδρομολόγησης και κατανομής πόρων σε επίπεδο παράλληλων περιοχών κι όχι σε επίπεδο προγραμμάτων, πετυχαίνοντας έτσι επιτάχυνση (όπου αυτό είναι εφικτό) και κάνοντας καλύτερη χρήση των πόρων. Προς αυτή την κατεύθυνση, παρουσιάζουμε πειραματικά αποτελέσματα που αναδεικνύουν τις διαφορές που μπορούν να έχουν παράλληλες περιοχές του ίδιου προγράμματος.

Κατόπιν, μελετούμε τα χαρακτηριστικά απλών παράλληλων περιοχών αποσκοπώντας στην εύρεση εκείνων που μπορούν να χρησιμοποιηθούν για να προβλέψουμε τη συμπεριφορά της παράλληλης περιοχής και παραθέτουμε τους περιορισμούς μιας τέτοιας ανάλυσης. Παρουσιάζουμε ένα μοντέλο, το οποίο λαμβάνει υπόψιν το λόγο των υπολογισμών προς τις προσβάσεις μνήμης, χρησιμοποιώντας μια στατική ανάλυση του πηγαίου κώδικα του προγράμματος, σε συνδυασμό με πληροφορία κατά το χρόνο εκτέλεσης του προγράμματος.

Επιπλέον, παρουσιάζουμε ένα απλό μοντέλο μνήμης, που χρησιμοποιείται για να εκφράσει την επίδραση που έχει η μνήμη στην κλιμάκωση ενός παράλληλου προγράμματος, το οποίο σε συνδυασμό με την προηγούμενη στατική ανάλυση μπορεί να παράξει μια ελαφριά και συνεπή πρόβλεψη για τη δυνατότητα επιτάχυνσης της παράλληλης περιοχής, ως μια "συμβουλή" προς το χρονοδρομολογητή.

Τέλος, παρουσιάζουμε μια αρχική υλοποίηση που χρησιμοποιεί το μοντέλο αυτό και επιδεικνύουμε την αποτελεσματικότητα της παρούσας ανάλυσης, εφαρμόζοντάς την σε παράλληλες περιοχές από τις σουίτες προγραμμάτων *NAS Parallel Benchmark Suite* (NPB) και *Polyhedral Benchmark Suite* (PolyBench), συγκρίνοντας την προβλεπόμενη συμπεριφορά με πειραματικά αποτελέσματα από έναν 8-πύρηνο κόμβο της οικογενείας Clovertown και έναν 24-πύρηνο κόμβο της οικογενείας Dunnington.

Λέξεις Κλειδιά

OpenMP, Πρόβλεψη Κλιμάκωσης, Στατική Ανάλυση, Πληροφορίας Χρόνου Εκτέλεσης, Χρονοδρομολόγηση παράλληλων Προγραμμάτων

Abstract

This diploma thesis lies in the general area of Parallel Systems and copes with the study of parallel region characteristics and their use in a static analysis, which combined with runtime information can provide a lightweight prediction of the region scalability when the number of available processors increases, with the aim of a more efficient scheduling of parallel applications, by scheduling parallel regions, rather than parallel applications. We mainly focus on programs parallelized using the OpenMP API and regular parallel constructs, namely parallel-for constructs.

Initially, we support the idea of resource allocation and scheduling on a per-parallel-region basis, rather than on a per-program, thus achieving speedup (where possible) and better resource utilization. To that end, we present experimental results which highlight the different behavior that parallel regions of the same program can have.

Subsequently, we study the characteristics of simple parallel regions, opting to find those that can be used to predict the behavior of the parallel region and present the limitations of such an analysis. We also present a model, which takes into account the computation-to-memory ratio, by means of a static analysis of the program source code, as well as runtime information of the program.

Moreover, we present a simple memory model, used to express the effect memory can have to parallel speedup, which along with the aforementioned static analysis, can be used for a lightweight and consistent prediction of the speedup potential of the parallel region, as a "hint" to the scheduler.

Finally, we present a first implementation that uses this model and showcase the effectiveness of this analysis by applying it to parallel regions from the *NAS Parallel Benchmark Suite* (NPB) and *Polyhedral Benchmark Suite* (PolyBench) benchmark suites, comparing the predictions with experimental results from an 8-core Clovertown-based and a 24-core Dunnington-based multicore node.

Keywords

OpenMP, Scalability Prediction, Static Analysis, Runtime Information, Parallel Program Scheduling

Ευχαριστίες

Η διπλωματική αυτή πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη για την καθοδήγησή του κατά την εκπόνηση της παρούσας διπλωματικής εργασίας, καθώς και και για τη συμβολή του στη διαμόρφωση της επιστημονικής προσωπικότητάς μου με τη διδασκαλία του.

Θα ήθελα επίσης να ευχαριστήσω τα μέλη του Εργαστηρίου Υπολογιστικών Συστημάτων και ιδιαίτερα τους Μεταδιδακτορικούς Ερευνητές Γεώργιο Γκούμα, Κορνήλιο Κούρτη και Νίκο Αναστόπουλο για τη καθοδήγησή και τη βοήθειά τους σε κάθε βήμα της εργασίας και για την ενθάρρυνση που μου προσέφεραν τόσο σε επιστημονικό, όσο και σε προσωπικό επίπεδο.

Επίσης, θα ήθελα να ευχαριστήσω όλους τους φίλους και συμφοιτητές μου που με στήριξαν όλα τα χρόνια των σπουδών μου, σε καλές και κακές στιγμές, με την κατανόηση και τη συμπαράστασή τους.

Κυρίως, όμως, θα ήθελα να ευχαριστήσω την οικογένειά μου που βρισκόταν πάντα δίπλα μου σε κάθε επιλογή μου, υποστηρίζοντάς με σε κάθε βήμα της ζωής και των σπουδών μου.

Γεώργιος Χατζόπουλος

Acknowledgements

This diploma thesis is the result of work done at the Computing Systems Laboratory of the School of Electrical and Computer Engineering of the National Technical University of Athens, under the supervision of Associate Professor Nectarios Koziris.

First of all, I would like to thank my supervising professor Nectarios Koziris for his guidance during my work on this thesis, as well as his contribution in shaping my scientific personality with his teaching.

I would also like to thank the members of the Computing Systems Lab and especially the Postdoctoral Researchers Georgios Goumas, Kornilios Kourtis and Nikos Anastopoulos for their guidance and help in every step of this thesis and for their encouragement on a scientific, as well as on a personal level.

Moreover, I would like to thank all my friends and co-students for their support during my studies, being next to me through the good times and the bad, with their understanding and support.

Finally, and mostly, I would like to thank my family for being by my side in every choice, supporting my every step during my life and my studies.

Georgios Chatzopoulos

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In the decade of 1960, Gordon E. Moore published a paper [1] that was meant to change the world of technology. In this paper, he presented an observation on computer hardware history, stating that approximately every two years, the number of transistors on integrated circuits doubles, predicting the continuation of this trend. This prediction has undoubtedly affected the computer hardware industry to this day, sometimes being the "push" behind modern efforts for increased performance. During the first decades of computer history, this increase in transistor density was exploited by an increased clock frequency, leading to a "clock race" between manufacturers. However, this came to an end recently, as heat dissipation and energy consumption became more important and dictated a shift in the industry towards multicore architectures. This shift has unraveled a new world for Computer Science, introducing new runtime environments and programming interfaces, designed to exploit the new hardware.

With almost a decade having passed, multicore systems and parallel applications have become the standard. Operating Systems have evolved to support the new hardware, desktop applications use multiple parallel threads and even traditionally serial algorithms have been replaced by parallel or distributed ones, promising better scaling and increased performance in multicore environments. Another area that also met with renewed interest is resource allocation, as parallel programs tend to have unpredictable behavior regarding their speedup and the extend to which they manage to benefit from more resources. Resource allocation consists of information gathering from the running programs and decision making regarding the resources that are allocated to each one. While the latter one has been studied extensively, with allocation algorithms that meet various goals, such as best throughput or low energy consumption, the first area has only recently started to attract interest, leading to different approaches and interesting results.

In this thesis, we present a model designed to aid the scheduler in deciding the speedup potential of a parallel program, by means of a static analysis of the program source code. By analyzing the source code of a program parallelized using the OpenMP API, we obtain useful information regarding the work assigned to each thread and the computational density of the parallel region, as well as other useful information on memory access patterns and data reuse, which we combine with existing knowledge and an architecture-dependent model of the underlying processor units and memory, leading to a prediction regarding the speedup potential of the program, which can be used as a "hint" to the scheduler, enabling it to make a better decision, according to its scheduling policy.

The rest of this diploma thesis is organized as follows. In Chapter 2 we give a brief overview of memory architectures, the OpenMP API and discuss the traditional views on parallel program scheduling and resource allocation. In Chapter 3 we present the motivation behind this thesis and highlight existing related work on the subject. We showcase the problem of poor scaling for parallel programs, as well as the divergence in performance for parallel regions of the same program and suggest that a different way of scheduling parallel programs should be favored, presenting experimental results to support the aforementioned arguments.

In Chapter 4, which is the main Chapter of the thesis, we present the model used for predicting the scalability of parallel regions. We give detailed description of the static analysis used to extract necessary information from a program, the variables of the model and their meaning and the ways the results can be interpreted, as well as and the trade-offs that naturally occur with every such model.

Chapter 5 is dedicated to the current implementation, with description of the tools used, as well as the extensions and new ones developed for the purposes of testing and benchmarking for this thesis.

In Chapter 6 we present the execution environment used for benchmarking, with a detailed description of the layout of the multicore nodes used, the cache hierarchies and the configurations used. We then apply our model to the *NAS Parallel Benchmark Suite* [2] (NPB) and *Polyhedral Benchmark Suite* [3] (PolyBench) benchmark suites and discuss the results of the application and possible reasons for misspredictions.

Finally, in Chapter 7 we draw some concluding remarks and discuss possibilities of future work.

Chapter 2

Background

2.1 Multiprocessor Architectures

Multiprocessor architectures can be categorized based on their memory organization. They can be placed in three categories, Shared Memory Architectures, Distributed Memory Architectures and Hybrid Architectures.

2.1.1 Shared Memory Architectures

A Shared Memory Architecture is a memory organization scheme that offers a shared memory address space to the programmer. Communication in Shared Memory Architectures is carried out using shared variables in memory, which are accessed and modified using loads and stores. Each processor has its own cache hierarchy. A typical Shared Memory Architecture is shown in figure 2.1. Shared Memory Architectures can provide *Uniform Memory Access* (UMA), where accesses from any processor to any memory address take the same amount of time, or *Non-Uniform Memory Access* (NUMA), where some memory accesses are faster than others, depending on the processor and the topology. It is obvious that NUMA architectures introduce challenges in program development and analysis, but offer very low latencies for nearby memory accesses and lower memory bus congestion when used correctly.

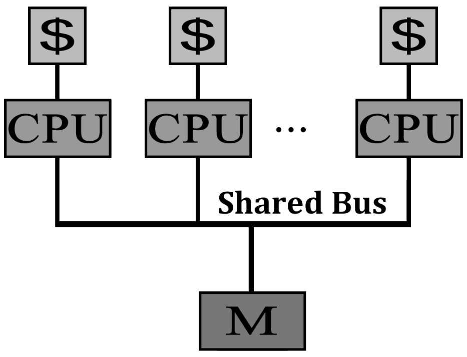

Figure 2.1: Shared Memory Architecture

Shared Memory Architectures offer ease of use to programmers, since parallel programs can operate on the same collections of data, which are present in memory only once. However, such an approach requires a synchronization mechanism, to ensure the validity of data that may be modified by different processors. Such mechanisms are usually the usage of locks, so as to ensure that only one processor modifies a specified memory address at a time. Moreover, although attractive for parallel programming, Shared Memory Architectures can be used for connecting only small numbers of processors, up to a few dozens, since such architectures don't scale well, mainly due to shared bus and memory bandwidth limitations.

2.1.2 Distributed Memory Architectures

A Distributed Memory Architecture is a memory organization scheme that offers no shared memory address and each processor has access to its own private memory address space. Each processor has its own cache hierarchy and processors are connected using an interconnection network, with different implementations varying in characteristics, such as latency and throughput. A typical Distributed Memory Architecture is shown in figure 2.2. Computational tasks can only operate on local data and if remote data is required, the computational task must communicate with one or more remote processors to serve its request. Communication in Distributed Memory Architectures is carried out using explicit send and receive routines to send and receive data.

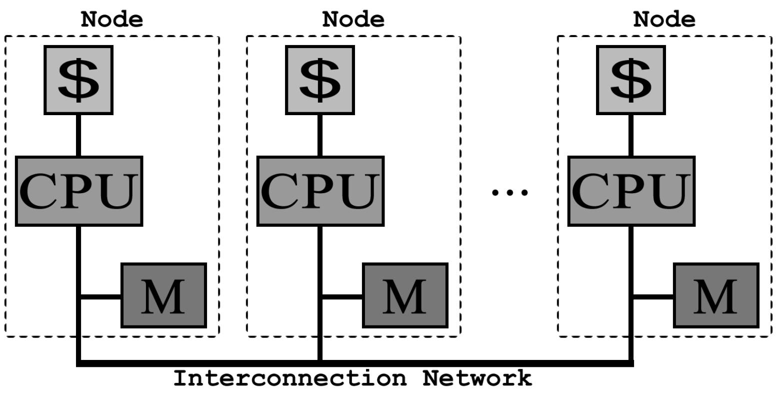

Figure 2.2: Distributed Memory Architecture

Distributed Memory Architectures offer a more challenging programming model for programmers, since every communication and data transfer has to be identified in advance and implemented explicitly and so parallelizing sequential programs using a message-passing model is harder than using a shared memory model. However, Distributed Memory Architectures scale up to thousands of nodes, since they are constructed using independent nodes and interconnection networks, avoiding the bottlenecks that appear in Shared Memory Architectures.

2.1.3 Hybrid Memory Architectures

A Hybrid Memory Architecture is a memory organization scheme that combines Shared Memory Architecture for a small number of processors, that make up an SMP Node, with each node having its own private memory address space and nodes connected using interconnection networks, as in Distributed Memory Architectures. A typical Hybrid Memory Architecture is shown in figure 2.3. This architecture combines the advantages of both memory architectures and is the typical architecture of modern clusters and supercomputers.

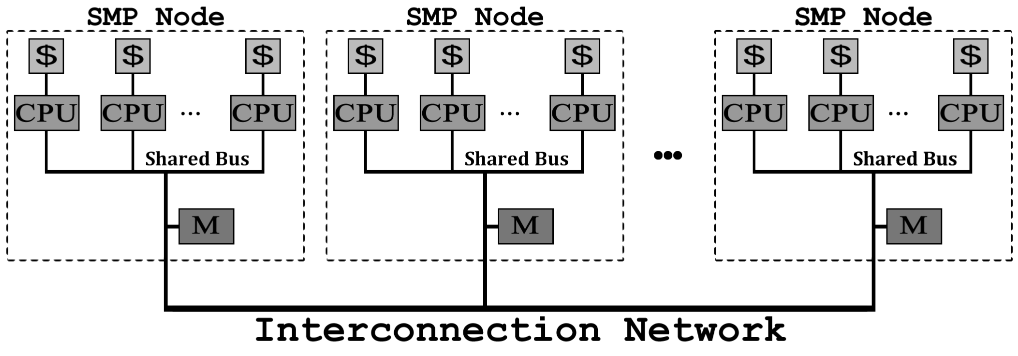

Figure 2.3: Hybrid Memory Architecture

2.2 OpenMP

OpenMP [4] is a specification for shared memory multiprocessing programming. It defines an *Application Programming Interface* (API), with implementations that support multi-platform shared memory multiprocessing programs written in C, C++ and FORTRAN, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior.

2.2.1 Core Elements

The OpenMP implementation in GNU GCC for the C and C++ languages uses the preprocessor directive "*#pragma omp*" to signify OpenMP-specific constructs. The core elements of OpenMP are constructs for thread creation, workload distribution, data management, thread synchronization, user-level runtime routines and environment variables. More specifically:

- **Thread Creation:** To denote a code block that will be executed by multiple threads, the "*#pragma omp parallel*" directive is used. An example of the use of the directive is depicted in listing 2.1. The OpenMP specification uses the fork-join model for thread creation. The initial thread, denoted as *master thread* with thread ID 0 forks a specified number of slave threads, with the runtime allocating threads to different processors and starting the threads. An illustration of the model is depicted in figure 2.4.

- **Workload Distribution:** To denote a code block that will be distributed to threads for execution, the OpenMP API has different directives, to cope with different workloads. For simple parallel loops that will be distributed to threads for execution, with each thread assigned to a collection of iterations, the "*#pragma omp for*" directive is used. An example of the use of the directive is depicted in listing 2.2. To denote that a block of code will be executed by only one thread, the "*#pragma omp single*" directive is used and the OpenMP 3.0 specification includes the "*#pragma omp task*" directive for dynamically changing workloads, in order to address load imbalances.

- **Data Management:** Since OpenMP is a shared memory programming model, variables in OpenMP code are in general visible to all threads by default. To avoid race conditions and pass values between the sequential and parallel regions, *data sharing attributes* are appended to OpenMP directives. The different types of clauses include "*private*", "*firstprivate*", "*lastprivate*", "*shared*" and "*reduction*".

- **Thread Synchronization:** To address the need for synchronization the OpenMP API includes constructs for this purpose, including barriers, critical sections and atomic operations.

- **User-Level Runtime Routines:** User-level runtime routines are user to modify and check the number of threads, detect the execution context, the number of processors, set/unset locks, etc.

- **Environment Variables:** Environment variables are used to configure the execution of OpenMP applications, such as loop iterations scheduling, default number of threads, processor affinities, etc.

Listing 2.1: OpenMP *Parallel* Example

```
void work()
{
  #pragma omp parallel
  {
    printf("Hello, parallel world.\n");
  }
  return 0;
}
```

Listing 2.2: OpenMP *For* Example

```
void work()
{
  int i;
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0;i<MAX_ITER;++i)
      printf("Hello, world. I am here to print: %d\n",i);
  }
  return 0;
}
```

Of course the previous list of core elements of the OpenMP API are far from complete. Listed above are some of the features of the OpenMP API, especially those used in this thesis. For more information, the reader is urged to read the OpenMP Specification [4].

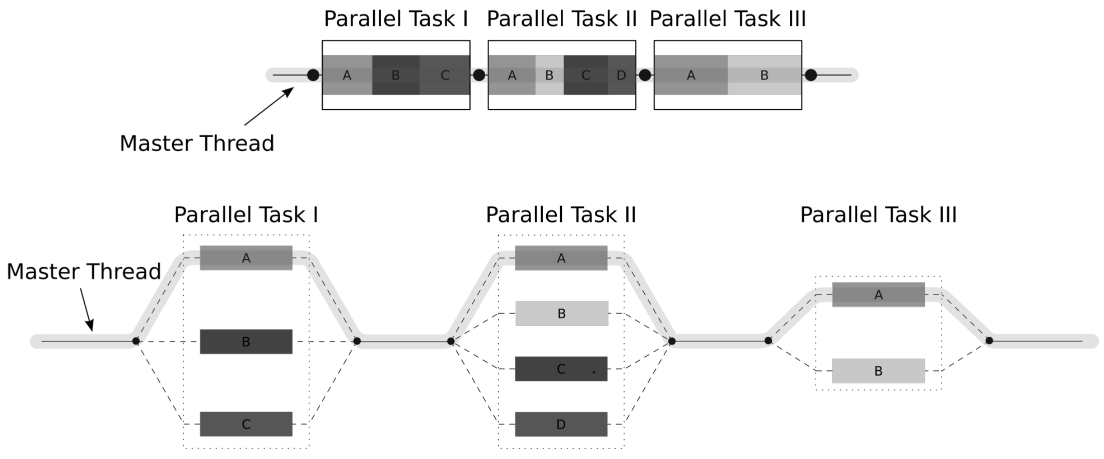

Figure 2.4: Fork-Join Model

2.3 Program Scheduling

Many approaches to program scheduling have been proposed, from older techniques that have been adjusted to modern multicore architectures, to newer ones that have been proposed to specifically address the drawbacks of existing implementations. Some of the most used scheduling approaches are:

- **OS Scheduling:** The first approach is to let the Operating System (OS) handle the scheduling of programs. This approach has the advantage of being easy to implement and use, since scheduling is one of the main responsibilities of operating systems. However, performance is difficult to achieve, due to the generic tuning of operating system schedulers, which are used to schedule applications with different characteristics. Moreover, such a scheduling approach rarely benefits from data locality due to context switching and thread migration and usually leads to resource and energy wasting.

- **Gang Scheduling:** A scheduling approach that is usually favored for parallel programs is Gang Scheduling. The main purpose of gang scheduling is to schedule threads that communicate or share data together, to avoid communication or lock contention delays. Such an approach usually achieves better locality (when used in conjunction with processor affinities, a common practice in gang schedulers), but still don't achieve best resource allocation, since all processors are usually devoted to a process for a time quantum.

- **Space Partitioning:** A third approach, which has been recommended to specifically address the problem of resource utilization, is Space Partitioning, along with co-scheduling of applications [5]. In this approach, applications are scheduled according to resource requirements, with priorities modifying application preference. Using this approach we have the best resource utilization and data locality, since applications are "bound" to a set of processors and even with time-sharing, the number of applications sharing the set of processors is smaller than that in other approaches. Of course this approach can be combined with other techniques, such as using processor affinities to avoid thread migration. The main problem of such an approach is how to identify the needs of an application for resources, a problem that this thesis will address.

Chapter 3

Motivation

From what has been described in Chapter 2.3, the main issue of a Space Partitioning scheduling scheme is the identification of the resource needs of an application, so as not to waste resources in an application that will not use them. To this end, and with a focus on memory and processors as resources, different approaches have been proposed to estimate the speedup potential of a program given a set of resources, including:

- Execution of the program to estimate memory and communication traces, using trace sampling to speedup the process [6].
- Profiling and emulations of the program along with memory performance models [7].
- Static analysis of application binaries, in addition to cache miss counts prediction and memory reuse distance models [8].
- Critical path analysis to estimate speedup for serial programs [9].

However, there are instances where a more lightweight approach is preferred, while sacrificing prediction accuracy. For example, instead of estimating the speedup of a program, we could predict the utilization of resources using a coarse-grain analysis, such as choosing the number of packages that could be utilized. Of course, such an approach would not be suitable to any purposes, but it could for instance apply to the scheduling of scientific applications, which have long execution times and are usually executed on clusters. In this example, we could allocate the predicted number of packages to each application, achieving a balance between performance and energy consumption, as in this case no specific number of processors is required. On the other side, the energy consumption of the system is always an issue.

To identify the cases that would need to be investigated, synthetic benchmarks were implemented and speedup measurements were taken, all of which are described in the following sub-chapter.

3.1 Initial Benchmarks

For the initial benchmarking, the benchmarks synthesized consisted of simple parallel loops, with the structure of listing 3.1. An example of the instruction sequences that were used can be seen in listing 3.2 with the characteristics of the loop that were controlled being:

1. The number of executions of the parallel loop, which determines whether we have reuse of the arrays, regardless of data locality.
2. The number of iterations of the parallel loop, which varies the workload for each thread.
3. The number of arithmetic instructions per memory access, which changes the contention on the memory bus from the threads.
4. The number of instructions in whole, also changing the workload for each thread.
5. The access pattern that was used to access the arrays, exhibiting different data locality and consequently different data reuse potential. The two options included *Spatial Locality* and *No Locality*, accesses used in the latter being random.
6. The size of the working set, as it affects the data reuse potential (in conjunction with the previous) and combined with characteristic (3) can become a limit in performance gains for memory-bound programs.

Listing 3.1: Initial Benchmark Structure

```
[ Headers]

int main (int argc, char **argv) {
  [ Declarations ]
  [ Arguments Parsing ]
  [ Timer Start ]

  #pragma omp parallel [Data Sharing Clauses] {
    #pragma omp for [Data Sharing Clauses]
    {
      [ Unrolled Sequence of Instructions ]
    }

  }
  [ Timer Stop ]
  [ Timer Report ]
  return 0;
}
```

Listing 3.2: Initial Benchmark Instruction Sequence

```
asm (
    "mov (%0), %%rax\n\t"
    "mov %%rax, (%1)\n\t"
    "add %%rax, %%rax\n\t"
    :
    : "r"(a + j),"r"(&j)
    : "rax","memory"
);
```

These benchmarks provided many useful results which are used in the prediction model described in Chapter 4, as well as some insightful findings which are discussed in the following sub-chapters. However, they also indicated that more factors should be taken into consideration and that different program structures should also be examined, so that the results would be more applicable to real-life applications.

3.2 Poor Program Scaling

The first finding of the initial benchmarks is the observation that strong scaling is rarely the case in OpenMP parallel applications, due to various reasons, as can be seen in existing research [10, 11], the main ones being the following:

- Sequential parts of parallel programs, since according to Amdahl's Law [12] the speedup potential of a program is is limited by the the time needed for the sequential fraction of the program, often summarized by the formula

  $$Speedup(N) = \frac{1}{r_s + \frac{r_p}{N}}$$

  where N is the number of available processors, $\mathrm{r_p}$ is the portion of the program that can be parallelized and $\mathrm{r_s}$ the serial portion of the program.

- Bottlenecks on resources that are shared by the processors. Bottlenecks are usually the memory bus for shared memory architectures and the interconnection network for distributed memory architectures. Bottlenecks are caused by the limitations in bandwidth for these shared resources.

- Load imbalances when processing cores are assigned calculations. This can be caused either by the implementation of the program, with parts of the program that are allocated to different processors being imbalanced as in listing 3.3, or by the heterogeneity of the processing cores, in systems that are made up of different processors.

- Synchronization and/or communication overheads, the former being more usual in shared memory architectures, while the latter in distributed memory architectures.

- Overheads concerning thread creation, management and scheduling.

Listing 3.3: Load Imbalance Example

```
void thread_work(int thread_id) {
  int i,j,N = 1000*thread_id;
  for (i=0;i<N;++i)
    for (j=0;j<N;++j)
      doWork(i,j);
}
```

The above factors all contribute, to different extends, to weak parallel program scaling, since according to the roofline model [13], every one of the mentioned factors poses a limit in performance gains. We can understand from the above that programs that do not scale well are very usual, and as such, their identification is important to avoid wasting resources without performance gains.

3.3 Parallel Region Divergence

A second very interesting finding is that the parallel regions of a program may exhibit very different behaviors. Let's consider the listing 3.4. The first parallel region (in this example the combined directive "*#pragma omp parallel for*" is used, which denotes a parallel block of code that is also the workload distribution block) is the initialization of the arrays used, which has a lot of memory transactions and scales poorly, while the second one has a clearly better behavior, due to the operations on the arrays that are accessed.

Listing 3.4: Parallel Region Divergence Example A

```
int doWork() {
  #pragma omp parallel for [Data Sharing Clauses]
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
      [ Array Initialization ]

  #pragma omp parallel for [Data Sharing Clauses]
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
      [ Result Calculation ]
}
```

In this case, what should the scheduler do? Allocate processors so that the second parallel region has performance gains, wasting them on the first one, allocate a minimum number of processors, so that resources are not wasted in the first parallel region, limiting the speedup of the second parallel region, or a solution somewhere in the middle, wasting fewer resources, with a limitation in performance gains? Taking the example a step further, let's assume that at the same time, a second application is also executing, its code being the one shown in listing 3.5. Here we have the opposite situation. The first parallel region scales well, while the second one scales poorly, as it merely copies the data between two arrays. The problem of the scheduling decision exists in this case as well, but now we have another problem as well. We see that in the first part of the programs, example B can utilize more resources, while the opposite is the case in the second part. So in this case, any of the three scheduling decisions mentioned earlier will not be optimal, while the resources may suffice for both programs to have maximum performance gains.

Listing 3.5: Parallel Region Divergence Example B

```
int doWork() {
  #pragma omp parallel for [Data Sharing Clauses]
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
      [ Result Calculation ]

  #pragma omp parallel for [Data Sharing Clauses]
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
      [ Array Copying ]
}
```

It is obvious that the *Space Partitioning* scheme mentioned in Chapter 2.3 should also be enhanced by a dynamic resource allocation mechanism, where a program is not considered a concrete entity, but rather a sequence of regions, each one with different needs for resources and individual scheduling. Of course, such an implementation should take into consideration the work in each region, because this dynamic decision-making introduces an overhead that in some cases could be comparable to the execution time.

Chapter 4

Prediction Model

In this Chapter, we present the Prediction Model that we developed, with detailed description of the method used for the analysis of the code, as well as the memory model that is used.

4.1 Algorithmic and Execution Model

As with any model, our work applies to a specific Algorithmic Model that the programs should implement. More specifically:

- The programs should be parallelized using an implementation of the OpenMP API. For this thesis, the implementation in GCC v4.7.0 was used.
- The current implementation works with programs written in C, but the Prediction Model could also be applied to programs written in FORTRAN.
- The features of the OpenMP API that are supported in the current implementation are a subcategory of the full OpenMP API specification. There is support for *Parallel For* constructs and no nested parallelism, which although may sound restrictive, is an algorithmic model that is widely used and many programs can be altered to have this structure. This is the reason that in the rest of this thesis, the terms ”Parallel Region” and ”Parallel Loop” are used interchangeably.

Also, there is a specific Execution Model that our work applies to, with its most important characteristics being:

- Different levels of Cache are not treated as such and we only consider the largest cache available on the system. Although such an assumption may affect the prediction accuracy, due to the difference between cache-to-cache and cache-to-memory latencies and throughput the accuracy is not affected in a major way, and we have a clearly more compact and easy to configure implementation.
- For the purposes of this thesis, we only tested our model with UMA architectures. This means that changes would have to be made for this model to be applicable to NUMA architectures, as discussed in Chapter 7.2.

Finally, our model assumes a speedup curve that resembles that of figure 4.1 according to which parallel regions scale up to a certain point and then have no further performance gains from more cores, but also their performance does not degrades (at least not significantly) when allocated more cores than they can utilize.

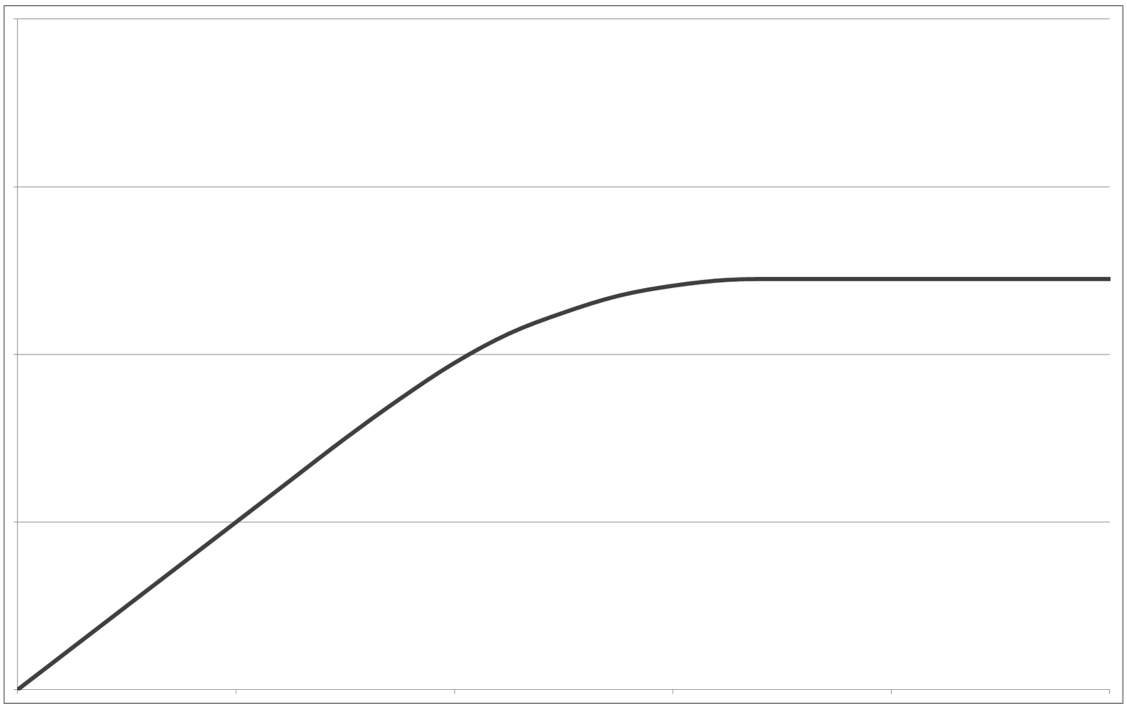
Figure 4.1: Parallel Region Speedup Curve Assumed

4.2 Prediction Model Elements

In this section, the elements that make up our Prediction Model are presented, with examples and experimental results from specific benchmarks, devised to shed light in specific cases, as necessary during the design of the model.

The main factors that contribute to the Prediction Model are the computational density of the parallel region and the application of the memory model, in an attempt to quantify and model existing empirical knowledge, as analyzed in the following subsections.

4.2.1 Computational Density

The term *Computational Density* should not be confused with the term *Operational Intensity*, although both are used to express the same thing: the ratio of operations to the total data accessed. However, by using the term ”Computational Density”, we refer to the ratio of computations to memory accesses of the parallel loop's body. In order to quantify the Computational Density of a parallel loop, we use a scoring system, with two different scorings, one ”good” and one ”bad”. We then use a function to combine the two scores, with possible functions begin the ratio of the two scores, or the relative difference of the two, given by the formula

$$Total\_Score = max(0, \frac{Good\_Score - Bad\_Score}{Good\_Score + Bad\_Score}) \tag{4.1}$$

with the max function used to eliminate negative values. This is what we used for the results of our application of the model, as presented in Chapter 6.4. The "good" and "bad" scores are calculated for every statement of the loop body and then their average values are used in the calculations. This way, every statement has different weight in the average, according to the work it includes.

To the "good" scoring contributes activity that benefits from more processors. Such activity includes of course arithmetic operations, as this activity has performance gains in the form of speedup when more processors are allocated to the parallel region. Moreover, in this type of activity, we include memory accesses that are served by the caches, as more processors means bigger overall cache size (with the assumptions mentioned in Chapter 4.1). Arithmetic operations and cache hits contribute to the positive scoring with different weights, according to the different latencies of different operations, which is expressed by different values for each type of operation, dependent on the architecture the model is used on. The values used do not need to be absolute, as due to the use of ratio, relative values can be used as well.

To the "bad" scoring contributes activity that causes bottlenecks from more processors. In a Shared Memory Architecture, the bottleneck is usually the shared memory bus, so such activity includes memory references that are served by the Main Memory, as more processors means more parallel memory references, leading to memory bus bandwidth saturation and poses a limit to performance gains. There is a distinction here in Read and Write Memory Accesses, as well as a distinction according to the pattern of the memory accesses, with relative values used in this case as well. More on this can be found in Chapter 4.2.2.

The values used for the scorings are of course architecture-dependent, since they rely on instruction latencies, memory architecture and the effect each one has on the scalability of a parallel program. For these values a small benchmark can be used, executed only once for each architecture to extract the values. For the purposes of this thesis, we used existing results from such benchmarks [14] for the two execution environments we used (see Chapter 6.1).

4.2.2 Memory Model

In order to accurately model the effect memory has on the scalability of a parallel region, a series of factors were considered and inserted in the model, all of which are presented in the following sub-chapters.

Memory Accesses

The first decision that was made regarding the Memory Model was which memory accesses affect the execution of the program. For this purpose, we ignore all scalar variables and arrays with smaller number of dimensions than the rest, since their memory footprint is usually an order of magnitude smaller than the rest. Dimensions are not calculated based on the array definition, but rather based on the use of the array in the parallel region, which can be extracted from the indices used, as can be seen from listings 4.1 and 4.2. We see that both reference the same arrays, defined in both cases as 3D arrays. However, in

Example A, all three arrays are referenced using variable indices, while in Example B, two of the three arrays have one of their indices non-changing. So, in Example B, the arrays *matrixA* and *matrixB* are treated as 2D arrays and are ignored as well, since their number of dimensions is smaller that that of *matrixC*.

Listing 4.1: Array Dimension Example A

```c
double matrixA[Ni][Nj][Nk];
double matrixB[Ni][Nj][Nk];
double matrixC[Ni][Nj][Nk];

void doWork(){
  int i,j,k;
  #pragma omp parallel for shared(matrixA,matrixB,matrixC) private(i,j,k)
  for (i=0;i<Ni;++i)
    for (j=0;j<Nj;++j)
      for (k=0;i<Nk;++k)
        matrixC[i][j][k] = matrixC[i][j][k] -
                           (matrixA[i][j][k] * matrixB[i][j][k]);
}
```

Listing 4.2: Array Dimension Example B

```c
double matrixA[Ni][Nj][Nk];
double matrixB[Ni][Nj][Nk];
double matrixC[Ni][Nj][Nk];

void doWork(){
  int i,j,k;
  #pragma omp parallel for shared(matrixA,matrixB,matrixC) private(i,j,k)
  for (i=0;i<Ni;++i)
    for (j=0;j<Nj;++j)
      for (k=0;i<Nk;++k)
        matrixC[i][j][k] = matrixC[i][j][k] -
                           (matrixA[0][j][k] * matrixB[0][j][k]);
}
```

Reuse Distance

In order to be able to categorize memory accesses as ”hits” or ”misses”, we use the reuse distance of an access as follows: by identifying the indices used to reference the array, we consider an access to be a ”miss” if the indices used are the fastest changing ones. In any other case, the access is considered a ”hit”. The terms ”hit” and ”miss” are used to refer to accesses that are served by either the cache or the main memory, contributing to ”good” and ”bad” scoring respectively. To explain this assumption better, let’s consider listing 4.3. If we analyze the accesses to the arrays, we see that accesses to the array *matrixC* are reused Nk times before new data is needed. For accesses to array *matrixA*, we see that each line is reused Nj times before we access the next line. Of course, here we assume that an array line is small enough to fit in any level of cache, which is true in most cases, because if a line of the array is bigger than the larger cache available, then the array would probably not fit in the main memory, in which case performance gains are almost impossible to achieve. So, assuming that a line of the array fits in the cache, we have Nj cache hits for every cache miss, which can safely be considered a ”hit” in our analysis. Finally, considering accesses to *matrixB*, we see that the array is accessed in its whole for every iteration of i, so of course there is some reuse of the data, but since the array size is usually 5-20 times bigger than the cache size, which is a realistic choice, most of the accesses will be cache misses. Being conservative in our analysis, we consider these accesses to be ”misses”, with a possibility of happening, more on which will be discussed in the following section.

Listing 4.3: Array Reuse Distance Example

```
double matrixA[Ni][Nk];
double matrixB[Nj][Nk];
double matrixC[Ni][Nj];

void doWork(){
  int i,j,k;
  #pragma omp parallel for shared(matrixA,matrixB,matrixC) private(i,j,k)
  for (i=0;i<Ni;++i)
    for (j=0;j<Nj;++j)
      for (k=0;i<Nk;++k)
        matrixC[i][j] = (matrixA[i][k] * matrixB[j][k]);
}
```

Working Set Size and Data Reuse

In the previous section, we mentioned that in some cases, misses occur based on the size of the working set of the parallel region, which does not fit in the cache, causing accesses to the main memory. Of course not all accesses are the same, since due to cache policies, there is some data reuse. The effect of this case can be seen in figure 4.2, where we compare the code of listing 4.4 for different total working set sizes. The size of the L3 cache is 16 MB for every 6 cores (see Chapter 6.1). We see that as the working set becomes bigger, the reuse that occurs is less, causing more accesses to the main memory.

To model this reuse, we introduce the possibility of the miss happening, using the formula

$$P = max(0, 1 - \frac{Cache\_Size}{Total\_Working\_Set\_Size}) \tag{4.2}$$

with the max function used to eliminate negative values. The possibility ranges from 0, when the total working set size is smaller than the cache size and all the arrays fit into the cache, to 1, when the total working set size is infinite when compared to the size of the cache. In reality, for total working set sizes that are 10 times bigger than the cache size, the possibility is 0.9 and further increases in the working set size have little effect, as seen in figure 4.2

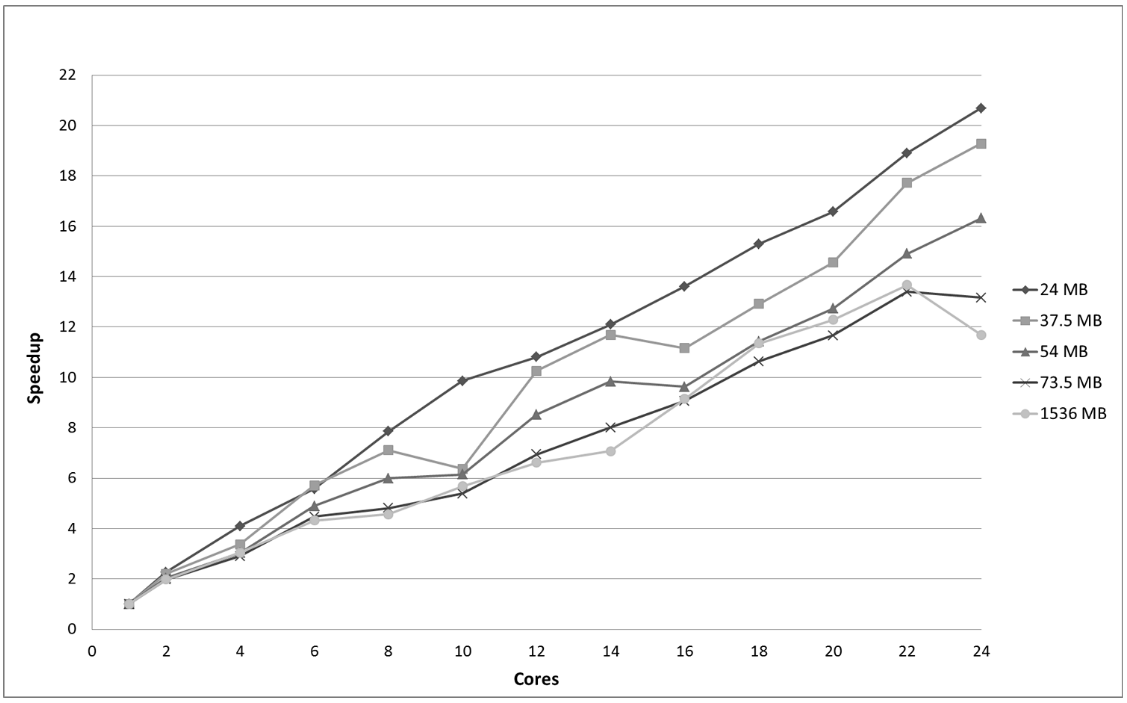

Figure 4.2: Working Set Size Benchmark Graph

Listing 4.4: Working Set Size Benchmark Code

```
double matrixA[Ni][Nj], matrixB[Ni][Nj], matrixC[Ni][Ni], con;

void doWork(){
int i, j, k;

#pragma omp parallel for shared(matrixA,matrixB,matrixC,con)
                                        private(i,j,k) schedule(static)
for (i = 0; i < Ni; i++)
  for (j = 0; j < Ni; j++)
    for (k = 0; k < Nj; k++)
    {
      matrixC[i][j] += con * matrixA[i][k] * matrixB[j][k];
      matrixC[i][j] += con * matrixB[i][k] * matrixA[j][k];
    }
}
```

Access Patterns

The final factor that was considered for the memory model was the access patterns used when accessing the arrays of the parallel region. Let's consider listings 4.5 and 4.6.

Listing 4.5: Access Pattern Benchmark A Code

```
double matrixA[N][N], matrixB[N], matrixC[N];

void doWork(){
int i, j;

#pragma omp parallel for shared(matrixA,matrixB,matrixC)
                                        private(i,j) schedule(static)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    matrixC[i] = matrixC[i] + matrixA[i][j] * matrixB[j];
}
```

Listing 4.6: Access Pattern Benchmark B Code

```
double matrixA[N][N], matrixB[N], matrixC[N];

void doWork(){
int i, j;

#pragma omp parallel for shared(matrixA,matrixB,matrixC)
                                        private(i,j) schedule(static)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    matrixC[i] = matrixC[i] + matrixA[j][i] * matrixB[j];
}
```

We see that they consist of accesses to two vectors and one 2D array. The vectors are used because of their small size (compared to the array size), so they have no practical effect on the results. The size of the working set is ten times the size of the cache and as shown in section *Working Set Size and Data Reuse* further increases of the working set will not make any difference as well. The difference between the two benchmarks is the pattern that is used to access *matrixB*. In the first example, the array is accessed in a streaming way, accessing each row sequentially. In the second example, we access the columns of the array sequentially, leading to memory accesses that take longer to be served by the main memory. This causes the second benchmark to be slower than the first one, due to the latency of its memory accesses. However, this higher latency lead to lower needs in memory bandwidth, giving the parallel region greater potential for speedup. This can be seen in figures 4.3 and 4.4, where we see the difference in execution time, with Benchmark A being clearly faster, while Benchmark B, although not nearly as fast as Benchmark A, scales better.

To further investigate this behavior, we used synthetic benchmarks with the structures in listings 4.7, 4.8, 4.9 and 4.10 that serially access *matrixA* using all the different access patterns possible and compared the times measured to a 1-dimensional sequential access, for equal working set sizes.

Listing 4.7: 2D Access Patterns Comparison - 1D Code Structure

```c
double matrixA[Ni*Nj];

void doWork(){
int i, j;
int count = -1;

for (i = 0; i < Ni; i++)
  for (j = 0; j < Nj; j++) {
    count++;
    matrixA[count] += i+j;
  }
}
```

Listing 4.8: 2D Access Patterns Comparison - 2D Code Structure

```c
double matrixA[Ni][Nj];

void doWork(){
int i, j;
int count = -1;

for (i = 0; i < Ni; i++)
  for (j = 0; j < Nj; j++) {
    count++;
    matrixA[i][j] += i+j;
  }
}
```

Listing 4.9: 3D Access Patterns Comparison - 1D Code Structure

```c
double matrixA[Ni*Nj*Nk];

void doWork(){
int i, j, k;
int count = -1;

for (i = 0; i < Ni; i++)
  for (j = 0; j < Nj; j++)
    for (k = 0; k < Nk; k++) {
      count++;
      matrixA[count] += i+j+k;
    }
}
```

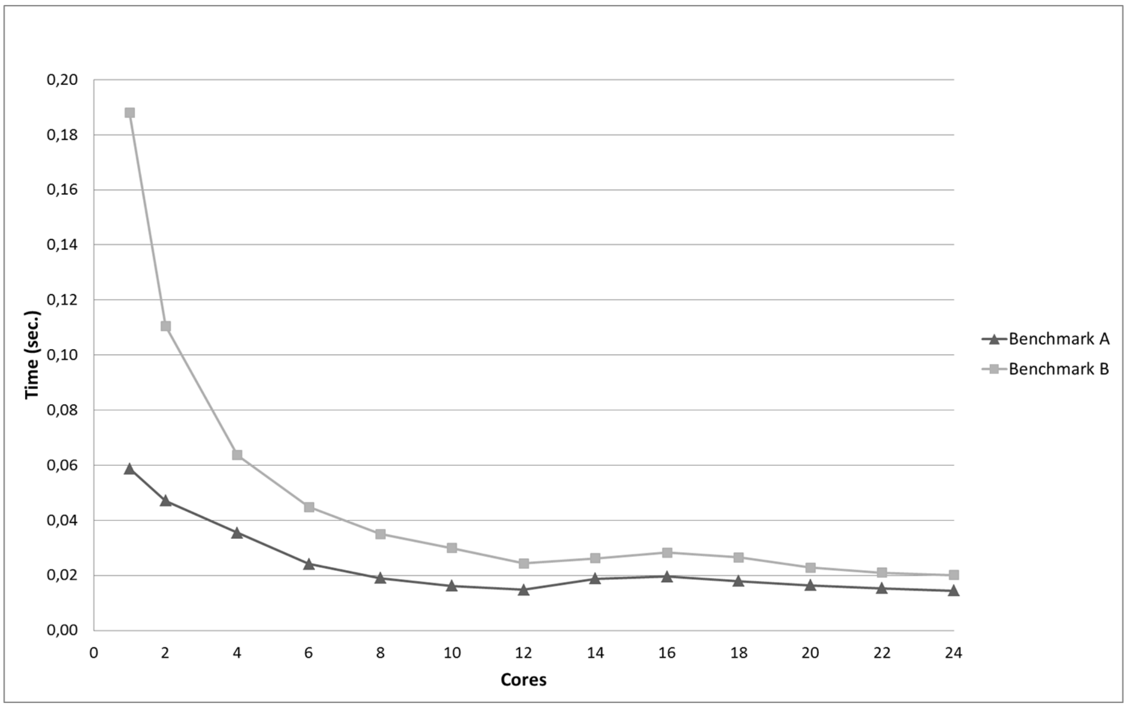

Figure 4.3: Access Pattern Benchmark Time Results

The results for the the 2D and 3D access patterns for working set equal to the size of the largest cache available can be seen in figures 4.5 and 4.6, while the results for a working set 8 times that of the largest cache available can be seen in figures 4.7 and 4.8 (this size was chosen due to the findings of the previous section, which show that larger working sets would make no difference). We see that a streaming access (taking into consideration the way arrays are stored in memory) has the same duration regardless of the dimensions of the array and the working set size, while a different access pattern can have a significant effect on the latency of the memory access, strengthening our argument. This appears to be the case with the benchmark suites used in Chapter 6 as well.

Our model includes knowledge of this behavior, by using modified weights when calculating the "bad" score, based on whether the access is a "streaming" or "latency" access, as discussed above. A "latency" miss has a smaller score, as it contributes less to bottlenecks when more processors are accessing the main memory this way, which is how we defined the "bad" score.

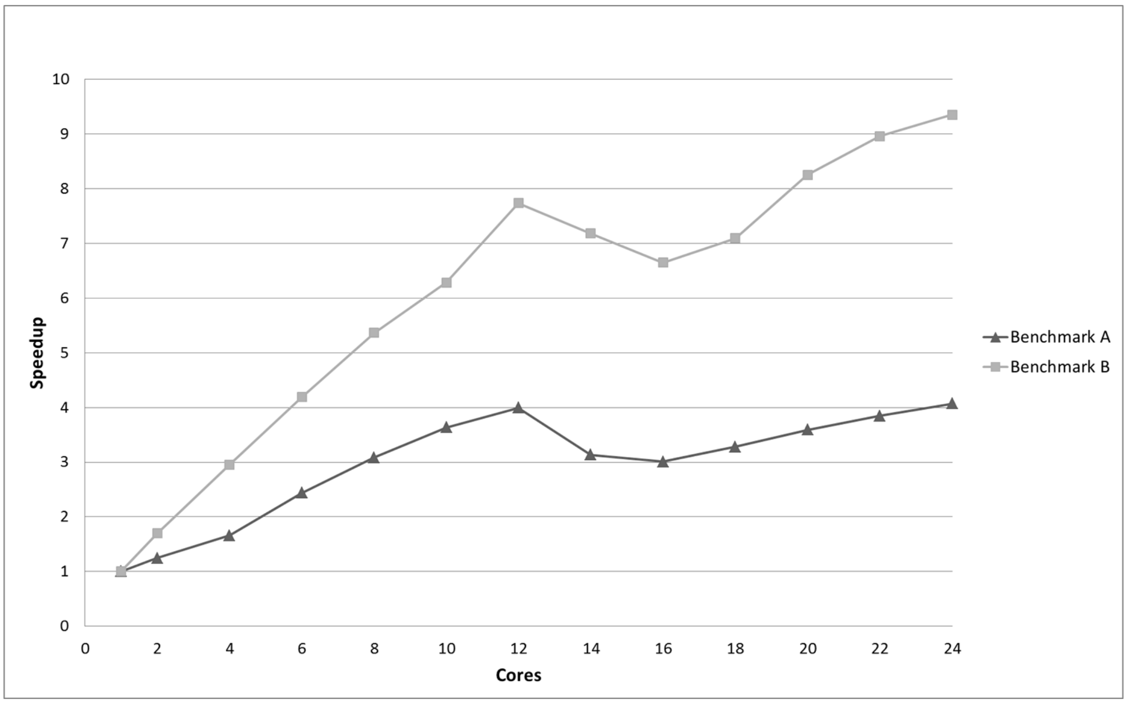

Figure 4.4: Access Pattern Benchmark Speedup Results

Listing 4.10: 3D Access Patterns Comparison - 3D Code Structure

```
double matrixA[Ni][Nj][Nk];

void doWork(){
int i, j, k;
int count = -1;

for (i = 0; i < Ni; i++)
  for (j = 0; j < Nj; j++)
    for (k = 0; k < Nk; k++) {
      count++;
      matrixA[i][j][k] += i+j+k;
    }
}
```

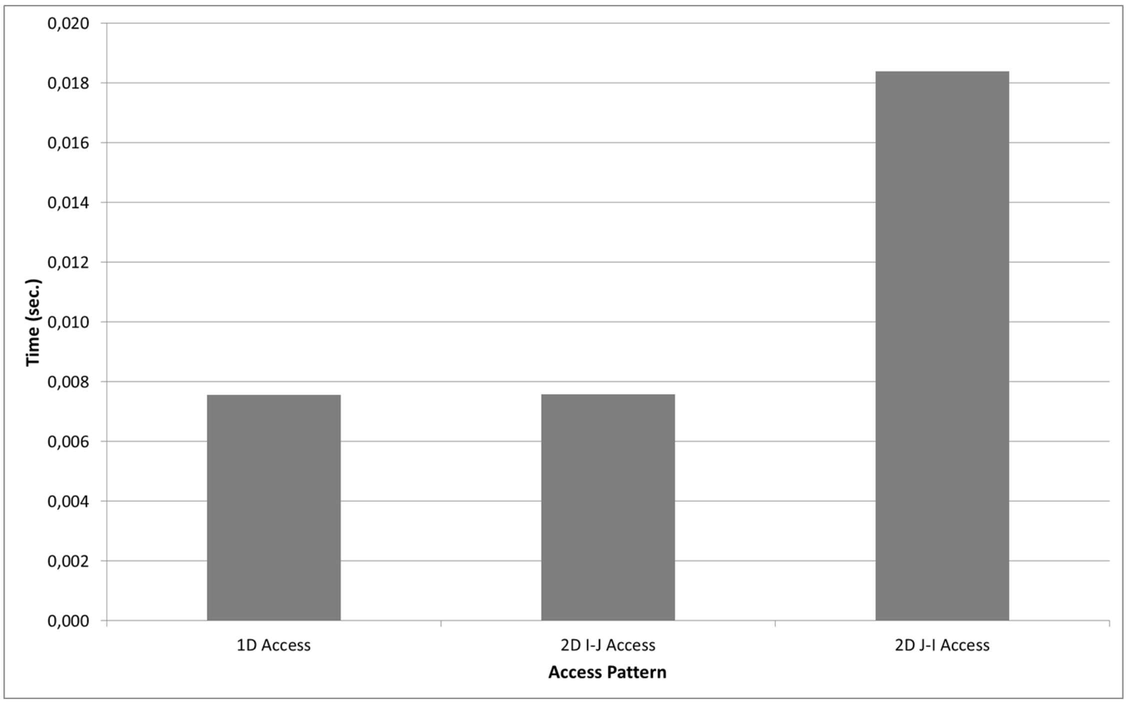

Figure 4.5: 2D Access Patterns Comparison - Small Working Set

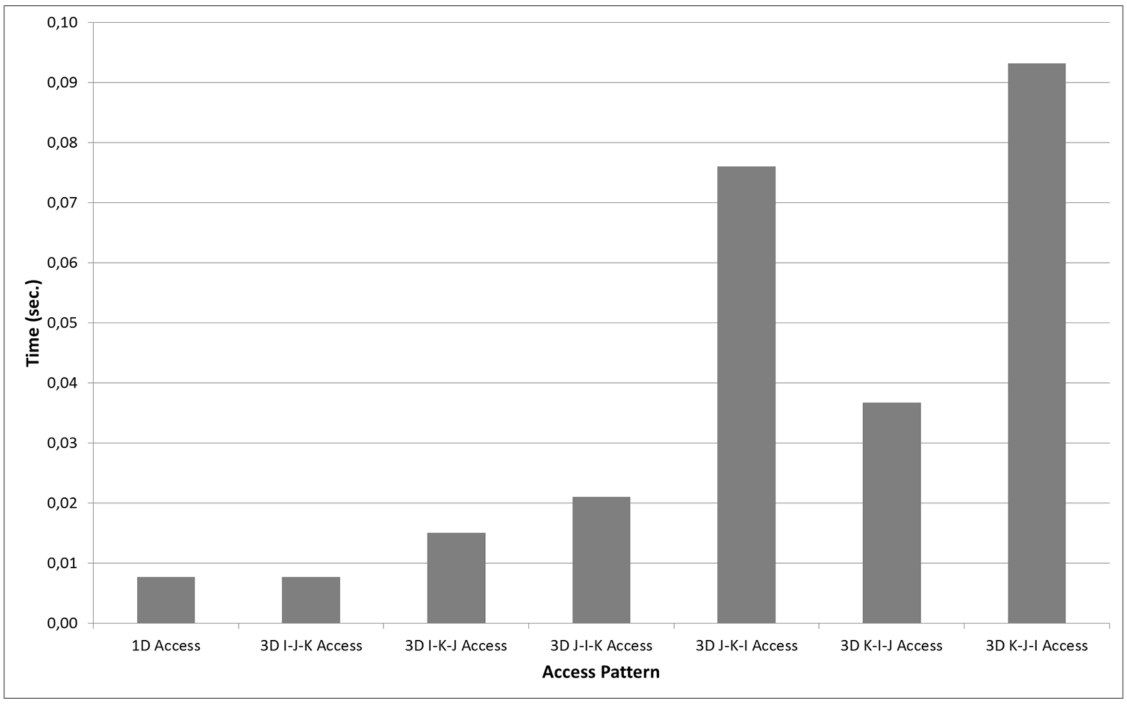

Figure 4.6: 3D Access Patterns Comparison - Small Working Set

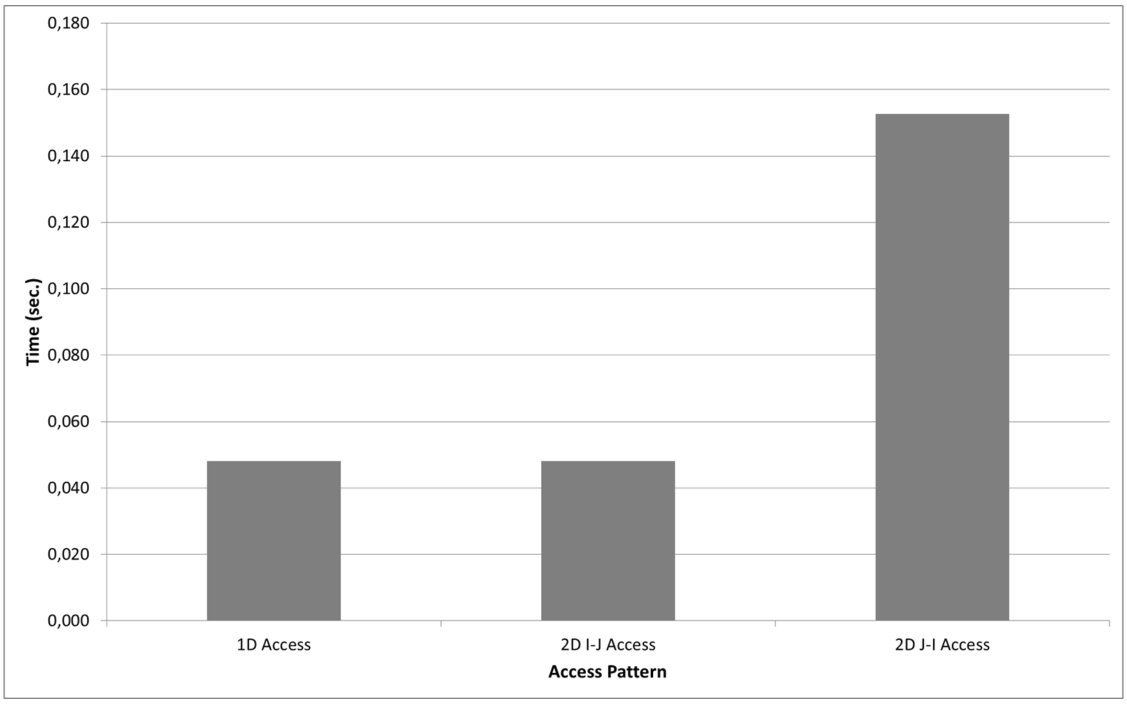

Figure 4.7: 2D Access Patterns Comparison - Large Working Set

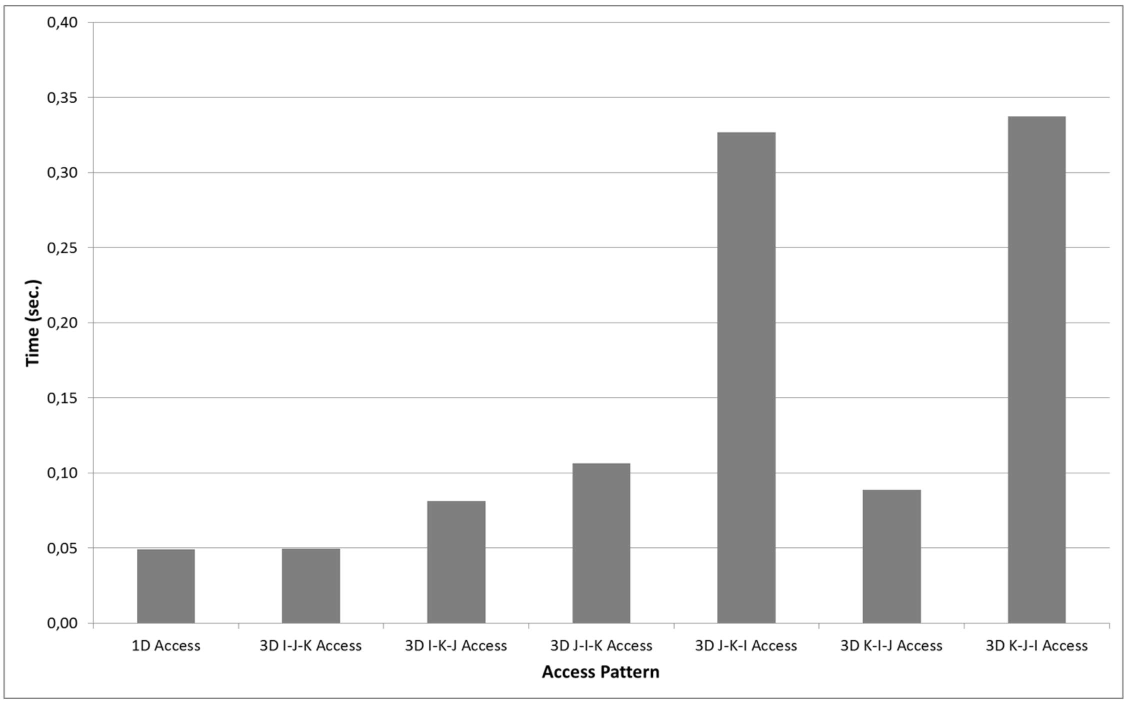

Figure 4.8: 3D Access Patterns Comparison - Large Working Set

Chapter 5

Implementation

In this Chapter, we describe our current implementation, the tools used and their possible extensions. We also discuss some thoughts on the overhead they insert in the execution of programs compiled using it.

5.1 Current Implementation

Our implementation is based on a GCC plug-in, which was extended for the purposes of this thesis. The GCC plug-in is compatible with version 4.6 and greater of the *GNU Compiler Collection* and was originally developed by Dr. Kornilios Kourtis. Its functionality is the following: it inserts one extra pass of the source code during compilation, which includes the following:

1. It builds the graph of the OpenMP constructs, where every OpenMP construct is a node of the graph, with various fields that hold information regarding its type and its relation to other constructs. Nodes are connected using these fields, e.g two consecutive parallel regions are connected via the *next* pointer of the node, while a parallel region and its workload distribution are connected using the *inner* pointer of the parallel region. This process is based on functions that the GCC plug-in development header files provide (*build_omp_regions()*), so new versions of the GCC should not break compatibility with our plug-in.

2. It creates a file with extension ".sc_region_lines" for every source file with parallel regions, inside which it prints the lines at which parallel regions can be found.

3. For every parallel region, if it matches our algorithmic model (parallel regions with a parallel for workload distribution construct inside) it traces the expressions used in the loop initialization, condition and increment statements and tries to find their last values, in the form of an expression tree (which is the way values are stored in the syntactic tree of a program). It then creates a new expression tree that calculates the number of iterations, based on these expression trees. This expression will get its value at runtime, when all the expressions get their respective values.

4. A new function call is created and inserted (using functions that the GCC plug-in development header files provide, such as *gsi_insert_before()*) right before the forking of the master thread of the parallel region (that is a basic block before the one that contains the start of the parallel region), and the expression that represents

the number of iterations is passed as an argument to the call, along with an id of the parallel region, which is the line inside the source code file at which the parallel region starts. This external function prints the parallel region id and the number of iterations, and also starts a high-precision timer, used for the purposes of this thesis. During the execution of a program compiled using this plug-in, a runtime library is necessary, containing the external functions called.

5. A function call is created and inserted (using functions that the GCC plug-in development header files provide, such as *gsi_ insert_ before()*) right before the merging of the threads (that is a basic block before the one that contains the end of the parallel region), which calls a function of the OpenMP runtime to extract the number of threads used in the parallel region. This is necessary as the number of threads can be affected by many factors, such as preprocessor directives, calls to the runtime and environmental variables that are used by the OpenMP runtime and this number is available only inside the parallel region.

6. Finally, a function call is created and inserted (using functions that the GCC plug-in development header files provide, such as *gsi_ insert_ before()*) right after the merging of the threads (in the basic block after the one that contains the end of the parallel region), which in the current implementation is used to print the number of threads used in the parallel region, as well as the time spent in the parallel region, as measured by the high-precision timer used.

Aside from the GCC plug-in, for the purposes of this thesis and the benchmarks executed and timed, other tools were used as well. These include mostly bash scripts, used to parse the info printed by the external functions during the execution of programs, as well as programs such as awk, sed and paste, to export the results in a CSV format, which is ideal for plotting and spreadsheet applications.

5.2 Implementation Overheads

As with any implementation, a concern is the overhead our implementation introduces, both during the compilation of programs and during execution. Of course the execution overhead is more important than the compilation overhead, however both should be considered in every implementation.

For our implementation, the compilation overhead is minimal, since the creation of the graph of OpenMP constructs, which depends on the number and layout of the OpenMP constructs, can be considered as negligible when compared to the compilation process, which also includes the building of the OpenMP regions. The calculation of the iterations of the parallel loop and the static analysis of the code also introduces a small overhead, since we only need to do a single pass of the parallel loop body and finally calculate the possibilities and scorings for the parallel region using known arithmetic expressions. The execution overhead is obviously directly connected to the function calls that are inserted in every parallel region. The current calls introduce an overhead due to the printing functions included in the remote function called. However, these printing functions are not really necessary and are only included for benchmarking and debugging reasons, so omitting them would result to lightweight function calls and a small execution overhead. These assumptions are obviously valid only in UMA architectures, since the overhead of a call in a NUMA architecture is unpredictable due to the nature of the NUMA architectures, however this analysis is beyond the scope of this thesis and is only mentioned as a possible future research topic (see also Chapter 7.2).

Chapter 6

Experimental Evaluation

In this Chapter, we present the execution environment for the benchmarks executed, both for the development of our prediction model and its application to the benchmark suites used. We describe the execution machines and the configuration of the environment. We then apply our prediction model to parallel loops from two different benchmark suites, the *Polyhedral Benchmark Suite* and the *NAS Parallel Benchmark Suite*, and discuss the results of our predictions.

6.1 Execution Environment

The execution environment consists of two different machines used, an 8-core Clovertown-based and a 24-core Dunnington-based multicore node, described in the following sections.

6.1.1 Clovertown

The first node is an 8-core Clovertown-based node with the following characteristics:

- 2 physical packages
- 4 cores per package
- 32 KB L1 cache per core
- 4 MB L2 cache per 2 cores
- 8 GB RAM

as shown in figure 6.1.

6.1.2 Dunnington

The second node is an 24-core Dunnington-based node with the following characteristics:

- 4 physical packages
- 6 cores per package

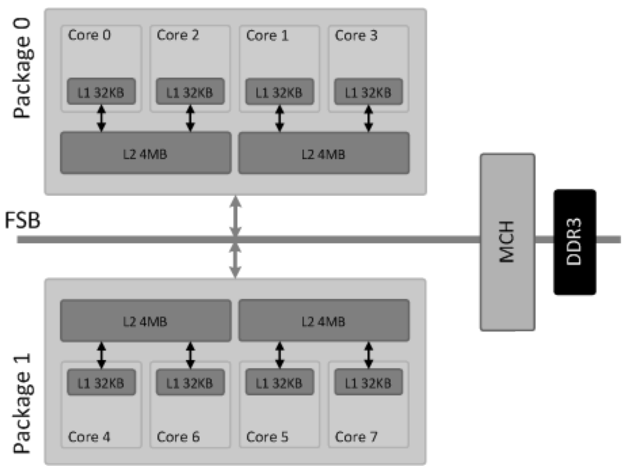

Figure 6.1: Clovertown Layout

- 32 KB L1 cache per core
- 3 MB L2 cache per 2 cores
- 16 MB L3 cache per package
- 8 GB RAM

as shown in figure 6.2.

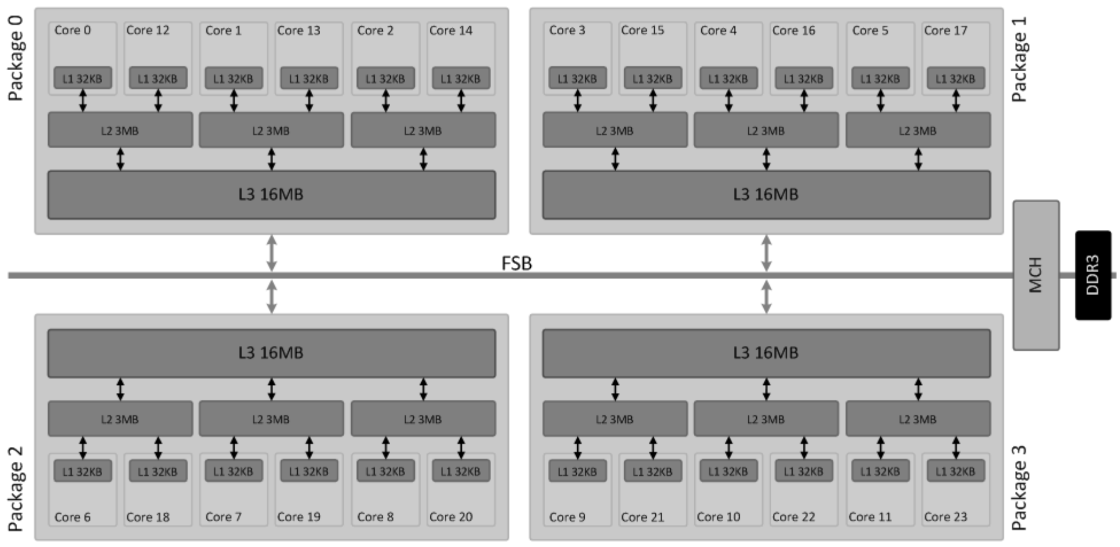

Figure 6.2: Dunnington Layout

6.1.3 Configuration

A specific configuration was used for the OpenMP runtime, using environmental variables and preprocessor directives to control the assignment of threads in available cores, the affinities of threads and the scheduling of parallel regions. More specifically:

- OpenMP threads were assigned to cores in a way that they use different L2 caches when possible (when not all cores are used) while staying in the same package, since we want to apply our model and measure its success in predicting utilization in a per-package level. Also, this way we benefit from cross-processor reuse of data, where this is possible. For this reason, we pinned threads to specific processors, using affinity masks, which for the two different execution environments were:
  - **Clovertown:** For the Clovertown node, the affinities used were *0 1 2 3 4 5 6 7* (see figure 6.1). So when using 2 threads, the are positioned in cores 0 and 1, when using 4 in processors 0, 1, 2 and 3, etc,
  - **Dunnington:** For the Dunnington node, the affinities used were *0 1 2 12 13 14 3 4 5 15 16 17 6 7 8 18 19 20 9 10 11 21 22 23* (see figure 6.2).
- Wherever possible, static scheduling was used, to avoid thread management and scheduling overheads. In any other case, scheduling was set to dynamic.

6.2 Polyhedral Benchmark Suite

The *Polyhedral Benchmark Suite* [3], also referred to as PolyBench, is a collection of benchmarks which include calculation kernels, typically used in past and current publications, which were parallelized using OpenMP for the purposes of this thesis. PolyBench features include:

- Benchmarks in a single file, tunable at compile-time, used for the kernel instrumentation.
- Non-null data initialization, and live-out data dump.
- Syntactic constructs to prevent any dead code elimination on the kernel.
- Parametric loop bounds in the kernels, for general-purpose implementation.
- Clear kernel marking, using #pragma scop and #pragma endscop delimiters.

The PolyBench Suite implementation in C version 3.2 was used, with working sets configured to be ten times the size of the largest cache available on a system. This suite was chosen since the algorithmic model of our work applies to the benchmarks included. Moreover, this benchmark suite provides easy configuration of the kernels, the data types used and the working set sizes.

6.3 NAS Parallel Benchmark Suite

The *NAS Parallel Benchmark Suite* [2], also referred to as NPB, is a small set of programs programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications in the original ”pencil-and-paper” specification (NPB 1). The benchmark suite has been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids. The NPB version 2.3 was used, implemented in C and parallelized using OpenMP, with small modifications to suit the algorithmic model our work applies to. The benchmarks of the suite that were used were:

- BT - Block Tri-diagonal solver
- FT - discrete 3D fast Fourier Transform, all-to-all communication
- LU - Lower-Upper Gauss-Seidel solver
- SP - Scalar Penta-diagonal solver

with the working set size class used being Class C.

6.4 Application Results

This section presents the results of the application of our model and discusses its effectiveness and possible reasons for misspredictions.

First, figure 6.3 shows the results of the PolyBench Suite for the Clovertown node. The X axis is the score of our prediction model, while the Y axis is the utilization of the machine's resources for 8 cores (2 packages). Since we have two possible choices, to allocate either one or two packages, the threshold for this choice is a score of 50%. The dots represent parallel regions that our model was successful in predicting, while the triangles represent missed predictions. The success rate for this suite and execution environment is 78.05%. The trendline should ideally be from (0,0) to (100,100). We see that obviously this is not the case, however it is very close, which is a success regarding our prediction model.

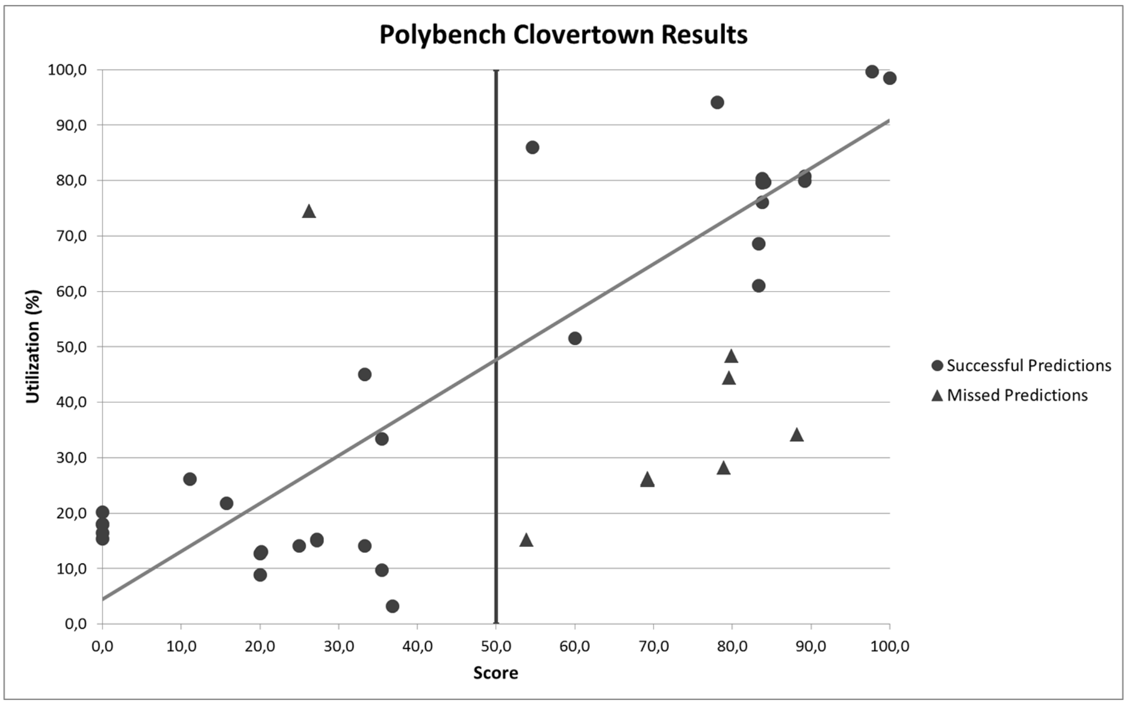

Figure 6.3: PolyBench Clovertown Results - 2 Packages

The second figure, figure 6.4 shows the results of the PolyBench Suite for the Dunnington node, using only two of the four packages (12 out of 24 cores). The X axis is once again the score of our prediction model, while the Y axis is the utilization of the machine's resources for 12 cores (2 packages). Since we have two possible choices, to allocate either one or two packages, the threshold for this choice is a score of 50%. The dots once again represent parallel regions that our model was successful in predicting, while the triangles represent missed predictions. The success rate for this suite and execution environment is 87.8%. We see that the trendline in this case is also close to the ideal, although a little worse than the previous one.

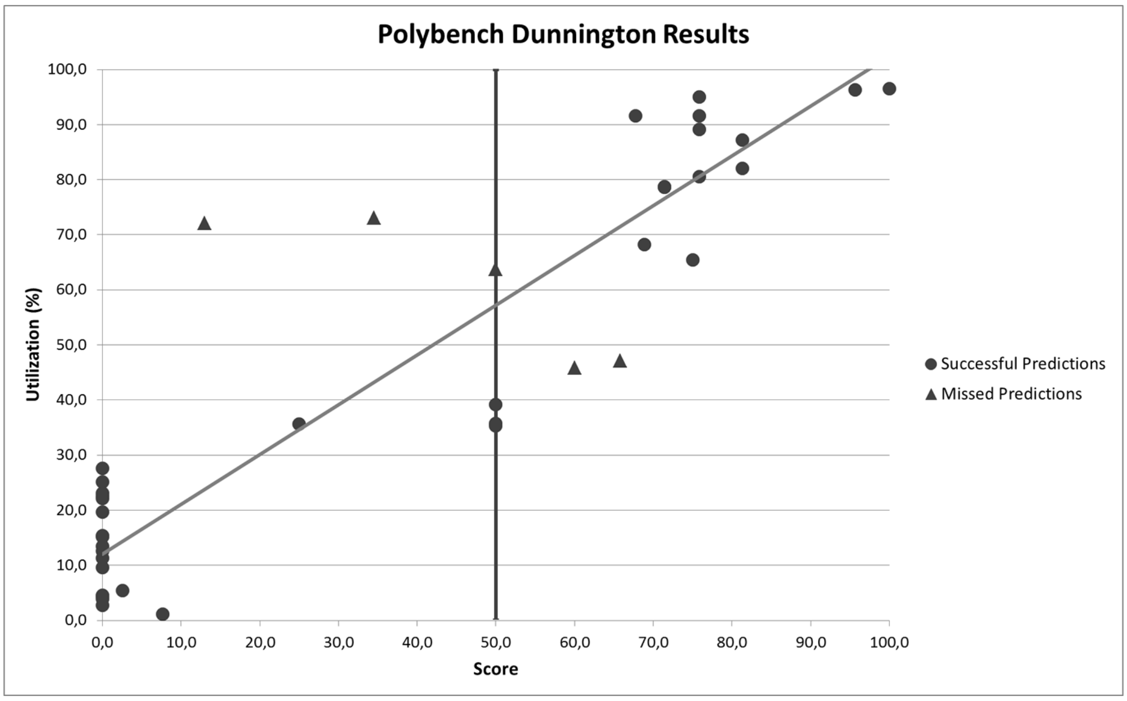

Figure 6.4: PolyBench Dunnington Results - 2 Packages

The third figure, figure 6.5 shows the results of the PolyBench Suite for the Dunnington node, using all four packages (24 cores). The X axis is once again the score of our prediction model, while the Y axis is the utilization of the machine's resources for 24 cores (4 packages). In this case, we have 4 possible choices, to either allocate one, two, three or four packages, so there are 4 areas of decision, with boundaries at 25%, 50% and 75%. The dots once again represent parallel regions that our model was successful in predicting, while the triangles represent missed predictions. The success rate for this suite and execution environment is 78.05%. We see that the trendline in this case is quite similar to that of figure 6.4, and close to the desired one.

Finally, figures 6.6 and 6.7 show results of the NPB Suite for the Dunnington node, using either two or all four packages (12 or 24 cores). The axes are the same as in the previous, as well as the areas of decision. The dots once again represent parallel regions that our model was successful in predicting, while the triangles represent missed predictions. The success rates for this suite are 92.31% for 2 packages and 89.74% for 4 packages. We see that the parallel regions of this implementation of the NPB Suite scale poorly and as such a trendline is not possible.

Regarding the missed predictions, their (inevitable) existence can be attributed mostly to the memory model, since even with the consideration of the reuse distance and the inclusion of the reuse potential for data that has already been accessed, there is always

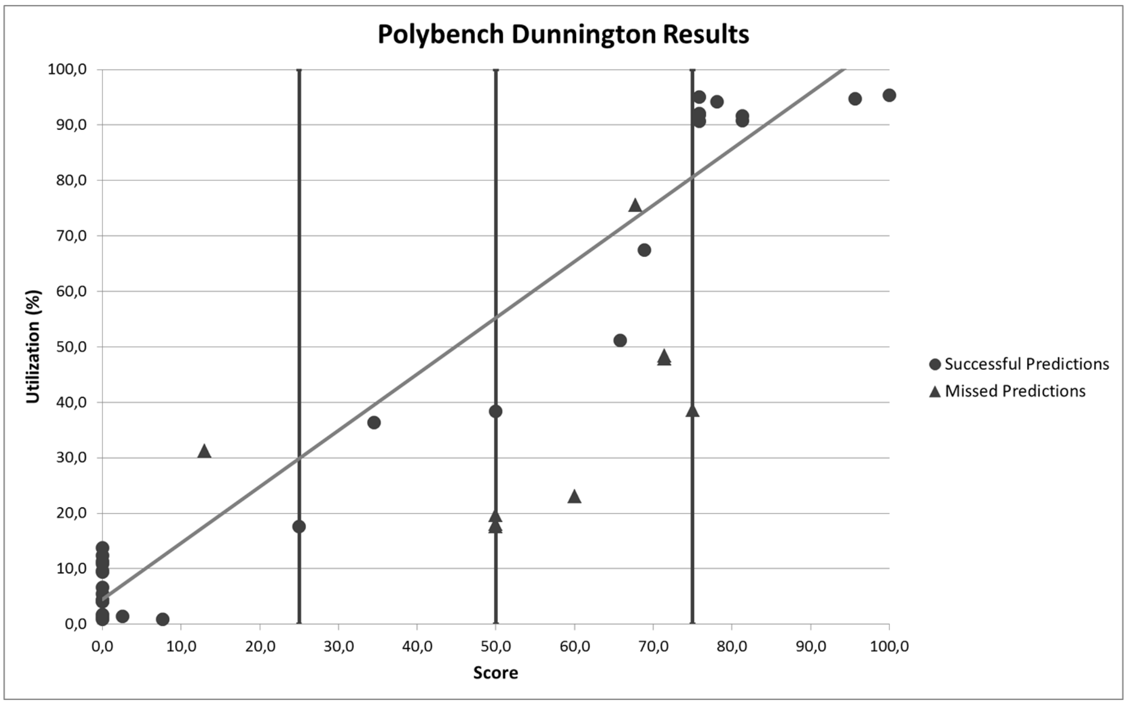

Figure 6.5: PolyBench Dunnington Results - 4 Packages

room for error, since even with the initialization of an array, we have some reuse potential that cannot be measured easily. Moreover, the effect of cache hierarchies is another factor that contributes to these missed predictions. Finally, since we are conservative regarding our prediction, there will be cases where programs exhibit better behavior that what we predict, however cases where we predict better behavior are extremely rare.

Another observation is that usually missed predictions are on the borderlines, either of decision areas, or of utilization, which means that extremes can be easily and quickly (due to the lightweight analysis and implementation) identified. For predictions that are close to the borderlines, we could either handle the predictions with uncertainty, using known analysis such as that of certainty factors, which could adapt over time, or have relaxed limits regarding the scores, which mean that the scheduler could choose (based on its policy) whether to allocate more or less resources to a parallel region that is close to the borderline.

Finally, for the missed predictions for benchmarks of the NPB Suite, the missed predictions can also be attributed to the fact that the parallel region speedup curve is far from the assumed one (as mentioned in Chapter 4.1 and presented in figure 4.1) but rather resembles that of figure 6.8, which causes some missed predictions for our model.

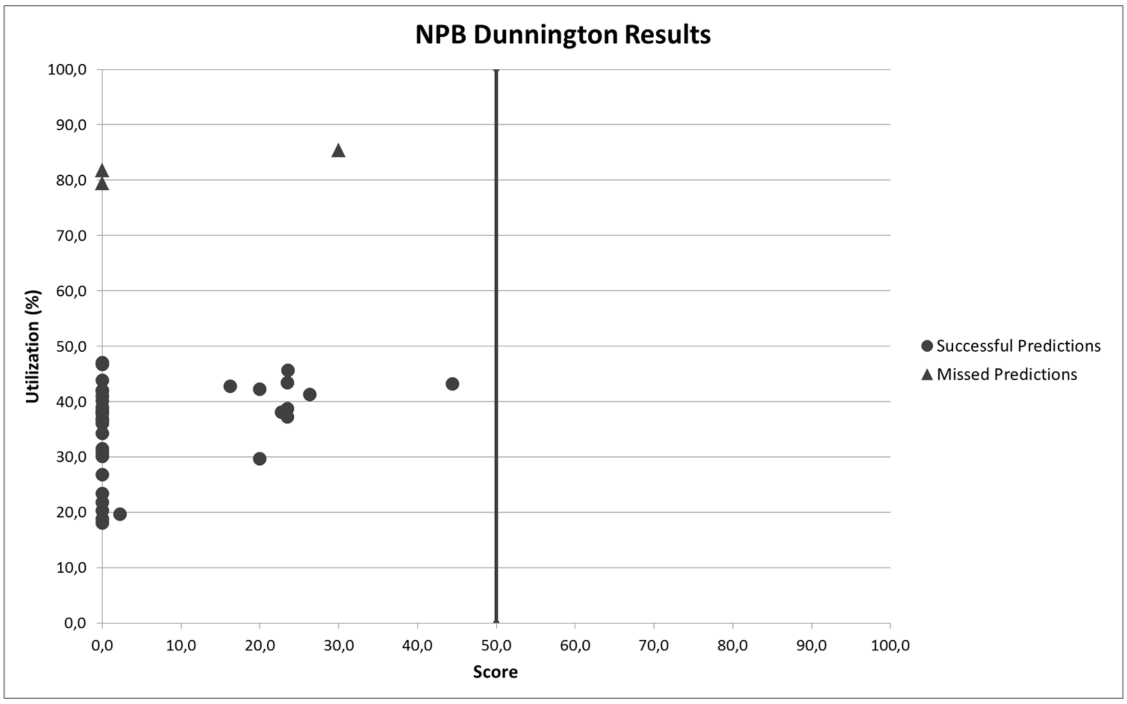

Figure 6.6: NPB Dunnington Results - 2 Packages

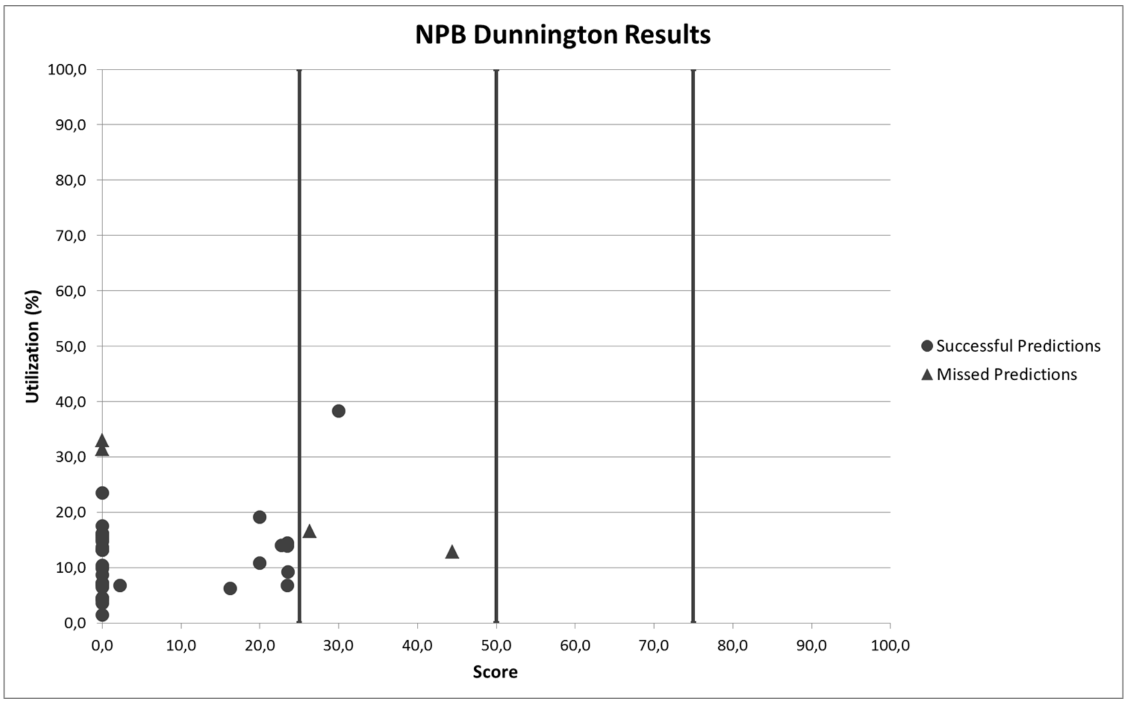

Figure 6.7: NPB Dunnington Results - 4 Packages

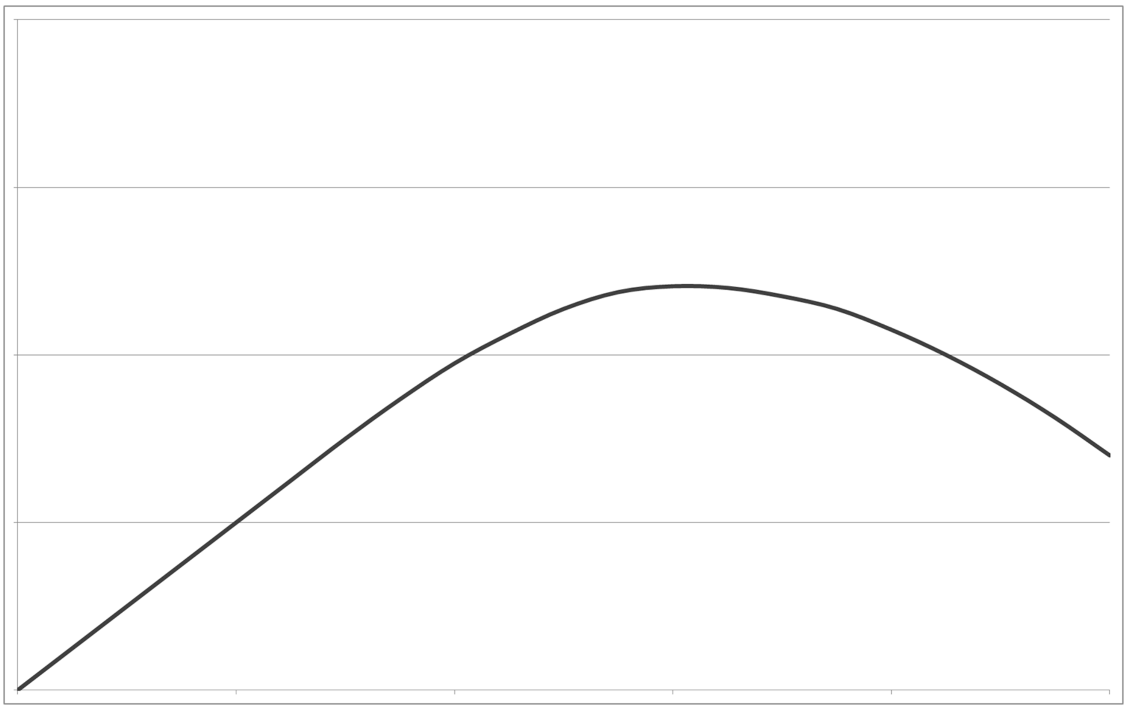
Figure 6.8: Parallel Region Performance Degradation Curve

Chapter 7

Conclusion

7.1 Concluding Remarks

This thesis described the technical and design issues of a lightweight prediction model for parallel OpenMP regions using static analysis of the source code during compilation, combined with runtime information and an architecture-dependent model, used to generate a lightweight predictions of the extent to which the parallel region will utilize resources, on a coarse-grain scale, making the scheduling of parallel programs more efficient and the co-scheduling of applications possible. We also presented a first implementation, which can be easily extended and configured to work with different models and APIs.

After introducing all the necessary background information, we thoroughly presented the prediction model and implementation details and applied it to two different benchmark suites. We successfully predicted more than 78% of the parallel regions for every suite and execution environment and discussed possible explanation for the missed predictions, most of which are in intermediate areas. All in all, we consider our work *successful.* We managed to create a model that applies to a large category of parallel regions, providing a quick and successful prediction of the behavior of the parallel regions.

7.2 Future Work

Although our work was successful in predicting the behavior of parallel regions of the benchmark suites used (see Chapter 6), we consider it to be far from complete. Possible future work includes:

- The extension of the Memory Model to consider Cache Hierarchies.
- The analysis of different access patterns and their effect on parallel region behavior.
- The further extension of the implementation to automate working set analysis and score calculation.
- The application to different
  - Architectures, including different processor and memory architectures (such as NUMA architectures).
  - Benchmarks, to further validate the results of our work.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, 1965.

[2] "Nas parallel benchmarks official site." `http://www.nas.nasa.gov/publications/npb.html`, Sept. 2012.

[3] "The polyhedral benchmark suite official site." `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, Sept. 2012.

[4] O. A. R. Board, "Openmp application program interface." `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[5] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for cmps," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 189–199, ACM, 2010.

[6] L. Carrington, A. Snavely, X. Gao, and N. Wolter, "A performance prediction framework for scientific applications," in *Proceedings of the 2003 international conference on Computational science: PartIII*, ICCS'03, (Berlin, Heidelberg), pp. 926–935, Springer-Verlag, 2003.

[7] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, (Washington, DC, USA), pp. 1318–1329, IEEE Computer Society, 2012.

[8] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, (New York, NY, USA), pp. 2–13, ACM, 2004.

[9] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: parallel speedup estimates for serial programs," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, (New York, NY, USA), pp. 519–536, ACM, 2011.

[10] K. Fürlinger, M. Gerndt, and J. Dongarra, "Scalability analysis of the spec openmp benchmarks on large-scale shared memory multiprocessors," in *Proceedings of the 7th international conference on Computational Science, Part II*, ICCS '07, (Berlin, Heidelberg), pp. 815–822, Springer-Verlag, 2007.

[11] K. Fürlinger and M. Gerndt, “Analyzing overheads and scalability characteristics of openmp applications,” in *Proceedings of the 7th international conference on High performance computing for computational science*, VECPAR’06, (Berlin, Heidelberg), pp. 39–51, Springer-Verlag, 2007.

[12] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[13] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[14] A. Fog, “Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus,” Feb. 2012.