Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Dipartimento di Elettronica e Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

A FRAMEWORK FOR DESIGN-TIME CONFIGURATION OF A RUNTIME MANAGEMENT SYSTEM FOR MANY-CORE ARCHITECTURES

Relatore: Prof. Gianluca Palermo

Tesi di Laurea Specialistica di:
Massimiliano Gentile
Matricola n. 731238

Anno Accademico 2009-2010

To all my dears

We learn wisdom from failure
much more than from success.
We often discover what will do,
by finding out what will not do;
and probably he who never made a
mistake never made a discovery.

Samuel Smiles

Abstract

The *Many-Core Architectures* are the next big turning-point in the development of architectures. We have to be ready to make the most of them by exploiting the advantages that they can offer, but it is not trivial to efficiently utilize the available computing power. The currently used solutions, by themselves, do not scale well with great number of cores, particularly in the presence of concurrent parallel applications.

The goal of this thesis is the development of an efficient *Runtime Management System* that allows to exploit the advantages offered by the *Many-Core Architectures* through the dynamic choice of the parallelization level of the applications. In order to be used with a wide range of applicative scenarios, it has also been implemented a Design-Time Configuration Framework that helps finding the best configuration of the *Runtime Manager* parameters.

Experimental results have been conducted on a real *Many-Core* system, to analyze the difference in system throughput and individual application performance when using the *Runtime Management System*. They show that it is introduced low overhead and that it is obtained a consistent performance improvement in the execution of multiple concurrent running applications.

Ringraziamenti

Prima di tutto vorrei ringraziare chi mi ha permesso di arrivare fin qui, sostenendomi e incoraggiandomi: grazie mamma e papà per avere esaudito ogni mia richiesta, per i riti scaramantici prima degli esami (!) e per il supporto che è stato indispensabile, grazie a Federica per tutto l'amore che mi ha dato in questi anni e per avermi sopportato anche nei momenti peggiori, riuscendo a farmi ritrovare l'allegria, grazie a Federico per aver sempre creduto in me.

Grazie al Professore Gianluca Palermo per la proposta di questa tesi, per la costante disponibilità e per avermi sempre fatto sentire a mio agio.

Grazie anche agli amici emigrati a Milano: Cinzia e Gianluca, Clara, Elio e Giallo per la compagnia, le mangiate, le serate al birrificio e per tutte le esperienze passate insieme. Grazie anche a tutto il gruppo Quinta E, anche se ci vediamo poco, per avermi fatto capire come una amicizia forte può resistere alle lunghe distanze. Grazie a Fausto, per avermi fatto superare in allegria il trauma dei primi mesi a Milano, anche se avrei voluto averlo accanto per molto più tempo.

Infine grazie a Dezo, Manuel, Yan, Ghira, Ghiso e Giorgio, fondamentali per riuscire a sopravvivere ai semestri più noiosi e con i quali ho passato cinque anni fantastici tra Worms Golf, Tokyo, Talking Paper, Poker, Mundial e tutto il resto.

Massimiliano Gentile

Introduzione

Quando i primi sistemi di elaborazione furono sviluppati, erano grandi computer capaci di elaborare piccole quantità di dati attraverso l'esecuzione di una singola applicazione, scritta specificatamente per quel computer. In questa situazione non vi era bisogno di condividere risorse tra diverse applicazioni. Quando il potere di elaborazione dei computer e il numero di applicazioni iniziò ad aumentare, si rese necessario sviluppare sistemi capaci di condividere le risorse disponibili. Inoltre è via via aumentato il livello di complessità delle applicazioni e l'opportunità di eseguire in parallelo alcune porzioni di codice ha portato alla creazione delle applicazioni *multi-task*: alcune elaborazioni sono suddivise in sezioni di codice più piccole ed eseguite parallelamente. In questo modo il numero di processi in esecuzione su un singolo computer è aumentato ancora di più.

Per far fronte alla sempre maggiore richiesta di prestazioni, i grandi produttori di processori hanno, fino a qualche tempo fa, tentato di trovare una soluzione attraverso l'aumento delle frequenze e della complessità delle architetture. Questa scelta ha portato in breve tempo alla nascita di problemi legati all'eccessivo calore e alla dissipazione di potenza. I produttori hanno quindi cercato un rimedio attraverso lo sviluppo di architetture composte da *core* multipli. Oggigiorno un personal computer in commercio è capace di gestire centinaia di processi eseguiti su sistemi che possiedono fino ad otto *core*. Altre architetture utilizzate in campi diversi, come ad esempio i Network Processors o le unità grafiche, presentano un numero di core ancora più elevato. Attualmente la tendenza nello sviluppo di nuovi processori prevede la

produzione di architetture costituite da un numero sempre maggiore di *core*, dalle centinaia alle migliaia di unità, e che prendono il nome di *Architetture Many-Core*.

Sfruttare i vantaggi offerti da questi nuovi prodotti è una grande sfida perché non è banale riuscire ad ottenere un utillizzo efficiente della potenza computazionale che viene fornita. Le applicazioni dovranno essere modificate o, persino, completamente riprogettate per poter essere eseguite su unità di esecuzione multiple. Il lavoro di parallelizzazione delle applicazioni è un compito molto complesso e richiede notevoli sforzi. Fortunatamente sono stati sviluppati strumenti software, come ad esempio OpenMP [1], che aiutano il programmatore semplificando questa operazione e che richiedono l'introduzione di piccole modifiche al codice sorgente per ottenere l'esecuzione parallela di porzioni di codice. È stato recentemente verificato che questa soluzione, da sola, non ottiene miglioramenti delle prestazioni adeguati su processori con un elevato numero di *core*, e in modo particolare quando devono essere eseguite più applicazioni parallele [2].

In questa tesi mi sono concentrato su una categoria particolare di sistemi, detti *Sistemi Embedded Many-Core*. Si tratta di sistemi costituiti da una *Architettura Many-Core*, sviluppati per l'esecuzione di compiti dedicati. Per questo motivo, durante la fase di progettazione del sistema vengono implementati anche i software dei servizi principali. Queste applicazioni devono rispettare specifici requisiti di *Qualità del Servizio* (QoS), ovvero garantire che l'esecuzione dei servizi avvenga senza creazione di disagi all'utente. Questa categoria di sistemi deve essere, inoltre, capace di eseguire applicazioni che non possiedono questo genere di requisiti e che sono aggiunte al sistema dopo la sua commercializzazione.

Principali Contributi di Questa Tesi

L'obiettivo di questa tesi è lo sviluppo di un *Runtime Management System* efficiente, che permetta di sfruttare i vantaggi offerti dalle *Architetture Many-*

Core attraverso la scelta dinamica del livello di parallelizzazione delle applicazioni in ambiente *multi-task*. Per poter essere usato in un ampio insieme di scenari applicativi, è stato inoltre implementato un *Design-Time Configuration Framework*, che aiuta a trovare la migliore configurazione dei parametri del *Runtime Manager*. Per maggiore chiarezza, vengono di seguito forniti maggiori dettagli sui diversi contributi realizzati in questo lavoro.

Il primo contributo è costituito dallo sviluppo di un *Runtime Management System*, capace di gestire l'assegnamento delle risorse, in particolare le unità di esecuzione, alle applicazioni attive e che sia capace di scegliere il loro livello di parallelizzazione sulla base dello stato di utilizzo del sistema. Il *Manager* implementato è stato specificatamente progettato per *Sistemi Embedded Many-Core* e, quindi, è in grado di gestire l'esecuzione, su sistemi con centinaia di *core*, sia dei servizi principali del sistema, che delle applicazioni aggiunte successivamente. Il *Manager* è stato realizzato prestando particolare attenzione all'overhead introdotto. Una caratteristica fondamentale è la possibilità di configurare il *Manager*. Infatti non è possibile progettare un *Runtime Manager* che sia efficiente per qualunque *Sistema Embedded*. Attraverso la modifica di una serie di parametri, il *Manager* può essere adattato e ottimizzato per gestire nel migliore dei modi diversi scenari applicativi.

Trovare l'assegnamento dei parametri che rappresenta la migliore configurazione per un sistema può essere un compito oneroso. Eseguire questa operazione manualmente è impraticabile perché bisognerebbe testare singolarmente un numero molto elevato di configurazioni. Per risolvere questo problema, ho sviluppato un *Design-Time Configuration Framework*. Si tratta di uno strumento che nasce dalla integrazione di due applicazioni: un *Simulatore di Scenari Applicativi* e uno strumento di *Design Space Exploration*. La prima è una applicazione capace di simulare e analizzare il comportamento del *Runtime Management System*, di cui si specifica una configurazione, quando è utilizzato per gestire un particolare scenario applicativo. In altre parole, una volta specificato uno scenario di esecuzione, cioè un insieme di applicazioni di cui si definisce la dinamica e l'intervallo di esecuzione, il *Simu-*

latore valuterà il comportamento del *Runtime Manager* durante la gestione delle applicazioni specificate, per una data configurazione dei parametri. Il secondo strumento, invece, è capace di eseguire il simulatore di scenari applicativi su insiemi di configurazioni del *Manager* e, in seguito, permette di analizzare i risultati delle simulazioni per trovare la soluzione che ottimizza il comportamento del *Runtime Manager* per lo specifico scenario definito.

Inoltre, per permettere la simulazione del *Runtime Management System* e del *Design-Time Configuration Framework* su una *Architettura Many-Core*, ho implementato un *Simulatore di Architetture*, capace di simulare sia l'architettura di un processore con un numero di *core* configurabile, che lo scheduler che assegna i processi alle unità di esecuzione seguendo le direttive del *Runtime Manager*.

Organizzazione

La tesi è organizzata come segue.

Il Capitolo 2 presenta i concetti chiave che sono considerati necessari come conoscenza di base per la comprensione di tutti gli aspetti discussi in questa tesi. Viene descritto lo stato dell'arte per la gestione della esecuzione e le applicazioni autonome, e una descrizione delle tecnologie usate durante lo sviluppo della tesi.

Il Capitolo 3 definisce e descrive il *Runtime Management System*, concentrandosi sulla composizione architetturale del software, sulla interfaccia offerta alle applicazioni e sulle politiche di parallelizzazione adottate.

Il Capitolo 4 illustra la struttura del *Design-Time Configuration Framework*, attraverso la descrizione del *Simulatore di Architetture Many-Core*, del *Simulatore di Scenari Applicativi* e di come si integrano con lo strumento di *Design Space Exploration*.

Il Capitolo 5 mostra i risultati sperimentali delle analisi eseguite sul *Runtime Management System*, che riguardano le politiche di parallelizzazione e i

tempi di esecuzione aggiuntivi introdotti dal *Manager*.

Infine, il Capitolo 6 riassume i principali risultati della tesi e presenta alcuni possibili sviluppi a partire da questo lavoro.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When the first computing systems were deployed, they were very big computers able to process a small amount of data by executing a single application, that was written specifically for that hardware. In this situation there was no need to share resources among multiple applications. As the computational power of computers and the amount of applications started to increase, it was necessary to develop systems capable of sharing the available resources. Moreover, the applications have become more complex and the opportunity to parallelize some portion of them has led to the creation of multi-task applications: some calculations are divided in smaller parts and executed concurrently. In this way the number of processes running in a single computer has increased even more.

To face the increasing request of computational power, the major processor vendors, until recently, have tried to find a solution by increasing the frequency and the circuit complexity. This choice has given rise to problems relative to excessive heating and power consumption. Therefore, the vendors have remedied by starting developing architectures composed of multiple cores. Nowadays, a commercial personal computer is capable of handling hundreds of processes that share up to eight cores. Other architectures, used in different fields, like Network Processors or Graphics Processing Units, are composed of even more cores. The current trend in the design of new proces-

sors is to embed more and more cores. These new architectures are named *Many-Core Architectures* and will be constituted of hundreds of cores.

Exploiting all the advantages offered by these products is a great challenge because it is not trivial to efficiently utilize the available computing power. The applications should be modified, or even completely rewritten, in order to be executed over multiple execution units. The task of the parallelization of the applications is very difficult and requires a lot of efforts. Fortunately, it has been developed tools, like for example OpenMP [1], that help the programmer and simplify this operation. It is required to slightly modify the sources, in order to let the applications execute selected portions of code over all the available cores. However, it has been verified that, using these solutions by themselves, the performance does not scale well with great number of cores, particularly in the presence of concurrent parallel applications [2].

In this thesis I focus on *Many-Core Embedded Systems*. They are systems with an underlying *Many-Core Architecture*, that are designed to perform dedicated functions. Therefore, they are deployed with a set of core services applications, developed in the design phase of the system, that have to respect *Quality of Service* (QoS) requirements. In other words, it should be guaranteed that the execution of those services does not cause annoyances to the user because of quality reductions. These systems can also execute applications that do not have such requirements and that are loaded in the system after the deployment.

1.1 Thesis Contribution

The goal of this thesis is the development of an efficient *Runtime Management System* that allows to exploit the advantages offered by the *Many-Core Architectures* through the dynamic choice of the parallelization level of the applications. In order to be used with a wide range of applicative scenarios, it has also been implemented a *Design-Time Configuration Framework* that helps finding the best configuration of the *Runtime Manager*.

First, it has been designed the *Runtime Management System*, capable of handling the assignment of the resources, in particular the execution units, to the active applications and that is able to choose their parallelization level on the basis of the current status of the system. The implemented runtime manager has been designed for *Many-Core Embedded Systems* and, therefore it is able to handle both core services and not constrained applications running on processors with hundreds of cores. Particular attention has been given to the minimization of the overhead. An essential characteristic is the possibility to configure the manager. Indeed, it is not possible to design a runtime manager that is good for every *Embedded System*. Therefore, through the modification of a set of parameters, it can be adapted and optimized for any applicative scenario.

Finding the assignment of the parameters that represents the best configuration for a system can be a difficult task. There are too many configurations that should be tested and it is not feasible to do this task manually. To solve this problem, as the second step of the thesis, I have developed a *Design-Time Configuration Framework* that automatize the analysis of the space of the possible configurations. It is the integration of two software applications: a newly implemented scenario simulator and an already existent *Design Space Exploration* tool. The first is a tool capable of simulating and evaluating the behavior of the *Runtime Management System* with a specified configuration, when it is used to manage a particular applicative scenario. In other words, it is possible to specify a scenario of execution, by indicating a set of applications with their execution dynamic and the interval of execution, along with the values of the parameters to configure the runtime manager and evaluate how it performs when handling those applications. The second tool is, instead, capable of executing the scenario simulator over a set of configurations of the runtime manager and, then, it allows the analysis of the results to easily find the solution that optimizes the behavior of the *Runtime Manager* for the specific scenario.

In addition, to allow the simulation of the *Runtime Management System*

and the *Design-Time Configuration Framework* over a *Many-Core Architecture*, it has been implemented an architecture simulator, that is able to simulate a processor architecture with a configurable number of cores and a scheduler, that assigns the threads to the execution units following the directions provided by the runtime manager.

1.2 Thesis Organization

The thesis is organized as follows.

Chapter 2 presents the key concepts that are considered necessary as a background for the comprehension of all the aspects of the thesis. It is described the state of the art for the runtime management and the autonomic applications, and a description of the technologies used in the thesis.

Chapter 3 defines and describes the *Runtime Management System*, focusing on the architectural composition, the interface offered to the applications and the adopted parallelization policies.

Chapter 4 illustrates the structure of the *Design-Time Configuration Framework*, with the descriptions of the *Many-core Architecture Simulator*, the scenario simulator and how they integrate themselves with the Design Space Exploration tool.

Chapter 5 shows the experimental results of the analysis conducted over the *Runtime Management System*, more specifically on the parallelization policy and on the overhead.

Finally, Chapter 6 summarizes the main contributions of the thesis and reports some possible future developments, starting from this work.

Chapter 2

Background

This chapter presents the background necessary to understand all the aspects of this thesis. First, it is introduced OpenMP, that will be considered the basis technology for the parallelization of the applications. Then it is outlined the state of the art for the Runtime Management and Autonomic Applications. The first is the field from which this thesis starts, while the second studies the applications that are able to adapt themselves depending on the status of the system. Finally it is presented the *Design Space Exploration* technique, used to optimize parameters at design-time when the solution space is large and complex.

2.1 OpenMP

Open Multi-Processing [1], better known as OpenMP, is an application programming interface for parallelizing programs in a shared-memory environment. OpenMP provides a set of compiler directives, runtime routines, and environment variables that programmers can use to specify shared-memory parallelism in Fortran, C, and C++ programs. It is a widely accepted specification, jointly defined by a group of major computer hardware and software vendors like Intel, AMD, IBM, and Oracle. When OpenMP directives are used in a program, they direct an OpenMP-aware compiler to generate an ex-

ecutable that will run in parallel using multiple threads. Despite the necessity of just little source code modifications, it has been verified that applications that are not memory-intensive exhibit large speedups [3]. OpenMP directives enable you to use an elegant, uniform, and portable interface to parallelize programs on various architectures and systems by creating and managing the threads for you. All you need to do is to insert appropriate directives in the source program, and then compile the program with a compiler that supports OpenMP, specifying the appropriate compiler option. When using compilers that are not OpenMP-aware, the OpenMP directives are silently ignored. The code is automatically parallelized using the fork-join model of parallel execution. At the end of the parallel region, the threads terminate or are put to sleep. The work inside such a region may be executed by all threads, by just one thread, by distributing the iterations of loops to the executing threads, by distributing different sections of code to threads, and any combination of these. OpenMP also allows for nested parallelism, as threads working on a parallel region may encounter another parallel directive and start up new threads to handle it. Nested parallelism can dynamically create and exploit teams of threads and is well-suited to codes that may change their workloads over time.

In order to allow the compiler to parallelize an application it is necessary to insert into the source code the appropriate pragmas. They are defined as compiler directives and specifies how to process the block of code that follows. The most basic pragma is the `#pragma omp parallel` to denote a *parallel region*. An OpenMP program begins as a single thread of execution, called the initial thread. When a thread encounters a parallel construct, it creates a new team of threads composed of itself and zero or more additional threads, and becomes the master of the new team. All members of the new team, including the master, execute the code inside the parallel construct. There is an implicit barrier at the end of the parallel construct. Only the master thread continues executing the user code beyond the end of the parallel construct.

OpenMP supports two basic kinds of work-sharing constructs to specify that the work in a parallel region is to be divided among the threads in the team. The `#pragma omp for` is used for loops, and `#pragma omp sections` is used for sections, that are blocks of code that can be executed in parallel. In the version 3.0 of the specification it has been added a new pragma, `#pragma omp task`, that defines an explicit task. It can be useful for parallelizing irregular algorithms, such as pointer chasing or recursive algorithms, for which other OpenMP work-share constructs are inadequate. Of course, all of these pragmas can be used only in the context of a parallel region.

The pragmas can be followed by one or more so-called clauses, each separated by a comma. They are used to further specify the behavior or to control the parallel execution. One example is the `shared` clause: it is used to specify which data will be shared among the threads executing the region it is associated with. On the contrary, with the `private` clause it is possible to specify that a variable must be used locally by each thread, so that each of them can safely modify the local copy.

The number of threads in the team executing a parallel region can be controlled in several ways. One way is to use the environment variable `OMP_NUM_THREADS`. Another way is to use the `num_threads` clause in conjunction with the parallel pragma. These ways to control the number of threads do not take into account the current occupation of the execution units of the system. If the programmer does not specify a preference for the number of threads, this value is typically set by default equal to the number of the cores of the system.

OpenMP is fully hardware architecture agnostic and if threads and data of an application happen to live apart, the performance will be reduced. Terboven et al. [4] show how to improve the performance of OpenMP applications, by exploiting data and thread affinity, with the implementation of a next touch mechanism for data migration.

2.2 Runtime Management

The management of the execution of applications, especially for embedded system and for multi-core architectures, is a well studied field, for which it is now described the current state of the art.

Illner et al. [5] focus on embedded system and their lack of resources to handle the runtime management. They propose to split the management in two phases, by creating a model at design-time and enforcing the policies at runtime.

Ykman-Couvreur et al. [6, 7] present a runtime manager for multi-processors, capable of supporting profiled applications with different configurations and Quality of Service requirements. The runtime manager choose a system configuration for all the active applications resolving a multi-dimensional multiple-choice knapsack problem (MMKP) with a greedy algorithm, after a design-time reduction of the search space.

Shojaei et al. [8] propose a solution for the same MMKP problem, that is optimized for embedded systems that contain chip-multiprocessors (CMPs).

Mariani et al. [9, 10] introduce a methodology, based on evolutionary heuristics, to identify optimal operating points at design-time with a reduction of the overall design space exploration time. They also introduce a light-weight resource manager that select the optimal parallelism of each application to achieve the Quality of Service constraints.

Corbalán et al. [11] propose an adaptive loop scheduler which selects both thread numbers and scheduling policy for a parallel region through a runtime measurement of the applications.

The Runtime Management of applications parallelized with OpenMP has been examined by different studies. The most known approaches are now presented.

Curtis-Maury et al. [12] find that OpenMP code scales better on CMPs architectures. They also propose an adaptive runtime manager that provides performance improvements for simultaneous multithreaded processors

(SMTs), monitoring the execution time to identify if the use of the second execution context is beneficial for performance.

A series of papers from Broquedis et al. [13, 14, 15] introduce a runtime system that transpose affinities of thread teams into scheduling hints. With the help of the introduced *BubbleSched* platform, they propose scheduling strategy suited to irregular and massive nested parallelism over hierarchical architectures. They also propose a NUMA-aware memory management subsystem to facilitate data affinity exploitation.

Yan et al. [2] implement a scheduler, called SWOMPS, that assigns the threads of all the concurrent applications based on the hardware configuration and on hints of scheduling preference for each application provided by a design-time analysis. The results show that exploiting the affinity of the threads of parallel applications is a valid way to improve performance over many-core architectures.

2.3 Autonomic Applications

Self-tuning, self-aware, or adaptive computing has been proposed as one method to help application programmers confront the growing complexity of multi-core software development. Such systems ease the application burden for the programmer by providing services that automatically customize themselves to meet the needs of the application. The state of the art of this field is now presented.

Blume et al. [16] give a comprehensive review of the state of the art in this area, as well as a good description of the challenges. They state that a combination of static and runtime techniques can improve compilers to the extent that a significant group of scientific programs can be parallelized automatically.

Anane [17] investigates the convergence between QoS management and autonomic computing, highlighting the autonomic ability of QoS mechanism to adapt to prevailing environmental conditions through monitoring

and management mechanism, inherent to autonomic behavior.

Kounev et al. [18] present an approach to autonomic QoS-aware resource management that makes a Grid Computing middleware self-configurable and adaptable to changes in the system environment and workload, to honor the service level agreements.

Chen et al. [19] present an adaptive OpenMP-based mechanism to generate different multi-threaded version of a loop, allowing the application to select at runtime the most suitable version to execute.

Scherer et al. [20] develop a system that allows parallel OpenMP programs to execute on network of workstations (NOW) with a variable number of nodes, thanks to adaptive parallelism techniques that enable the system to distribute the computation between idle nodes.

Wang et al. [21] use a model learned offline to select at runtime the best configuration for parallel programs which are adapted through dynamic compilation.

Hoffmann et al. [22] implement a framework, called Application Heartbeats, which provides a standardized interface that applications can use to indicate their performance and goals. Adaptive system software can use this interface to query the performance and choose better the policies for adaptation.

2.4 Design Space Exploration [1]

The term *Design Space Exploration* has its origins in the context of logic synthesis. Clearly, a circuit can be made faster by spending more parallel gates for a given problem description at the expense of area overhead. By extensively playing around with synthesis constraints, designers have been able to generate a delay-area tradeoff curve in the design space defined by speed and area costs. This process of systematically altering design parameters has been recognized as an exploration of the design space.

[1] For the description of the Design Space Exploration, I refer to [23]

Design Space Exploration tasks today often deal with high-level synthesis problems, such as the automation of resource allocation, binding of computation and communication to resources, and scheduling of operations, for varying design constraints, given a fixed problem description. In order to support early design decisions and due to increasing design complexity, exploration tasks are more and more performed on the system level. The solution space for a system-level design space exploration will quickly become large if arbitrary allocations and mappings are allowed. The complexity increases even further if multiple objectives are subject to the search. In this case, to evaluate a design it is necessary to use the concepts of Pareto dominance and Pareto-optimal solution, that are now introduced.

Definition 1 (Pareto criterion for dominance) *Given k objectives to be minimized without loss of generality and two solutions (designs) A and B with values* $(a_0; a_1; ...; a_{k-1})$ *and* $(b_0; b_1; ...; b_{k-1})$ *for all objectives, respectively, solution A dominates solution B if and only if*

$$\forall_{0 \leq i < k} i : a_i \leq b_i \quad and \; \exists j : a_j < b_j$$

That means, a superior solution is at least better in one objective while being at least the same in all other objectives.

Definition 2 (Pareto-optimal solution) *A solution is called Pareto-optimal if it is not dominated by any other solution. Non-dominated solutions form a Pareto-optimal set in which neither of the solutions is dominated by any other solution in the set.*

An example is visualized in Figure 2.1. The two-dimensional design space is defined by cost and execution time of a design, both to be minimized. Six designs are marked together with the region of the design space that they dominate. Designs 1, 4, 5, and 6 are Pareto-optimal designs, whereas design 2 is dominated by design 4 and design 3 by all other designs, respectively. Without further insights into the design problem, all designs in the set {1,4,5,6} represent reasonable solutions.

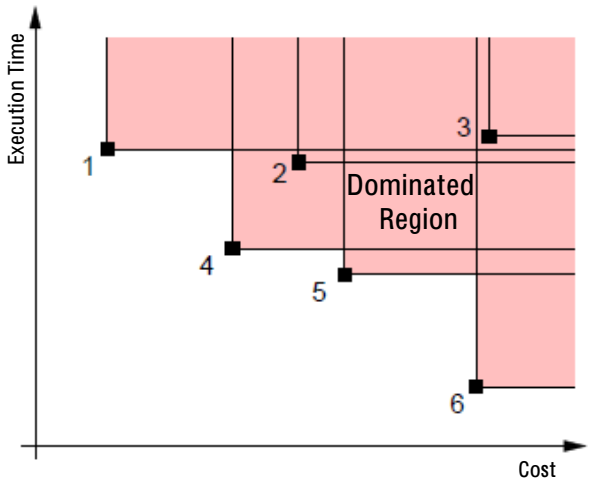

Figure 2.1: Two-dimensional design space with Pareto-optimal designs 1, 4, 5, and 6.

In order to evaluate one design whether it is Pareto-optimal with respect to a set of solutions, all objective values of the design must exhaustively be compared with the corresponding objective values of every other design in the set.

To evaluate a single design point there are different approaches, but the most used is the simulation. It means to execute a model of the system under evaluation, with a defined set of stimuli. Simulations are particularly well suited to investigate dynamic effects in the system. Simulation-based evaluation can only estimate a single stimulus setting at a time, representing one particular implementation of a problem specification. The simulated workload must be chosen by the designer in a way that it represents a variety of typical working scenarios to avoid the optimization of the design for a special case. One drawback of simulations is the need for an executable model. In an early phase of the design providing such a model may impose an unsubstantiated burden for evaluating early design decisions.

There are different strategies for covering the design space. The most used are:

- **Exhaustively evaluating every possible design point**
  This straightforward approach evaluates every possible combination of design parameters and therefore is prohibitive for large design spaces. The design space can be reduced by limiting the range of parameters and/or by parameter quantization.

- **Randomly sampling the design space**
  Evaluating only random samples is the obvious choice for coping with large design spaces. It also has the advantage of revealing an unbiased view of the characteristics of the design space.

- **Incorporating knowledge of the design space**
  Search strategies in this category try to improve the convergence behavior towards optimal solutions by incorporating knowledge of characteristics of the design space into the search process. The knowledge may be updated with every iteration of the search process or may be an inherent characteristic of the search algorithm itself.

To perform a *Design Space Exploration* and to analyze the results, it is necessary to use specific tools, able to simplify and automatize these tasks.

Multicube Explorer [24], shortly M3Explorer, is an interactive program that lets the designer to explore, analyze and optimize a design space of configurations for a parameterized architecture for which a simulator exists. It is an advanced multi-objective optimization framework used to drive the designer towards near-optimal solutions in architectural exploration problems, given multiple constraints. M3Explorer allows a fast optimization of a parameterized system architecture towards a set of objective functions such as energy, delay and area, by interacting with a system-level simulator, through XML files. The final product of the computation is a Pareto curve of configurations within the design evaluation space of the given architecture.

This tool is entirely command-line/script driven and can be re-targeted to any configurable platform by writing a suitable XML design space definition

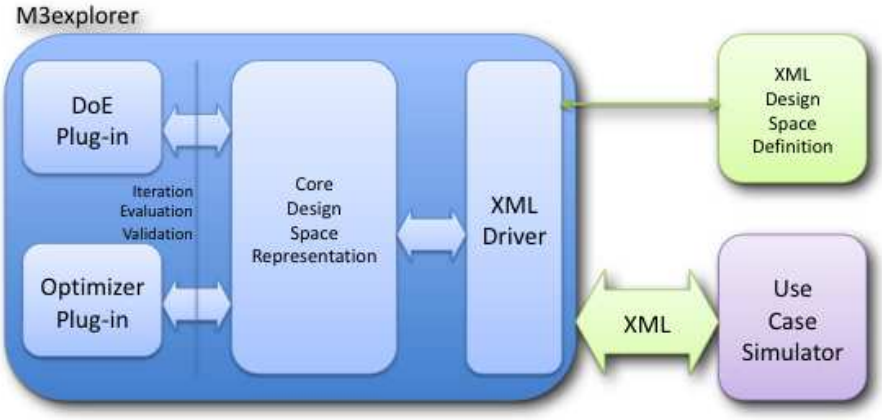

Figure 2.2: A components view of Multicube Explorer.

file and providing a configurable simulator. It supports also the construction of automated exploration strategies which can be implemented by means of command scripts interpreted by the tool without the need of manual intervention. One of the strengths of M3Explorer is the modularity of its components, as it is illustrated in Figure 2.2. The simulator, the optimization algorithms and the other DSE components are dynamically linked at runtime, without the need of recompiling the entire code base. The tool is basically composed by an exploration kernel which orchestrates the functional behavior of the design of experiments and optimization algorithms.

The design space exploration is performed by using the simulation abstraction layer exported by the XML driver to the optimizer plugins. The optimizer instantiates a set of architectural configurations by means of the design space iterators, and passes the corresponding representation to the XML driver which will execute the simulator.

Chapter 3

Runtime Management System

This chapter describes how the Runtime Management System, shortly RMS, is composed. In particular it provides a description of the software architecture, the constituting blocks, the interface offered to the applications and the policies that will be used to improve the resource sharing.

3.1 Overview of the System

The RMS is a system in which the main element is a local runtime manager. A runtime manager is considered local with respect to an application, when the decisions that it makes are related on the information collected analyzing the behavior of the application itself and on some general information on the system provided, for example, by the operating system. This is the main difference between this class of runtime managers and the so-called *System-wide* ones. In this case, the runtime manager is an independent process that collects information on every running process and chooses the resource assignment being aware of the overall status of the active applications. On the contrary, a local runtime manager is typically implemented with a library that is executed inside the context of the application. This is an advantage because, in this way, the overhead due to the invocation of the runtime manager is low.

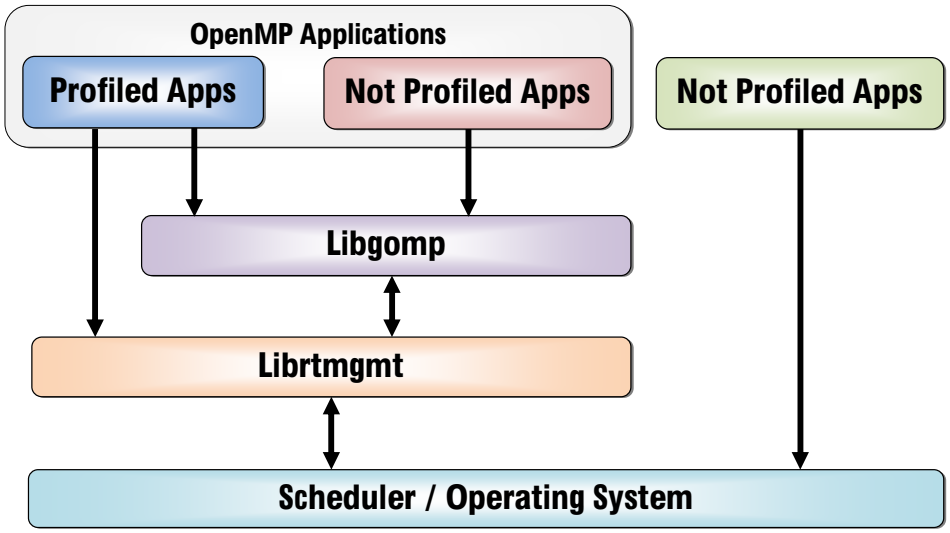

Figure 3.1: The interaction scheme of the Runtime Management System.

In this work the library has been developed with the name `librtmgmt`.

In Figure 3.1 it is possible to see the interaction between the elements of the RMS in the scenario used in this thesis. At the first level there are the applications. They have been classified into three groups:

- **Profiled Applications that use OpenMP:**
  During the design time, these applications are executed with different parallelization levels and some details of the execution, like for example the execution time, are saved in data structures called operating points. These will be used at runtime to identify the optimal number of execution units to use, as described in detail in section 3.7.1. It is possible to distinguish two kind of profiled applications, on the basis of the service class: Guarantee-Throughput and Best-Effort. The first ones are typically core services of a system and are characterized by the presence of Quality of Service (QoS) requirements. The parallel sections are handled by the GNU OpenMP library (*libGOMP*) and should be executed within a completion time to avoid the violation of the requirements. A typical example is a video decoder that should

decode at least a frame every 40 ms to obtain a smooth visualization. The Best-Effort Applications, instead, do not have to respect QoS requirements but should be executed in order to maximize the throughput.

- **Not-Profiled Best-Effort Applications that use OpenMP:**
  These applications can be considered as additional services added to the system after it is released. For this reasons they can't be profiled and, not being core services, they don't have requirements to fulfill but they are executed with a Best-Effort service, to maximize the application throughput. The applications are parallelized following the OpenMP specification. As it is possible to see in Figure 3.1, the not-profiled applications do not interact directly with the runtime management library. Therefore, all the applications of this class do not need any change in the source code to use the functionalities of the runtime manager. It is necessary just a recompilation with the toolchain that was developed during this work and that will be described in the following sections.

- **Not-Profiled Applications that do not use OpenMP:**
  These applications are also included in the proposed scenario, even though they are not controlled by the runtime management library and interact directly with the operating system. These programs do not benefit of the advantages of the runtime manager, but it has been modeled the possibility of the presence of legacy software, or applications that cannot be recompiled.

In Figure 3.1 it is also possible to see that the runtime management library (`librtmgmt`) is placed between the *libGOMP* library and the scheduler/operating system, acting as a mediator between them. The principal interaction of the `librtmgmt` library is to evaluate the correct level of parallelization, by collecting data from the operating system, every time *libGOMP* is going to start a parallel section. The information gathered from the operating system

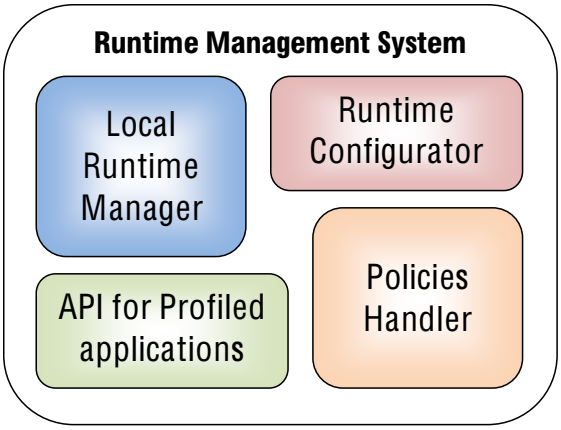

Figure 3.2: The block scheme of the Runtime Management System.

is the status of utilization of the resources, and it represents the view of the local runtime manager over the system and the complete set of active applications, included those that, as it has been described before, do not support the Runtime Management System. The library `librtmgmt` will then spawn the threads providing additional information to the scheduler to allow low level optimizations.

In Figure 3.2 it is shown the composition of the RMS. In addition to the already presented local runtime manager, that will be described further in Section 3.5, there are other three main blocks. The Runtime Configurator is a tool that allows to change the runtime manager configuration without the need of a recompilation. The configurability is necessary because it is not possible to produce a manager that is good for every system. The knowledge of the properties of a system, that can be intrinsic, like the kind of applications that will be executed, or requested, like the minimum throughput of a particular procedure of a core application in the worst case scenario, can be used to tune up the runtime manager and to obtain better results. The configurable parameters of the runtime manager can be changed in two ways: by modifying the default values that are stored in the `librtmgmt.h` header file and then recompiling the library, or at runtime by setting the values in environment variables that have the same name of the parameters. For the

latter case, it is necessary to set the environment variable `RTMGMT_CHANGED` to `1`. This is the way to indicate to the runtime manager that it has to update the value of the parameters from the environment variables.

The API for Profiled Applications are a set of functions that can be used by importing the library header file. They include the methods to load the operating points or to specify a priority for the application. They are described in detail in Section 3.6.

The Policies Handler is a block that manages the different policies for all the classes of the applications that the RMS is able to support. There are different policies for the Profiled Guarantee-Throughput, Profiled Best-Effort and Not-Profiled Applications and in the case of the Not-Profiled ones, there are specialized rules for different classes distinguished by the execution dynamics of the programs. It is possible to find a complete description in Section 3.7.

3.2 Operating points

For the applications that constitute the core set of a system, it is possible to evaluate their execution during the design phase of the system. This technique allows a runtime manager to have information on how a program behaves when it is executed with a particular configuration. The set constituted of a configuration of the system and the related execution information is called operating point. Considering that an application can be composed of various sections with different execution dynamics, an operating point is referred to a section. In the development of the RMS, an operating point is a tuple that contains:

- **Section ID:**
  An identification number for the section, as it will be specified better in Section 3.3.

- **Execution Units:**
  It represents the configuration of the system used during the profiling

of the section, intended as the number of processors used in parallel to execute the code of the program.

- **Execution Time:**
  The elapsed time to execute the section, expressed in milliseconds.

- **Power Consumption:**
  It represents the amount of power dissipated during the execution of the section with the specified configuration. It is expressed in milliwatts. This value is not used by the runtime manager, but it has been inserted to allow a further development in the direction of the minimization of the power consumption.

- **Affinity:**
  In many hardware architectures, but especially in those with Non-Uniform Memory Access (`NUMA`), the assignment of a set of threads, generated to execute a parallel section, to different sets of execution units may affect considerably the performance. Indeed, in a `NUMA` architecture, every cluster of processors has access to a local memory and the access time is different when accessing local and non-local memory. In particular, the parallel sections often work on the same set of data and, therefore, assigning the relative threads to processors of the same cluster can improve the performance. To generalize, it is possible to specify for every operating point the affinity of the threads with an integer number. The affinity can be considered an hint that should be provided to the scheduler, to improve the choice of the assignment of the threads. An affinity of zero means that the threads are not affine and that can be assigned to any processor. If the value is greater than zero, the threads generated for the particular section are affine and should be put in the same cluster or as close as possible, in order to exploit the advantages of the locality of the memory.

3.3 OpenMP Extensions

The OpenMP specification 3.0 has been extended with two new clauses, in order to allow a programmer to specify information on parallel sections in the applications that will be profiled. The first clause is named `section_id` and can be used in the `parallel`, `for`, `sections` and `task` directives. Through this clause it is possible to number a section. This is a way to identify and distinguish the different sections, and it allows the assignment of operating points to sections instead of to an entire application. Therefore, the information provided by the operating point as, for example, the execution time or the power consumption, is referred just to portions of code. An application is usually composed of many blocks of code with different characteristics. Operating points which define the execution time of an entire application, or even of a set of heterogeneous blocks of code, do not allow to find a good policy with a runtime manager because the information provided is too much coarse-grained. The `section_id` must be specified with a positive integer. If the clause is not specified, the compiler will assign a unique identification number, so the sections can be identified as different, during the execution. This feature is particularly useful for not-profiled applications, because the sections can be distinguished by the runtime manager without the modification of the source code.

The second clause added to the original OpenMP specification is named `completion_time`. It is used in the same directives of the `section_id` clause. It allows to indicate the maximum amount of time in which the programmer expects the execution of a section to be completed, in order to guarantee the Quality of Service (QoS). In other words, it is a way to define an execution deadline. The applications must be able to tolerate the situation in which the deadline is not respected, by causing only a decrease of the QoS. The `completion_time` must be specified in milliseconds and must be a positive constant or a variable. If the variable used in the clause assumes a negative value, the runtime management system will halt, so it is necessary to be careful when using a variable instead of a constant.

3.4 Scheduler and Operating System Role

The scheduler and the operating system have a very important role on the functioning of the runtime manager. The scheduler is the executor of the indications provided by the manager and must be able to handle the different kind of applications and to perform low level optimization when assigning the resources. Also, the locality of the manager makes necessary to have an external arbiter that provides overall information about the status of the system. In order to exploit all the functionalities offered by the runtime manager, the scheduler of the system should be implemented in order to be able to create a logical distinction of the execution units in two sets: the first one reserved for the guarantee-throughput processes, and the second one for best-effort applications. The guarantee-throughput processes need to have reserved units because they should always respect the QoS requirements and they should not be slowed down in a situation in which the best-effort processes are using all the resources of the system. In the case of a saturation of the units reserved to the guarantee-throughput applications, the scheduler should assign the exceeding threads to the units of the best-effort set. Obviously, if the best-effort units are all in use, the exceeding threads of best-effort applications will not use the units reserved for the guarantee-throughput applications, but will just be queued. The scheduler should also be ready to receive additional information from the runtime manager. As it has been already described, in every operating point the developer can specify the affinity of the threads of that section. This information can be used by the scheduler during the allocation of the threads to the processors, as an additional knowledge that can improve the execution performance.

The operating system should be able to provide information on the active processes differentiating between the set of processors that will be used by the guarantee-throughput applications and the remaining ones. To improve the performance of the runtime management system, the operating system should also provide information that is updated frequently, in order to allow the runtime manager to know the current status of the system.

The current Linux kernel, version 2.6.35.5, does not implement completely these requirements. The execution units cannot be divided logically in sets and it is not possible to provide the affinity information directly to the scheduler. It also provides information on the average load related to a temporal window of one, five and fifteen minutes and that cannot be considered valid indexes of the current status of the system. However, through the special file /proc/loadavg, it is possible to find on a single line the three average loads and the number of active processes, allowing a fast parsing of this information.

To verify that all the functionalities implemented in the runtime manager works appropriately, it has been implemented an architecture simulator that is able to simulate an hardware architecture with a configurable number of processors and, also, the scheduler that assigns the threads to the cores. Its implementation will be described in detail in Section 4.1.

3.5 Local Runtime Manager

As it was described in the overview, the runtime manager is the main block of the system. It has been developed to be light, fast and configurable. It should be light and fast because it is necessary to reduce the overhead introduced by the runtime management methods. The runtime manager procedures that handle the parallelization policies are invoked by a modified version of the *libGOMP* library. This choice was made to avoid the need to change the source code of an OpenMP application to use the runtime manager. The recompilation with a modified toolchain is the unique operation requested for not-profiled OpenMP applications in order to take advantage of the runtime manager optimizations. This is not a real concern for a software developer because it is already common use, for the hardware producers, to provide a specific toolchain to those who want to develop software for a specific system. For the profiled OpenMP applications and for not-profiled ones that do not use OpenMP directives, it is also necessary to add a few lines of code to the

```
int rtmgmt_start_section( int section_id,
                          int* thread_id,
                          unsigned nest_level,
                          double completion_time,
                          unsigned num_threads );
int rtmgmt_end_section( int section_id,
                        int* thread_id,
                        unsigned nest_level );
int rtmgmt_start_thread( pthread_t *thread,
                         const pthread_attr_t *attr,
                         void *(*start_routine)(void*),
                         void *arg );
void rtmgmt_exit_thread();
```

Listing 3.1: The API for the invocation of the parallelization policies.

sources, because it is necessary to add some functionalities that were not present before. In Appendix A it is possible to find an example of a profiled application where the added lines are highlighted and to see how few changes are necessary in the sources to utilize the runtime manager.

The *librtmgmt* library offers an Application Programming Interface (see Listing 3.1) that allows the evaluation of the parallelization policies. In this thesis it is used only by the *libGOMP*, but these methods can be invoked also from other libraries or from autonomic applications.

The runtime manager is executed before a parallel section begins its execution with a call to the method `rtmgmt_start_section`, that receives as input the id of the section, the id of the parent threads and the nesting level only if the section is nested, the execution deadline and the request of a preferred parallelization level by the programmer (that can be specified in OpenMP with the clause `num_threads`) and returns an integer that represents the parallelization level. The main aim of this method is to determine the best parallelization policy for the section that is going to start, using a

set of information collected from the operating system and from the behavior of the application itself during previous executions. For the applications that are not profiled, these are the only data that will be used. The profiled applications can also use the information stored in the operating points, that provides hints on the execution time of the section.

The runtime manager executes some operations also at the end of each section, in order to evaluate if the QoS requisites are fulfilled and to measure and store the effective execution time of that section, so it will be possible in the subsequent executions to make a better choice. The method invoked by the *libGOMP* library is called `rtmgmt_end_section`, in which it is necessary to supply the data to identify the ending section, that must have the same values of the three parameters used in the corresponding `rtmgmt_start_section` method when the section was starting.

In order to allow a low-level optimization that can be performed by the scheduler, the runtime manager library is also executed when the threads are spawned and killed through the invocation of the methods `rtmgmt_start_thread`, with the same input parameters of a `pthread_create` method, and `rtmgmt_exit_thread`. In this way the runtime manager can inform the scheduler of some properties of the threads before they are spawned, like the processor affinity. In some architectures it is important to choose where a process will be executed, because different processors have different access time to the memory hierarchy. If the scheduler knows in advance if the threads are affine and need to be executed on the same cluster of processors, its choices can be improved and optimized.

3.6 Runtime Management API

Other than the procedures invoked as an integration with *libGOMP*, the runtime manager library offers a set of functions that should be used in order to profile an application, load the operating points and get information on how the application itself is performing. It is defined an Application

```
int rtmgmt_load_op( char *op );
int rtmgmt_best_effort( char *op );
void rtmgmt_profile( int* section_ids,
                     unsigned num_sections,
                     unsigned num_repetitions );
void rtmgmt_end_profile();
int rtmgmt_set_priority( unsigned priority );
float rtmgmt_get_section_period();
```

Listing 3.2: The API for the applications.

Programming Interface (API), shown in Listing 3.2, for six methods that can be invoked inside the applications that include the `rtmgmt.h` header file. The first pair of methods can be used to load the operating points:

- **rtmgmt_load_op**:
  It loads the operating points from a file specified as the first parameter. When an application use this function and a valid file is provided, the operating points are loaded through a new thread and the runtime manager will consider the application as profiled. The use of a new thread is a way to avoid to slow down the execution of the application while the operating points are loaded.

- **rtmgmt_best_effort**:
  It loads the operating points as the previous method, but labels the application to be serviced as best-effort.

The runtime manager is also capable of profiling an application. This functionality can be used through two methods:

- **rtmgmt_profile**:
  With this method it is possible to perform a complete profiling of the application that uses it. First, it is necessary to load a set of operating

points, that represent the list of configurations that will be tested during the profiling. The points may contain any value as the execution time, because this value is not considered and it will be replaced with the result of the process. It is also necessary to provide a list of the id numbers of the sections that should be profiled and the number of times that each configuration, namely each operating point, should be measured. When the profiling mode is activated, the parallelization policy is modified. When a section is starting, the runtime manager will choose an operating point related to that section, randomly. When every operating point has been used for the number of times specified in the `rtmgmt_profile` call, the runtime manager will save all the profiled date and will shut down the application.

- **rtmgmt_end_profile**:
  If the application is not able to perform a complete profiling, for example if its execution time is short and there are a lot of operating points to be measured, it is necessary to invoke this method just before the application exits. All the profiling information collected will be saved and, therefore, it is possible to profile completely the application with few executions.

There are also two useful methods to specify the priority and to obtain an indication of the performance of the application:

- **rtmgmt_set_priority**:
  It allows to set a priority level for the current application as a non-negative integer. The maximal priority is zero and the priority level decreases as the specified number increases. The priority will be passed to the scheduler and used in the case of a preemptive scheduler and few idle processors. In this situation the scheduler may use the priority information to choose to preempt a process with low priority to privilege the execution of a process with higher priority. The use of this method is not allowed for not-profiled applications.

- **rtmgmt_get_section_period**:
  It is created to allow the application to know the execution time of a section. If the method is called inside a section it will return the last execution time of that block of code. If it is called outside any section or if it is the first execution, the result of the invocation will be zero. This functionality can be extremely useful for applications that can modify their behavior at runtime and is essential for the development of autonomic applications. A practical example is a video decoding application that can adapt the parameters of the decoding routine based on the number of frames decoded per second. If the application is able to identify if the performance is decreasing, for example if other applications are using all the resources of the system, it may decide to reduce the output quality or to change the decoding algorithm with a faster but lighter one.

3.7 Parallelization Policy

In this section it is described how the runtime manager decides the parallelization level that will be assigned to the application. In other words, the choice of how many threads will be spawned to execute the parallel section. The policy is different for the profiled and not-profiled applications. For the first ones, it is possible to use the information provided by the operating points in order to decide which is the best parallelization level. For the second ones, there is no information until the program executes some iteration of the parallel sections.

3.7.1 Profiled Guarantee-Throughput Applications

As it has already been shown before, a profiled guarantee-throughput application is composed of a binary executable and a text file that contains the operating points. The objective of this kind of applications is to provide a service that respects Quality of Service (QoS) requirements, expressed as a

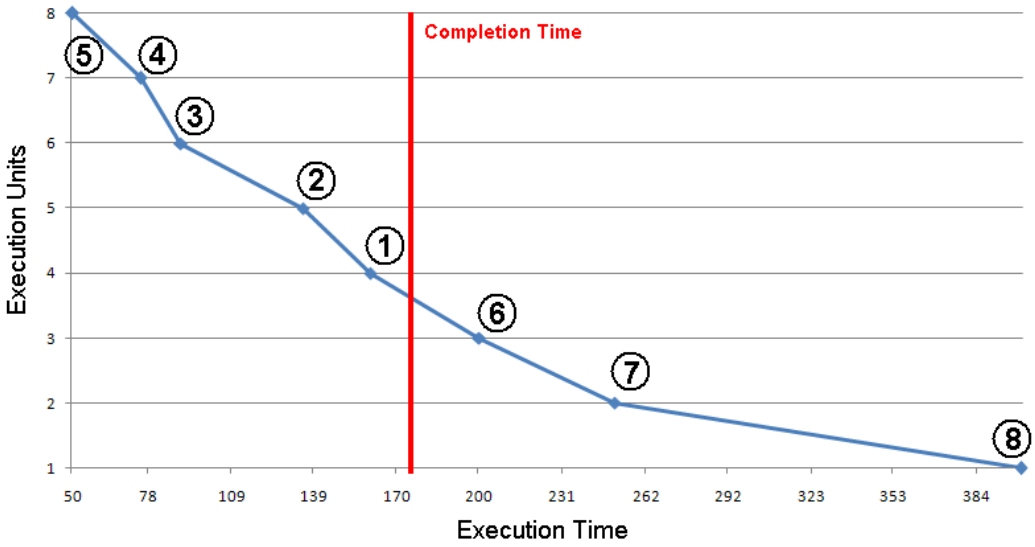

Figure 3.3: An example of how the operating points are ordered based on the completion time and the execution time.

completion time that is specified by the programmer for each parallel section. The runtime manager must identify the operating point that allows to execute the section without violating the QoS requirements, using the minimum number of resources in order to minimize the power consumption of the system.

The first step of the algorithm is the identification of the minimum completion time, in case of nested sections. In this situation, the runtime manager should guarantee that also the parent sections respect their completion time. For example, if a parent section specifies a completion time of 500 ms and a nested section specifies 1000 ms, the runtime manager will choose the parallelization policy in order to execute the nested section in at least 500 ms minus the time already passed between the start of the parent and the nested section. After having chosen the valid completion time, the operating points are ordered in a list using the difference between the execution time of the point and the completion time.

In Figure 3.3 it is shown an example of a set of operating points with their

execution time on the y-axis and the number of execution units on the x-axis. The red line represents the completion time of the section. The number in the circles represents the order in which the points are added to the list. The first chosen point is the closest one to the completion time line but that has an inferior execution time. It is the best operating point because it has an execution time sufficient to respect the requirement and it uses the minimum number of execution units. Then the other points are added to the list in descending order of the execution time (numbers 2-5). In this way the other operating points that respect the QoS requirement are inserted in the higher positions in the list. Then, the remaining ones are added to the list in ascending order of their execution time (numbers 6-8). These are all the points that have an execution time superior to the completion time specified by the programmer. After the creation of the list, it is requested that the operating system provides the number of currently idle processors reserved for the guarantee-throughput applications. The list is then traversed and the first point with a number of execution units that is at most the number of idle processors is selected. If a point with this characteristic is not found in the list, the first point of the list is selected.

3.7.2 Profiled Best-effort Applications

These best-effort applications are profiled, and therefore their behavior at runtime is described in a set of operating points, but they do not have to respect any QoS requirement. The runtime manager should execute these kind of processes in order to maximize the throughput. The policy used for these applications is very similar to the one used for guarantee-throughput applications. The operating points are still ordered in a list, but without considering the completion time line. The points are just added to the list in ascending order of the execution time. The parallelization level will be chosen, as for the profiled applications, traversing the list to find the first point with a number of execution units that is less than or equal to the idle units of the system reserved for the best-effort processes. Indeed, these

applications will not occupy the execution units reserved for the guarantee-throughput processes.

3.7.3 Not-profiled Best-Effort Applications

In the scenario of the applications that are not profiled, it is not possible to know in advance how the application itself performs when it is executed on a particular number of execution units. It is worth to remember that these applications do not have to guarantee QoS requirements but should be executed in order to maximize the throughput. The allocation of all the execution units to a single application can be the best choice to optimize the performance when there are not other applications that need to share the resources. In the case of systems with concurrent processes, it is necessary to use policies that maximize the performance of every process and allocate the resource equitably. If a single application saturates the resources, any other application would be queued and its throughput would be highly reduced. Rosti et al. [25] show that, in multiprocessors systems, processor saving scheduling policies, that are policies that keep some of the available processors idle in the presence of work to be done, yield better performance than their corresponding work-conserving counterparts, especially when the workload is heterogeneous or with irregular execution time distributions.

The policy for not-profiled applications is based on the measurement of the last execution time of each section. This value is then used to classify the section as fast, medium or slow dynamic. The distinction between these three classes allows the application of different rules and a better utilization of the resources. Indeed, a fast section executes short burst of computation and thus it is possible to assign more resources because it will release them in a short time and, in case of a saturation of the execution units, the assignment can be rapidly modified. On the contrary, a section with a slow dynamic could block the resources for too much time and it is better to make a more conservative assignment. The execution time of a section is evaluated as fast if it is below a threshold defined as a parameter of the runtime

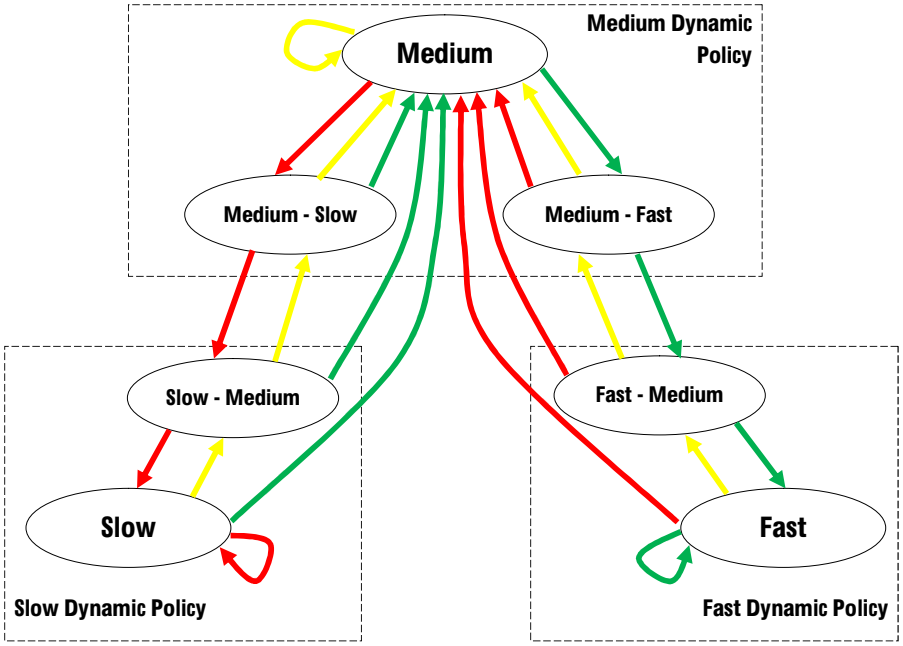

Figure 3.4: The scheme of the adaptation of the policy based on the last execution time.

manager, that is called `FREQUENT_NP_SEC`, expressed in seconds. It is evaluated as medium if the time is between the parameters `FREQUENT_NP_SEC` and `NOT_FREQUENT_NP_SEC`, otherwise the execution time is considered slow.

When a section is executed for the first time, it is classified as `Medium`. By using the last execution time of the section, the classification is adapted, following the scheme of Figure 3.4. If the last execution is evaluated as fast, the classification is modified following the green line. Otherwise if it is evaluated as medium it will be used the yellow line, and finally the red line when the execution time is marked as slow. If the section is classified as `Medium`, `Medium-Slow` or `Medium-Fast` it is adopted the `Medium Dynamic Policy`, if it is classified as `Slow` or `Slow-Medium` it will be chosen the `Slow Dynamic Policy` and for the remaining two classes, `Fast` and `Fast-Medium`, it will be adopted the last policy, namely `Fast Dynamic Policy`.

For example, if the section is classified as `Medium` and the last execution is measured as fast (green line), the section will be classified as `Medium-Fast`, but it will still use the `Medium Dynamic Policy`. If the subsequent execution time is measured again as fast, the section will be classified as `Fast-Medium`, and it will use the `Fast Dynamic Policy`. If the section is classified as `Fast` and the last execution is evaluated as Slow, the classification will become `Medium` and the policy will be changed. This scheme allows an adaptation between the three policies in case of a change in the dynamic of the section. It is also noise-resistant, because a single evaluation that is different from the current classification is not enough to adopt the relative policy.

The three policies used in the runtime manager are similar and differ only by one parameter. Therefore, it is possible to describe just the general scheme of the policies and focus on the differences to fully understand the algorithm. Firstly, the algorithm check if the section is nested. In this case the parallelization policy assigns just one unit, because a choice has already been taken in the parent parallel section. The application will continue to execute with the same level of parallelization assigned in the parent section. To avoid an excessive overhead in applications that execute a great number of short parallel sections, the execution of the code that choose the parallelization level is not performed at every start of a section, but it is executed once every a number of seconds, specified in a parameter called `NP_FILTER_SEC`. During this window of time, it is used the parallelization level found when it was performed the last choice.

The choice is based on the current number of idle execution units, on the average idle units during the last second and on the classification of the section. The average measure can be considered a prediction of the status of the system in the following second. The current and the average number of execution units are added up and multiplied by a factor that is different for the three classes. In other words, the result of this operation is a combination of the instantaneous number and the prediction for the next second, of idle units. If the factor has a value of one half, the result is

the average between the two numbers. The factors are specified with three configurable parameters of the runtime manager:

- `FAST_DYNAMIC_FACTOR` is the factor that will be used directly when the section is classified as fast
- `MEDIUM_DYNAMIC_CORRECTION` is a value that will be multiplied by the `FAST_DYNAMIC_FACTOR` to obtain the factor for the medium dynamic sections
- `SLOW_DYNAMIC_CORRECTION` is used in the same way of the `MEDIUM_DYNAMIC_CORRECTION` but for the slow dynamic sections.

The two correction parameters must have a value between zero and one, because the medium and slow factor must be lower than the fast factor.

The difference between the number of execution units assigned in the last execution of the section and the result of the multiplication of the class factor with the sum of the average and current idle units is, then, divided by two. This operation is made in order to create a transient. The result of the division is added again to the number of units assigned in the last execution to form the new assignment.

The policy that has been just described has the advantage of being deterministic. The assignment of the execution units between a set of applications can be described by an equation. It represents the total number of assigned units when the transient state ends and can be expressed as:

$$\frac{\frac{2}{\left(\frac{1}{factor1}-1\right)}+\frac{2}{\left(\frac{1}{factor2}-1\right)}+...+\frac{2}{\left(\frac{1}{factorN}-1\right)}}{1+\frac{2}{\left(\frac{1}{factor1}-1\right)}+\frac{2}{\left(\frac{1}{factor2}-1\right)}+...+\frac{2}{\left(\frac{1}{factorN}-1\right)}} * numEU \tag{3.1}$$

where factor represents the value of the class factor of the application, and numEU is the number of execution units assigned to the not-profiled applications. As it is possible to see in the equation, the order of execution of the applications is not relevant for the number of assigned units after

the transient state. It is also possible to determine the assignment for each application of the set:

$$(numEU - totalEUassigned) * \frac{2}{\left(\frac{1}{factor} - 1\right)} \tag{3.2}$$

where totalEUassigned is the result of the Equation 3.1. It is also possible to simplify the Equation 3.1 to show the assignment when the applications have all the same dynamic and, therefore, an equal factor:

$$\frac{2 * N}{\left(\frac{1}{factor} - 1\right) + 2 * N} \tag{3.3}$$

where N is the number of executing applications. Let's suppose to have a scenario in which two, three or four applications with the same dynamic, run concurrently on a system with 100 processors. In the Table 3.1 it is possible to see how the number of assigned units changes when the factor is set to 1/2, 2/3 and 3/4.

		Factor		
		1/2	2/3	3/4
2	Total	80	89	92
	Each	40	44	46
3	Total	86	92	95
	Each	29	31	32
4	Total	89	94	96
	Each	22	24	24

Table 3.1: The execution units assigned to different sets of applications with the same dynamic, that run concurrently.

An additional feature has been added in order to obtain the complete utilization of the resources. It is possible to specify, through a configurable parameter called `LOAD_LIMIT`, the maximum value of the load of the system for which the policy just described will be applied. When the load exceeds

the value specified in the parameter, an application that is going to start a parallel section will be parallelized over all the remaining idle execution units. In this way the resources are saturated and it is possible to reach the complete utilization of the execution units. It is necessary to be careful when choosing the value of the parameter, because if the value is too low, the system will saturate with few applications and all the processes that will start afterwards would remain queued.

Chapter 4

Framework Implementation

After the description of the Runtime Management System, this chapter reports how it can be tuned up through a design space exploration technique, in order to find the best configuration of the parameters for a system, based on the kind of applications that will be executed on it. In order to evaluate the RMS and the Configuration Framework on a Many-Core Architecture, it has been developed a simulator with a configurable number of processors and that simulates the allocation of threads to the execution units. Then, it will be described the software architecture of the Framework and it will be shown the results of the utilization of the tool in an example case.

4.1 Many-core Architecture Simulator

The main goal of this thesis is to develop a system that is able to optimize the execution of applications and the resource sharing between them. The hardware architecture target for this work is the family of processors that are called *Many-Core processors* because they support up to hundreds of cores. These kind of architectures are still in a developing phase and are not available in the consumer market. This is the reason why it has been necessary the implementation of a simulator.

The main task of this software are:

- the simulation of a processor architecture that is composed of a configurable number of cores;
- the simulation of a scheduler that assigns the threads to the execution units following the directives provided by the runtime manager;
- the possibility to show the status of the allocation of the cores during a simulation on a graphical user interface and to output a trace of the simulation that can be analyzed offline;

The cores of the simulated architecture are partitioned in two sets: the first is the set of execution units reserved for the guarantee-throughput applications, and the second is the set used by the best-effort applications. The scheduler provides information on the number of active and queued processes, the number of idle execution units and the average load in the last second with distinct values for the two sets of cores. It is also able to receive additional information on the threads like the affinity, but in this work this kind of data is not used to modify the allocation of the threads.
The runtime manager interacts with the simulator through shared memory. Every time the runtime manager is generating new threads to parallelize an application, it enqueues the number of threads to be allocated, plus other information, in a data structure of the shared memory segment. The same operation is done also when the parallelized section ends. The structure can enqueue in an array multiple set of information from different applications. Every 100 ms the main routine of the scheduler is executed. Firstly, it checks if the processes currently allocated are still running. If a process is found to be dead, the cores that was allocated to him are freed up. Then, all the sets that was added in the data structure are processed in order. The routine ends with an update of the graphical interface and with the output of a trace of the status of the cores after the assignments just made. In Figure 4.1 it is possible to see an example of the graphical interface during a simulation, with an architecture constituted of twelve cores, where four cores are assigned to

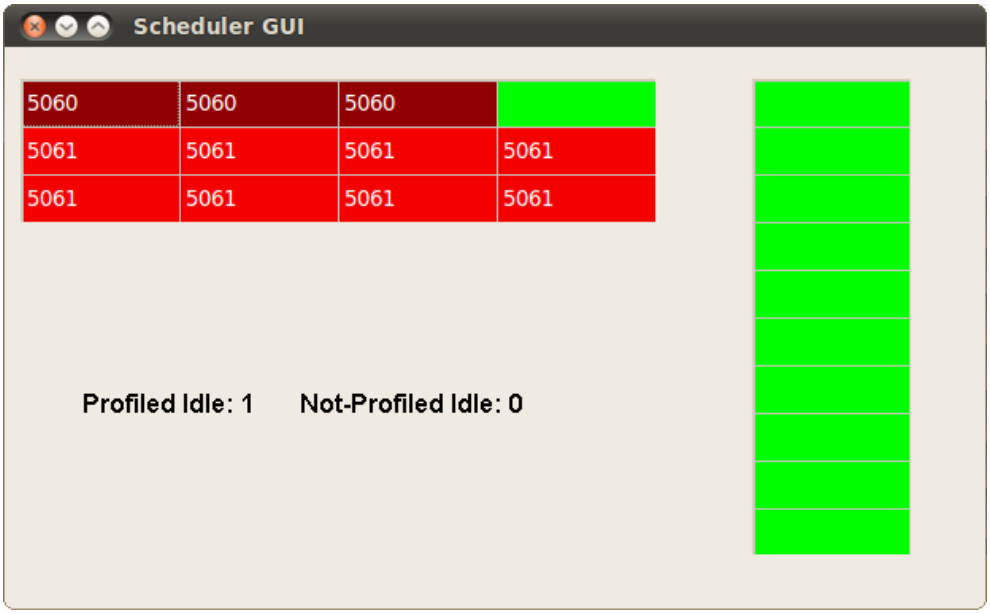

Figure 4.1: The graphical user interface of the Many-core Architecture Simulator.

the profiled applications. In the central part it is placed a grid of cells that represents the set of cores. The green cells are the free cores, while the cells with a number inside are the used cores. The number is the process identifier of the process that is currently using the core. The grid on the right side represents the queue of active processes and it is filled when a process cannot be allocated to a core because there are not free execution units. In the bottom of the figure it is also shown the number of idle execution units both for the cores reserved to the profiled application and for the remaining ones.

4.2 Design-time Configuration Framework

As it was described in Section 3.5, the runtime manager developed in this thesis is configurable in order to be adapted to different scenarios. There are five parameters that can be changed and that modify the behavior of the manager:

- **cpu_for_profiled:**
  It is the number of execution units reserved to the profiled guarantee-throughput applications.

- **np_filter_msec:**
  For this amount of time, the runtime manager does not modify the number of core assigned for the not-profiled applications. It is used to reduce the overhead.

- **frequent_np_msec:**
  If a section of a not-profiled application is executed within this amount of time, it is classified as a fast section.

- **not_frequent_np_msec:**
  If a section of a not-profiled application has an execution time greater than this amount of time, it is classified as a slow section.

- **load_limit_percent:**
  When the average load of the system is greater than this value multiplied by he number of execution units, the runtime manager will assign to the first request all the remaining idle resources. It is a solution to allow the complete utilization of the resources.

Obviously, it is not simple to choose the correct value for this parameters to allow an optimal execution of the application of the system. It is also intuitive to notice that there is not a single optimal configuration, but it depends on the set of applications that will be executed in the system. For these reasons, it is necessary to find a tool that may help a system administrator in the complex task of finding the best configuration for a particular system. The administrator should provide a description of an expected scenario of applications in execution and the tool should test different configurations of the Runtime Management System and evaluate their goodness. The best one will be the more appropriate configuration for the provided scenario.

```
<?xml version="1.0"?>
<frms path='/absolute/path/to/scenario-simulator/'>
        <profiled num='2'>
                <app filename='/absolute/path/to/app1'
                        args='argument1 argument2'/>
                <app filename='/absolute/path/to/app2'/>
        </profiled>
        <scenario simulation_time='4'>
                <slow start='0' end='3'/>
                <medium start='0'/>
                <fast start='1' end='2'/>
        </scenario>
</frms>
```

Listing 4.1: An example of the description of a scenario.

4.2.1 Scenario Simulator

In order to allow the simulation of a scenario of executing applications, it has been implemented a tool that receives as input the description of a scenario and the configuration of the Runtime Management System. Then, the scenario is simulated following the details specified in the inputs and the final result of this execution is a set of values that indicates some characteristics of the system during the simulation.
The first step for the utilization of the tool is the definition of a scenario. It should be described by the system administrator, that is aware of the set and of the dynamic of the applications that will be run on the system. It can be defined through an XML document, in which it is necessary to specify a set of profiled applications and the dynamic of a set of not-profiled applications.

In Listing 4.1 it is possible to see an example of scenario. In the `profiled` section it is possible to specify any number of applications that will be run during all the simulation. It is also possible to specify a list of arguments

```
<?xml version='1.0' encoding='UTF-8'?>
<simulator_input_interface>
        <parameter name='cpu_for_profiled' value=''/>
        <parameter name='np_filter_msec' value=''/>
        <parameter name='frequent_np_msec' value=''/>
        <parameter name='not_frequent_np_msec' value=''/>
        <parameter name='load_limit_percent= value=''/>
</simulator_input_interface >
```

Listing 4.2: An example of the configuration input.

that will used when the application is executed. These programs should be the core services of the system or, in general, the set of profiled applications that will be executed on the system. In the `scenario` section, it is possible to indicate the duration of the simulation, expressed in seconds, and a set of applications, defined by their execution dynamic. For each application in the scenario it is possible to specify if it has a `slow`, `medium` or `fast` dynamic and the execution interval. Through the attributes `start` and `end`, it can be specified after how many seconds the application starts and when it ends. If the attributes are not specified or exceed the simulation interval, the default values are `start` equal to zero and `end` equal to the `simulation_time`.
This tool receives as input also the set of parameters of the Runtime Management System that will be used during the simulation. It is possible to see an example of the format of the XML file in Listing 4.2. The five parameters are the same that was described earlier in this chapter and the attribute `value` must be filled with the value of the parameter.

If all the input are correct, the simulation starts. After that the simulation time, specified as input, has elapsed, all the processes are killed, and the tool creates an XML document with this set of metrics:

- **profiled_app_queued & not_profiled_app_queued**: the number of threads that have been queued because there were not available

cores.
This value is distinguished between profiled and not-profiled applications. This metric should remain close to zero, because it would be the cause of a slow down in performance due to the stalemate of some threads.

- **profiled_avg_load & not_profiled_avg_load**: the average load of the cores.
  Also this value is expressed as two numbers: the first is referred to the cores reserved for the guarantee-throughput applications and the second for the best-effort ones. This value should be maximized for the cores reserved for best-effort applications, because it indicates how much the execution units have been used.

- **qos_failures**: the number of violations of the QoS requirements, for the guarantee-throughput applications.
  This metric should be zero to consider acceptable a configuration. If the tool reports a value that is greater than zero, the system is not respecting the requirements in the simulated scenario.

- **profiled_overflows**: the number of threads of guarantee-throughput applications that use the cores reserved for the best-effort ones.
  This situation happens when the execution units reserved for the guarantee-throughput programs are all in use. If this number is greater than zero, the administrator should consider the possibility to reserve more cores for the profiled applications.

In Listing 4.3 it is possible to see the format of the XML output document where in the attribute `value` it will be placed the value of the metric.

4.2.2 Design Space Exploration

The Scenario Simulator tool is able to evaluate the execution of a set of application with a single configuration of the *Runtime Management System*.

```
<?xml version='1.0' encoding='UTF-8'?>
<simulator_output_interface>
 <system_metric name='profiled_app_queued' value=''/>
 <system_metric name='not_profiled_app_queued' value=''/>
 <system_metric name='profiled_avg_load' value=''/>
 <system_metric name='not_profiled_avg_load' value=''/>
 <system_metric name='qos_failures' value=''/>
 <system_metric name='profiled_overflows' value=''/>
</simulator_output_interface >
```

Listing 4.3: An example of the output XML document.

However, we are interested in the evaluation of a set of configurations, to find the best solution for the system that has been modeled during the definition of the scenario document. This kind of task is an optimization technique, called Multi-Objective Design Space Exploration, that has already been illustrated in Section 2.4. In this thesis, it will be used the framework *Multicube Explorer* to perform this task, for which it is just required to specify the design space to be explored. It is necessary to indicate the absolute path to the *Scenario Simulator* with the path of the scenario description as the first parameter, the input parameters to the simulator for the configuration of the *Runtime Management System* with their intervals, and the expected output.

In Listing 4.4 it is possible to see a reduced version of an example of the design space definition. The system metrics are the same that have already been described for the output of the *Scenario Simulator*. It is possible to specify the range for each input parameter and how the values are incremented. In the shown example, the parameter `cpu_for_profiled` will assume the values 2, 4, 6 and 8. The framework will invoke the simulator with all the possible combinations of the specified parameters, and it will collect the results. Other than executing a series of simulations, Multicube Explorer is also able to generate a graphical report of all the design space, that allows to easily identify the property of the system and to choose the

```
<?xml version='1.0' encoding='UTF-8'?>
<design_space xmlns='http://www.multicube.eu/'
version='1.3'>
    <simulator>
        <simulator_executable
        path='/absolute/path/to/the/scenario-simulator/
        --scenario=/absolute/path/to/the/scenario.xml'/>
    </simulator>
    <parameters>
        <parameter name="cpu_for_profiled"
        type="integer" min="2" max="8" step="2"/>
        <parameter name="np_filter_msec"
        type="integer" min="400" max="600" step="100"/>
        <parameter name="frequent_np_msec"
        type="integer" min="100" max="300" step="100"/>
        <parameter name="not_frequent_np_msec"
        type="integer" min="300" max="600" step="100"/>
        <parameter name="load_limit_percent"
        type="integer" min="70" max="90" step="10"/>
    </parameters>
    <system_metrics>
        <system_metric name="profiled_app_queued"
        type="integer" unit="" desired="small" />
        ...
     </system_metrics>
</design_space>
```

Listing 4.4: An example of the design space definition XML document.

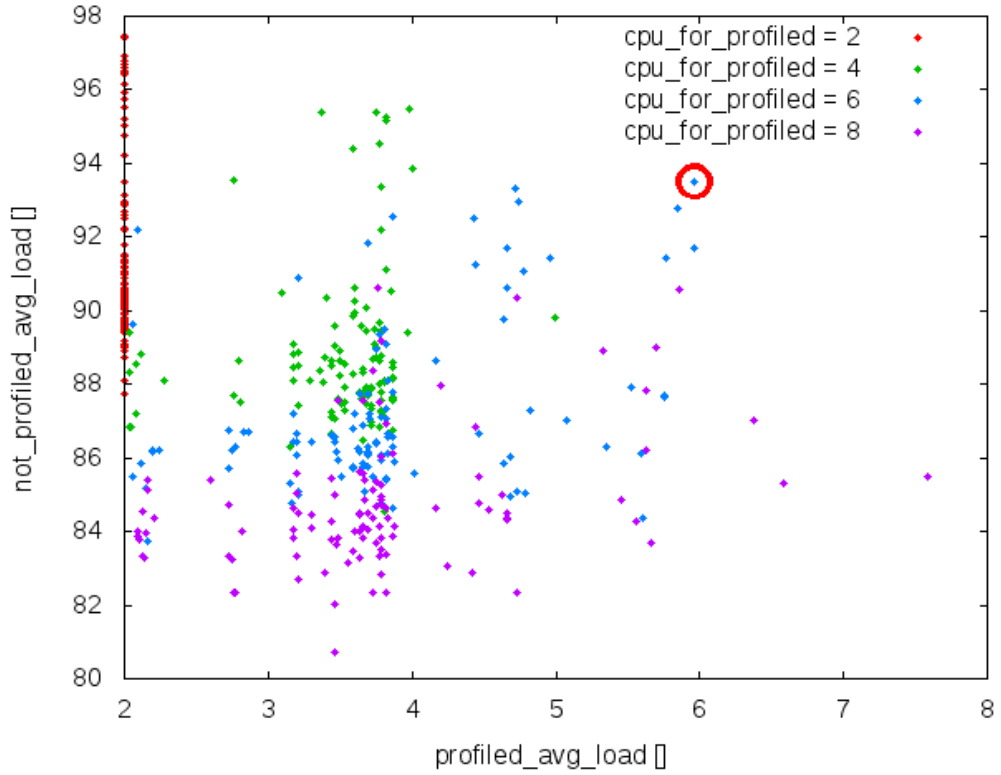

Figure 4.2: The objectives space focused on cpu_for_profiled, projected on profiled_avg_load and not_profiled_avg_load.

best values.

In Figure 4.2 it is possible to see a projection of the objectives space on `profiled_avg_load` and `not_profiled_avg_load`. The points are classified by the values of the parameter `cpu_for_profiled`. The dot with a red circle is the Pareto efficient point. This figure is indicating that the reservation of six cores for the profiled applications is the best trade-off that maximizes the average load both for profiled and not-profiled cores. It is also possible to see that if the goal is to maximize just the profiled average load, the best choice is the dot that is the first on the left side, and that represents eight reserved cores. On the contrary the best choice to optimize just the not-profiled average load is the first dot on the top of the graph, that represent a reservation of two cores for the profiled applications.

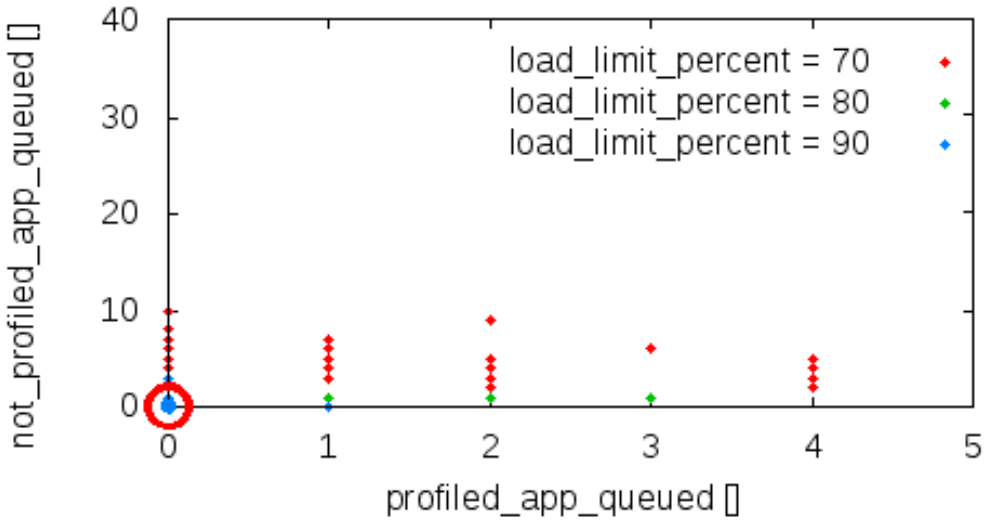

Figure 4.3: The objectives space focused on load_limit_percent, projected on profiled_app_queued and not_profiled_app_queued.

In Figure 4.3 it is possible to see another example of objectives space. In this case we are looking for a value of the parameter `load_limit_percent` that minimizes the threads queued for both profiled and not-profiled applications. The graph shows that this condition is respected when the value is 90. During these simulations, when the parameter is set to this value, the applications have never been queued.

Multicube Explorer allows also the representation of box plots, that shows the effect of the variation of a parameter on a system metric. In Figure 4.4 it is shown the variation of the parameter `np_filter_msec` and the effect on the metric `not_profiled_avg_load`. The black line represent the median, while the box is delimited by the first and third quartiles. It is possible to see that if the runtime manager maintains an assignment for a longer time, the performance of the not-profiled applications improves. In this case, on average, two more cores are used when the filter is set to 600 ms with respect to the other two values.

This Framework is a great tool for a System Administrator, because it allows to run automatically a series of simulations of the system that is going

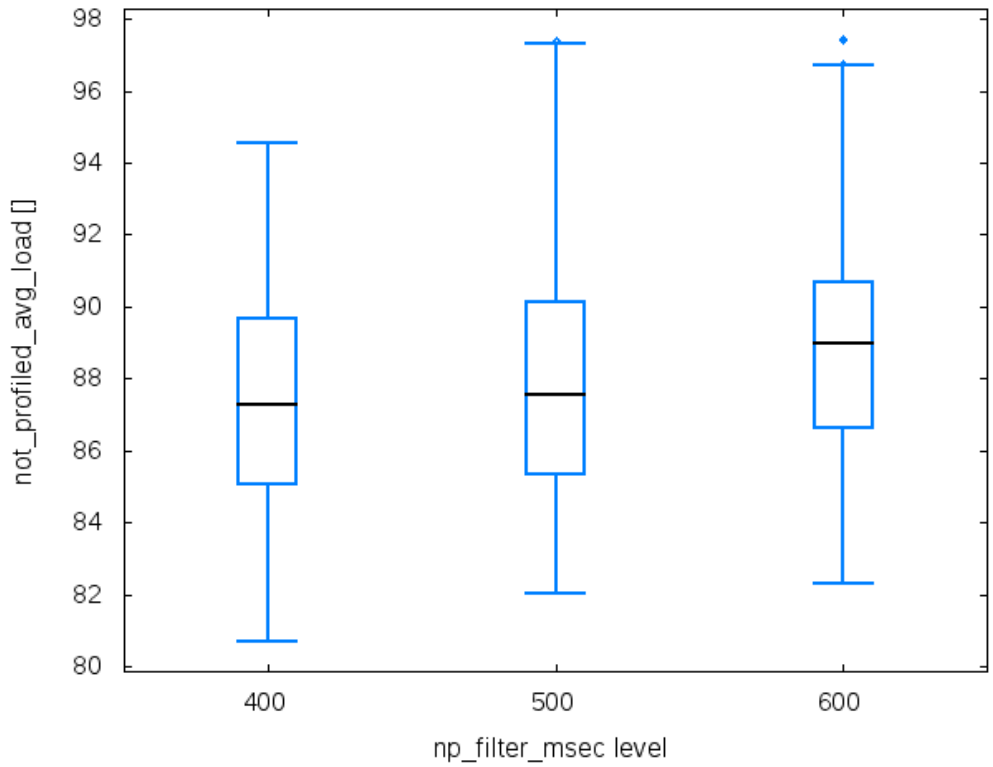

Figure 4.4: Box plot of not_profiled_avg_load, focused on np_filter_msec.

to be designed. The administrator should only define the test scenario and the intervals of the parameters. After that, he will have a series of graphs that shows the system behavior and allows to take the appropriate decisions to optimize the configuration, with a consistent time-saving.

Chapter 5

Experimental Results

In this chapter it is shown the behavior of the *Runtime Management System* through different experimental analysis. First, it has been traced the functioning of the parallelization policy for not-profiled applications. This analysis has been conducted on the *Many-core Architecture Simulator*, that has been introduced in Section 4.1, with one hundred cores. To verify the impact of the *Runtime Management System* on a real system, it has also been carried out an overhead analysis on a 16-cores processor.

5.1 Analysis of the Parallelization Policy

After the theoretical description of the policy that is applied to the not-profiled applications, it is interesting to analyze how the *Runtime Management System* performs when it is used on a system with many cores. The analysis has been realized on a simulated architecture with one hundred cores, all reserved to the not-profiled applications. The main goal of this analysis is to highlight the behavior of the system under particular situations. In the first example, in Figure 5.1, it is possible to see the assignment of cores with four equal applications that start sequentially every two seconds, in a simulation of ten seconds. The violet line represents the sum of all the assignments and it coincides with the line of the first application, the blue one, as long

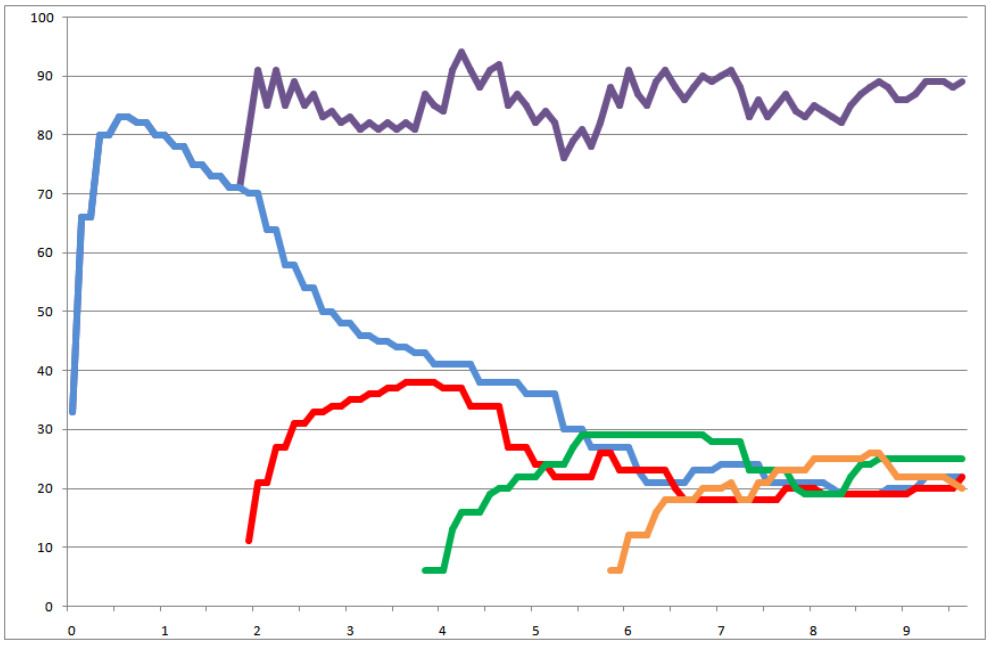

Figure 5.1: An example of the assignment of cores when four applications with fast dynamic are executed sequentially.

as it is the only application that is being executed. In the assignment of the first application it is noticeable a peak at 0.5 seconds. It is due to the use of the average utilization of the resources during the last second as a parameter for the calculation of the assignment. This parameter underestimates the real utilization before the first second has passed. The red, green and orange lines are the representation of the assignments for the other three applications. Because of the fact that the applications have the same dynamic, the assignment of execution units stabilize itself on equal or very close values. At the end of the simulation the sum of all the assignments is equal to 89 and the four applications use 22, 22, 25 and 20 cores. All these applications have a fast dynamic and the `FAST_DYNAMIC_FACTOR` parameter of the run-time manager is set to 1/2. In the Table 3.1, by looking at the third row and at the first column, it is possible to see that the values of the simulation have been well predicted by the equation 3.3, that specifies a total assignment of 89 with 22 cores for each one of the four processes.

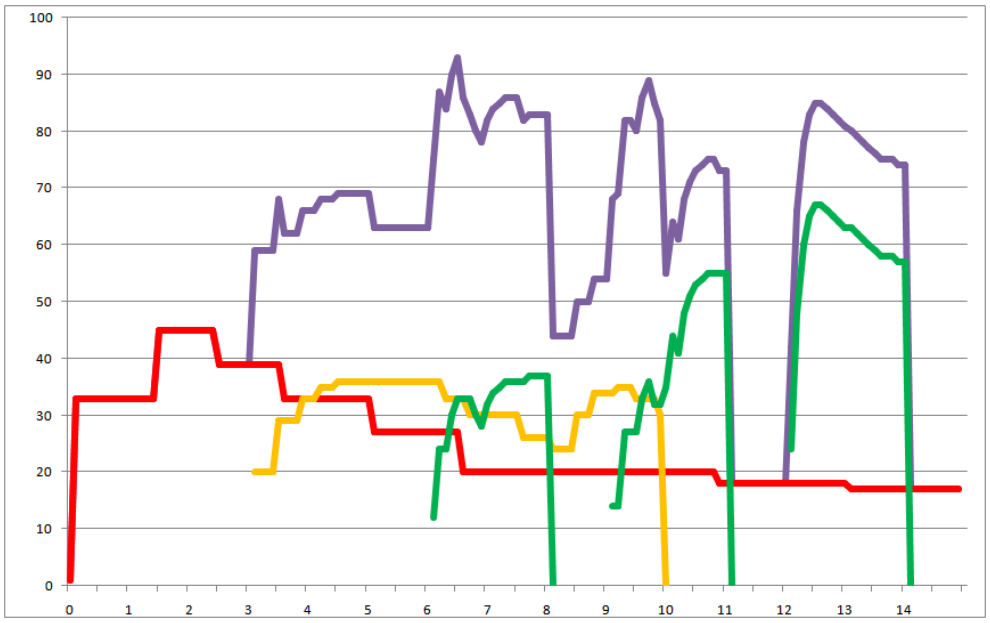

Figure 5.2: An example of the assignment of cores with applications that have all the three kind of dynamic.

In Figure 5.2 it is shown a simulation with a more variegated scenario. The first process that starts is a slow application, marked with a red line. As it is noticeable, it changes the assignment with a low frequency. This delay in the adaptation to the system changes is the principal reason why the policy elaborated in this thesis assigns less resources to this kind of applications: they are less ready to release the cores if the system is in a condition of saturation. As in the previous graph, the violet line represents the sum of the utilization of the execution units by all the processes. The second process is an application characterized by a medium dynamic and it is marked with a yellow line. It starts at the third second and ends at second ten. After one second, the number of assigned cores overcomes the assignment of the slow application and the values become stable. Then it is added to this scenario a fast application with three burst, represented with a green line. Each burst has a duration of two seconds. In particular in the second burst, when at second 10 the medium dynamic application ends, it is possible to see how the assignment is adapted when the number of active applications changes.

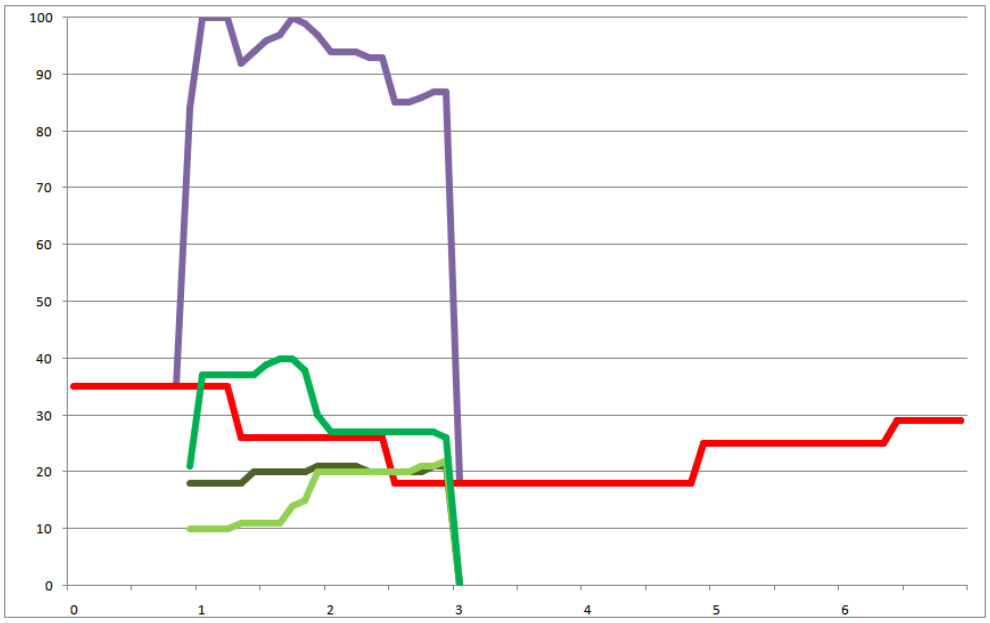

Figure 5.3: An example of the assignment of cores with a scenario that saturates all the available resources.

In the third example, in Figure 5.3, it is shown a scenario in which it is already active a process with a slow dynamic. Then, at the same time, three applications with fast dynamic start. The effect of this execution scheme is the saturation of all the resources for a few milliseconds, followed by the reduction of the resources assigned to the slow process and the adaptation of the three fast ones to leave some resource available for an eventual new process. The assignment of the execution units to the three fast processes is not equal when they start because of the order in which they are scheduled to start. The first of them sees more idle units and it will fill a part of them. Then, the second and the third will see less available resources. By the way, after few seconds, the assignments stabilize themselves to close values. When the three fast processes end, at the third second of the simulation, the slow one progressively increments its parallelization level, being the unique process in execution.

Finally, in Figure 5.4, it is possible to see the last example, that is a simulation of a realistic situation in which the assignment of cores to not-profiled

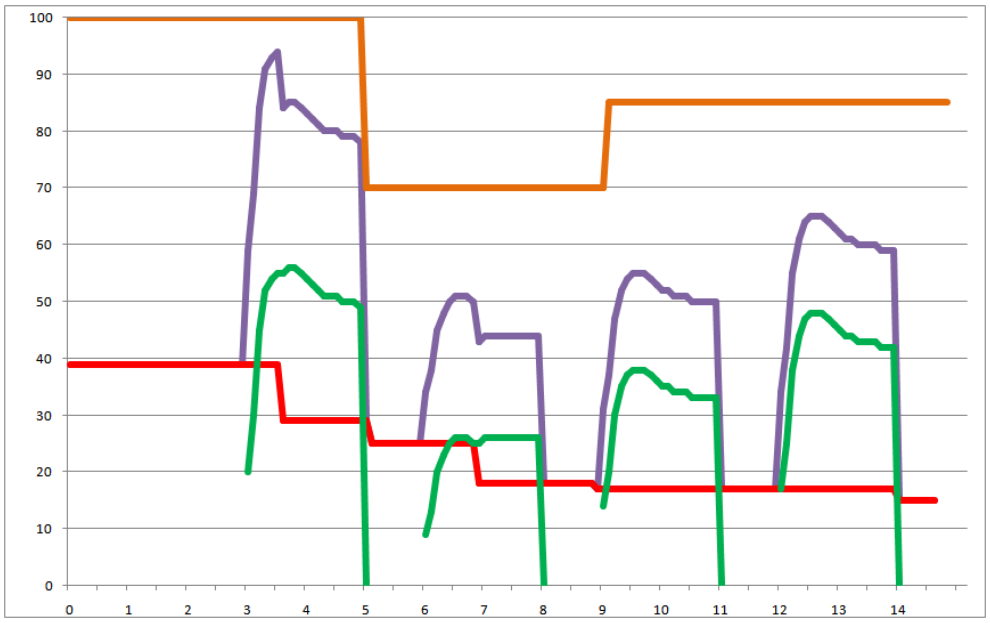

Figure 5.4: An example of the assignment of cores when the number of cores reserved for not-profiled applications varies.

applications needs to be adapted to a change in the number of the reserved cores. The variation of the reserved cores is determined by the operating system and can be due to various causes:

- the guarantee-throughput applications are saturating their reserved units and, to guarantee the QoS, more cores are reserved to them;
- some execution unit fails and does not work, so they are excluded from the available cores;
- for overheating problems, some execution units may be turned off and, later, reactivated;
- a legacy software, not handled by the runtime manager, starts to execute and the cores assigned to it are excluded from those reserved to the not-profiled applications.

The simulation is realized with the composition of a slow dynamic application already running (the red line in the graph), and a fast dynamic application

that executes four bursts (the green line). As usual the violet line represents the sum of the assigned cores and, in this graph, the orange line shows the cores reserved to the applications. In this simulation, at the fifth second, the number of reserved cores drops to 70 and it is possible to see that the assignment for the second burst is adapted to this new scenario, with a reduction of 25 cores. At the ninth second the reserved units are incremented to 85 and the assignment is, again, modified due to the possibility of using more cores. It is possible to notice that the adaptation is gradual when it is possible to increase the assignment, to avoid sudden changes.

5.2 Overhead Analysis

To evaluate the system it is fundamental to perform an analysis that measures the overhead caused by the runtime manager methods. The addition of this functionality to a system must not degrade considerably the overall performance. The minimization of the overhead is an important factor for a library to be adopted in real systems. Depending on the scenarios, a library with high overhead may be discarded, no matter how well it performs or how many new functionalities it adds.

It has been realized two analysis: a measurement of the execution time of the two principal methods of the library and an evaluation of the overhead and of the performance in the execution of a benchmark composed of 16 applications. The analysis has been conducted on a platform constituted of four AMD Opteron 4-cores processors, for a total of 16 physical cores. The configuration of the platform can be found in Table 5.1. The system is equipped with local L1 and L2 caches, shared L3 caches between each processor and 24GB of memory. The platform has a Non-Uniform Memory Access (NUMA) as the memory architecture, where the memory access time depends on the memory location relative to a core.

Number of Sockets	4
Processor	AMD Opteron 8378 @ 2.4 Ghz
Number of Cores	16
L1 Cache Configuration	128KB x 16
L2 Cache Configuration	512KB x 16
L3 Cache Configuration	6MB x 4
Memory Configuration	6GB x 4

Table 5.1: The configuration of the experimenting platform.

5.2.1 Execution Time of the Principal Methods

In this analysis, the two principal methods of the library have been profiled. These methods are `start_section` and `end_section`. They are executed every time a parallel section is starting or ending and they perform the invocation of all the other functions of the library, including for example the methods that handle the parallelization policies. The methods have been executed over one hundred times on the experimenting platform and it is possible to see the average execution times in Table 5.2.

	`start_section`		`end_section`
Not Profiled	First	300us	2us
	Normal	60-120us	
	No Update	2us	
Profiled	First	1250us	4us
	Normal	70-135us	
Best Effort	First	1250us	4us
	Normal	70-130us	

Table 5.2: The measurement of the execution time of two functions, performed on the experimenting platform.

The column that contains the results of the `start_section` method measurement, is divided into several parts. This is due to the different execution behavior of the runtime manager. At the first execution all the data structures are initialized, and in the case of the profiled and best-effort applications, the operating points are loaded from a file and, therefore, the execution time is increased. The `Normal` row reports the result for all the subsequent executions. Because of the non-uniform access to the memory, the execution time may assume two average values, that are reported in the table, based on the code location in memory. The not-profiled applications have a third modality of execution: the maintenance of the policy to avoid too many calculations and to reduce the overhead. As it is possible to see in the table, this modality adds an overhead that, on average, is 2 microsecond. The `end_section` method, on the contrary, has a unique behavior for all the executions. As a final remark, we can consider this library efficient: the overhead of the most used methods is in the order of one hundred microseconds, with the possibility of an execution time that is at most 2 microsecond for particular situations.

5.2.2 Evaluation of the Overhead over a Benchmark

In this analysis the objective is the measurement and comparison of the execution times of a benchmark, composed of sixteen applications, to show the differences between when they are executed without the *Runtime Management System* and when they are considered not-profiled applications and profiled applications. The set of applications used for this analysis is contained in the OpenMP Source Code Repository (*OmpSCR*) version 2.0 [26]. It is a collection of various programs that are all parallelized with the OpenMP directives.
In Table 5.3 it is possible to see the applications used during this analysis, the command line parameters and the number of parallel sections executed at runtime.

The benchmark has been compiled with GCC version 4.4.1. Considering

Name	Command Line Parameters	Parallel Sections
FFT	4096	8387808
FFT6	4096 10	61
Jacobi01	-test	2
Jacobi02	-test	1
loopA.badSolution	1000000 10000	10000
loopA.solution1	200000 10000	20000
loopA.solution2	1000000 10000	1
loopA.solution3	200000 10000	10000
loopB.badSolution1	1000000 10000	10000
loopB.badSolution2	1000000 10000	10000
loopB.pipeSolution	1000000 10000	1
Lu	5000	4999
Mandel	50000	1
Molecular Dynamic	-test	21
Pi	1000000000	1
Qsort	2000000	17808180

Table 5.3: The execution parameters and the number of parallel sections of the benchmark applications.

that the `libGOMP` library parallelize each section over all the cores of the system, the `librtmgmt` library has been modified to assign always an equal number of cores. Practically, the runtime manager executes all its routines but, in the end, the level of parallelization is forced to be equal to the number of the cores. In this way a difference in the execution time cannot be associated to a different parallelization of the application but only to the different methods used to evaluate how many cores should be assigned.
In the first execution it has been used the original version of the compiler and of the applications. For the second execution, the applications have to be executed as not-profiled using the *Runtime Management System*. Therefore

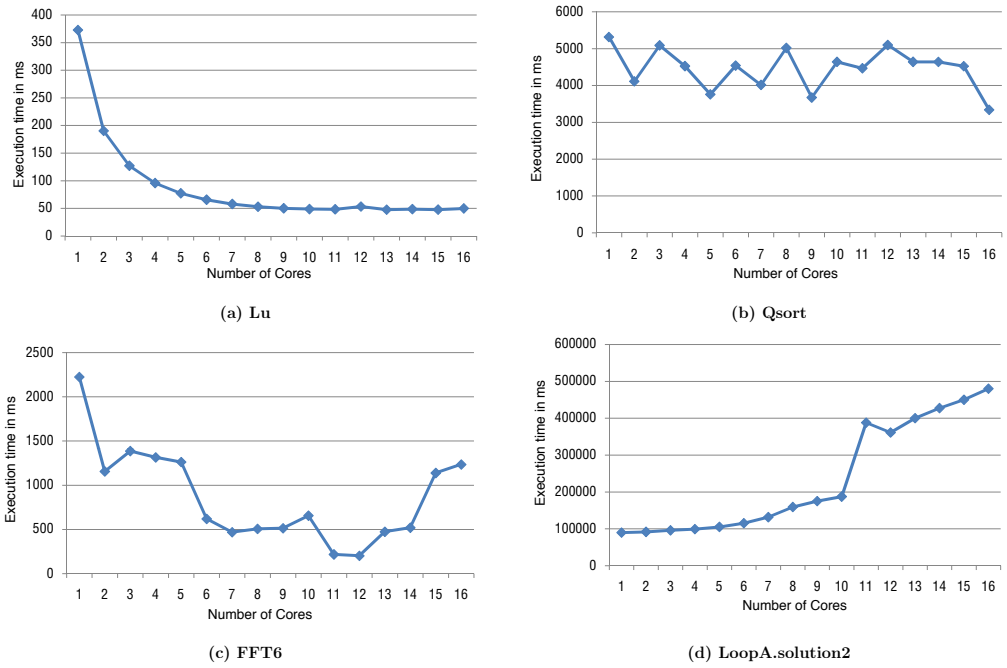

Figure 5.5: The result of the profiling of four benchmark applications.

it has been used the modified version of the compiler but still the original version of the applications. As it was said in Section 3.1, it is not necessary to modify the sources to execute a not-profiled application with the support of the runtime management library. During this execution, the parameter `NP_FILTER_SEC` has been set to 0.2 seconds. For the third execution, the goal was to measure the execution time of profiled applications. The first operation has been the generation of one file for each application with empty operating points. Then it has been done the modification to the sources in order to let the runtime manager know where to find the file with the operating points. At this point it was necessary to profile the applications, to have valid operating points and, after that, everything was done to complete the analysis.

The profiling phase has shown that not every application benefits from being executed by all the available cores. In Figure 5.5 it is possible to see four sets of profiled operating points. In the graphs, it is illustrated how

the execution time varies when the application is parallelized over different number of cores. In Figure 5.5(a) it is represented the profiling of the *Lu* application. This is an intuitive situation, in which the execution time decreases as the number of used cores increases. The Figure 5.5(b) is an example of an application that has a similar behavior with almost all the parallelization levels. The Figure 5.5(c) shows a different dynamic from the *FFT6* benchmark: the minimum execution time is associated to eleven-twelve cores, and not to sixteen. This means that when using all the cores, there is a reduction of the performance. The *LoopA.solution2* application (Figure 5.5(d)) has a completely counterintuitive behavior: the execution time increases as the number of cores increases and the best performance is obtained when the application executes on a single core and, therefore, it is not parallelized at all.
The simulations have been conducted five times for each execution type and, after the removal of the outliers, it is considered the median of them as the result. The comparison of the results is shown in Table 5.4 and in Figure 5.6.

Considering the sum of all the execution times, it is possible to see that the Not-Profiled policies add an overhead of just 1.77%, while the Profiled ones add less than 16%. By looking at the overhead of the single applications, it is shown that for most of the applications, the Not-Profiled version adds less than 1%, and for the Profiled version the overhead is less than 5%. It is possible to notice some negative values, that is mostly due to the variance of the simulation when the execution time is small (see *Mandel*, *Molecular Dynamic* and *Pi*) or to the presence of nested sections, that are not evaluated by the runtime manager if the nesting level is bigger than two, considering that the parallelization choice has been already taken in the parent sections (*FFT6*). The biggest overhead can be found in the applications *FFT* and *Qsort* that, as it is shown in Table 5.3, execute a very large number of sections within short execution times generating a great overhead, especially for the profiled applications because the policies are more computational intensive.

Name	Normal Execution Time	Not Profiled	Profiled
FFT	5.6499	+8.78 %	+126.97 %
FFT6	12.9068	+1.05 %	-17.85 %
Jacobi01	0.2583	+3.45 %	+0.53 %
Jacobi02	0.3022	+0.27 %	-0.20 %
loopA.badSolution	298.2916	+0.86 %	+0.68 %
loopA.solution1	129.3920	+0.66 %	+1.97 %
loopA.solution2	298.4712	+0.22 %	+48.74 %
loopA.solution3	17.4466	+0.41 %	+4.80 %
loopB.badSolution1	347.0923	+0.26 %	-13.02 %
loopB.badSolution2	299.9623	+0.34 %	+0.42 %
loopB.pipeSolution	323.2340	+7.58 %	+41.66 %
Lu	77.1879	+0.17 %	+15.23 %
Mandel	1.1248	+2.56 %	-4.12 %
Molecular Dynamic	5.8968	+1.41 %	-0.34 %
Pi	1.4114	-0.80 %	-1.17 %
Qsort	12.5801	+8.19 %	+251.55 %
Total	1831.2085	+1.77 %	+15.83 %

Table 5.4: The results of the overhead measurement over a set of benchmark applications.

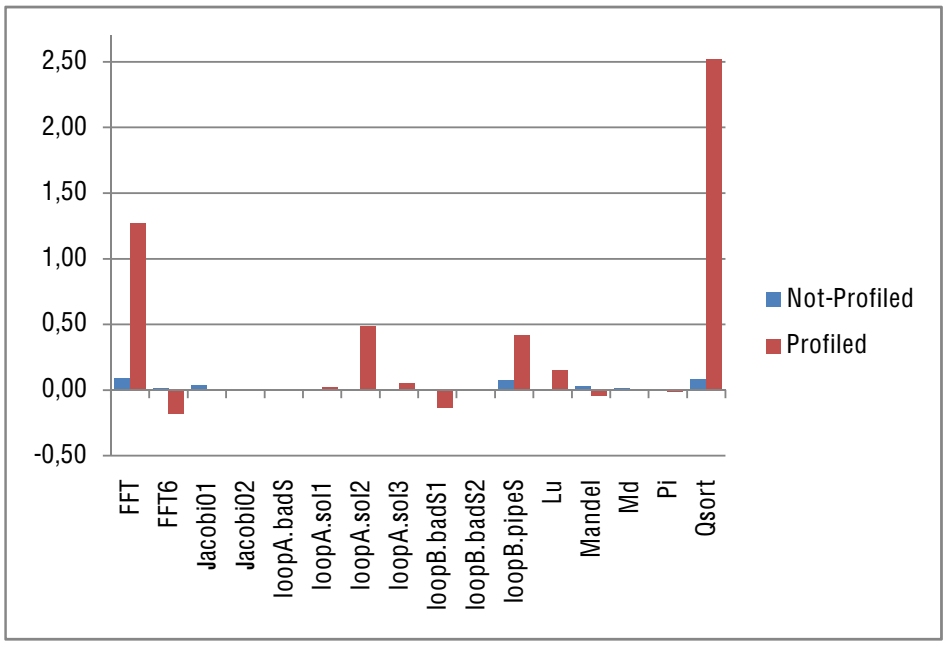

Figure 5.6: The performance variation for profiled and not-profiled applications of a benchmark executed with the same parallelization level.

The profiled and not-profiled application have also been executed with the parallelization policies described in Section 3.7. In this way it is possible to evaluate if the policies allow to achieve better performance on the execution of single applications. We expect a general worsening of the performance of not-profiled applications, because the policy allocates about half of the idle execution units for a single application, to leave the possibility of execution to new processes. For the profiled applications we expect, instead, a performance improvement due to the profiling that represents additional knowledge for the runtime manager. The result are shown in Table 5.5 and Figure 5.7.

These simulations shows that the parallelization policies are able to optimize the execution of single applications when they are profiled. The not-profiled applications, instead, have worst performance. It is interesting to see the result of the simulation for the *LoopA.solution2* benchmark. As it

Name	Normal Execution Time	Not Profiled	Profiled
FFT	5.6499	+19.48 %	+115.16 %
FFT6	12.9068	+67.81 %	-15.96 %
Jacobi01	0.2583	+5.20 %	-29.40 %
Jacobi02	0.3022	-28.83 %	-35.83 %
loopA.badSolution	298.2916	+4.97 %	+5.76 %
loopA.solution1	129.3920	+11.51 %	+12.89 %
loopA.solution2	298.4712	+8.82 %	-70.24 %
loopA.solution3	17.4466	-0.16 %	+3.57 %
loopB.badSolution1	347.0923	-8.47 %	-7.29 %
loopB.badSolution2	299.9623	+3.36 %	+8.52 %
loopB.pipeSolution	323.2340	-53.16 %	+54.80 %
Lu	77.1879	+8.32 %	+9.70 %
Mandel	1.1248	+160.46 %	-4.00 %
Molecular Dynamic	5.8968	+115.60 %	+135.14 %
Pi	1.4114	+159.90 %	-1.46 %
Qsort	12.5801	+8.87 %	+253.12 %
Total	1831.2085	-5.84 %	-16.41 %

Table 5.5: The results of the execution time measurement when a single benchmark applications is executed with the parallelization policies described in this thesis.

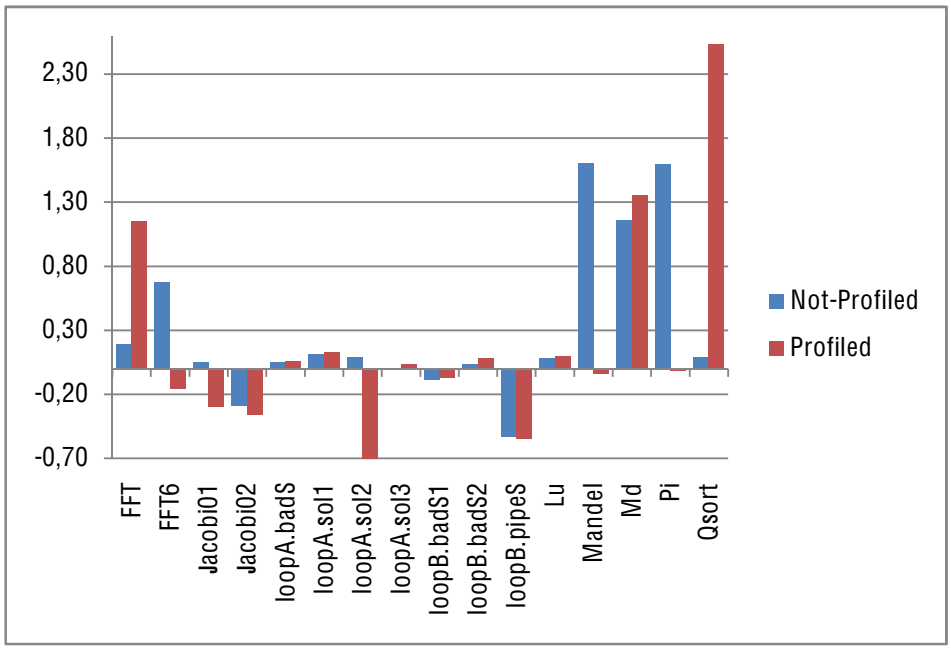

Figure 5.7: The performance variation for profiled and not-profiled applications of a benchmark executed with the parallelization policies described in this thesis.

has been shown in Figure 5.5, it performs better when executed over few cores: this is an information used by the runtime manager and not known by *libGOMP* and the result is a -70% in the execution time.

It has also been conducted another set of simulations, this time with two applications executing concurrently and, again, leaving the runtime manager to decide the parallelization level of the applications. During the execution of the benchmark without the use of the runtime manager, *libGOMP* will parallelize each of the applications over sixteen cores, creating in this way thirty-two threads. Because of the multi-task capability of the experimenting platform and, therefore, the presence of a preemptive scheduler, it is possible to consider the architecture as constituted of thirty-two equivalent units. For this reason, to obtain correct results, the runtime manager has been configured to work on a system with thirty-two cores. The results of this simulation are shown in Table 5.6 and in Figure 5.8.

Name	Normal Execution Time	Not Profiled	Profiled
FFT	6.3035	+0.20 %	+177.08 %
FFT6	13.0591	+67.66 %	-10.62 %
Jacobi01	0.4113	-37.84 %	-43.17 %
Jacobi02	0.4045	-48.92 %	-25.77 %
loopA.badSolution	428.8755	-20.26 %	-19.33 %
loopA.solution1	420.4414	-63.76 %	-40.90 %
loopA.solution2	473.3049	-55.31 %	-81.22 %
loopA.solution3	314.5117	-94.32 %	-94.27 %
loopB.badSolution1	466.5492	-11.62 %	-7.22 %
loopB.badSolution2	615.6112	-32.10 %	-27.03 %
loopB.pipeSolution	687.9206	-56.90 %	-59.09 %
Lu	166.4205	-35.35 %	-45.60 %
Mandel	2.2461	-2.45 %	-21.71 %
Molecular Dynamic	7.1144	-14.89 %	-24.86 %
Pi	2.1943	+27.22 %	-3.79 %
Qsort	13.0527	+2.68 %	+236.80 %
Total	3287.4687	-43.12 %	-42.07 %

Table 5.6: The results of the execution time measurement when a pair of benchmark applications is concurrently executed with the parallelization policies described in this thesis.

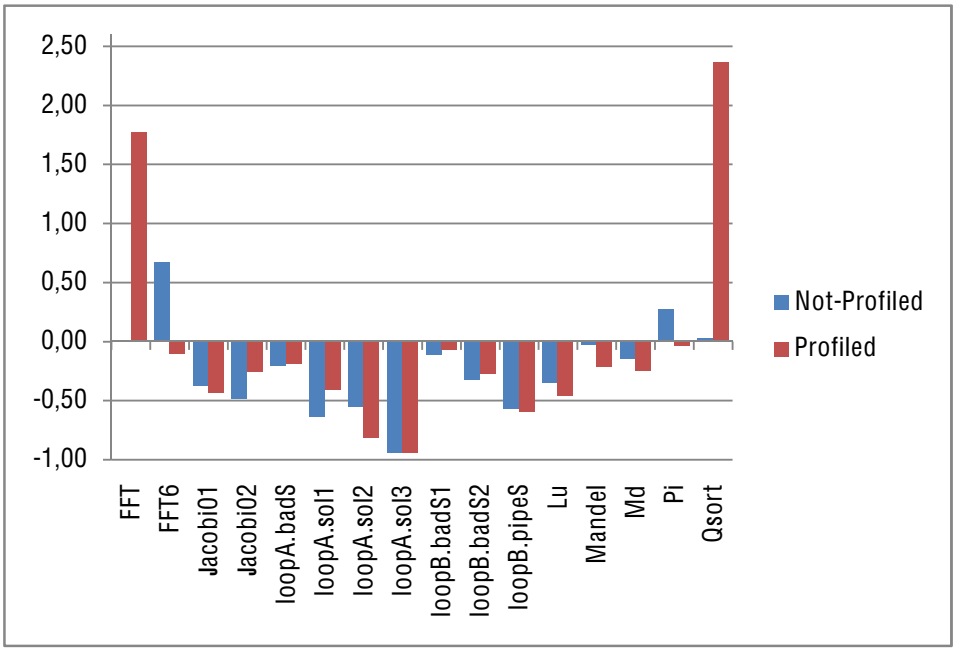

Figure 5.8: The performance variation for profiled and not-profiled applications of a benchmark executed pairwise with the parallelization policies described in this thesis.

In the *Total* row of the table, it is possible to see that both the profiled and not-profiled applications reduce the execution time of more than 40%. This is due to the fact that *libGOMP* parallelizes all the applications over all the cores of the system. The generation of 32 processes causes an additional overhead due to the continuous context switch. The runtime manager, instead, considers the current occupation of the execution units and the execution time of each parallelization level, thus avoiding the generation of too many processes. Almost all the applications obtain a speedup in the execution time, with the exclusion of *FFT* and *Qsort*, for which it has already been shown that they are slowed down by the great number of parallel sections with respect to the execution time.

As a final summary, the *Runtime Management System* shows low overhead both when it is used with profiled applications and with not-profiled ones. Some benchmarks worsen their performance, particularly if they are handled as profiled applications, if they tries to parallelize sections with few instructions. In the other cases, the a-priori knowledge of the different parallelization levels, provided by the operating points, allows the profiled programs to perform better. The not-profiled applications shows a general worsening of the performance when executed individually, due to the conservative policy that does not assign all the available execution units to a single application. It is, instead, proved that with just two applications executing concurrently, the performance increases consistently for both profiled and not-profiled applications, because of the adaptive policies that avoid the saturation of the cores and the context switch overhead.

Chapter 6

Conclusions and Future Works

Nowadays we are assisting to a change in the way the architectures of the processors are being implemented. The need to reduce the power consumption and the heating has led to the development of processors with an increasing number of cores. The *Many-Core Architectures* are the next big turning-point in the development of architectures and we have to be ready to make the most of them by exploiting the advantages that they can offer.

OpenMP follows this purpose, helping the programmers in the difficult task of the parallelization of the applications. However, the assignment of the parallelization level of an application does not take care of the current status of utilization of the system, and basically every portion of parallel code is subdivided over all the cores of the architecture. Although this can be a rational choice with architectures constituted of two, four or eight cores, it is hard to say the same when there are hundreds or thousands of available execution units. This consideration explains the necessity to develop a runtime manager, able to handle the parallelization over many cores.

In this thesis I have proposed a runtime manager that is capable of supporting both profiled and not profiled applications, and that can be integrated and improve the parallelization policy of the GNU OpenMP library and that is, also, configurable through a series of policy parameters. Therefore, it can be adapted to a wide range of applicative scenarios just by tuning it. Con-

sidering the difficulty of a trial and guess assignment of the parameter to find the optimal configuration, I have also implemented a tool that is capable of simulating and evaluating the behavior of the runtime manager with a particular configuration when it handles a set of applications. By integrating it with an already existent *Design Space Exploration* tool, capable of testing the runtime manager over different configurations and plotting the results of all the evaluation, I have obtained a *Design-time Configuration Framework*. Following the results given by this framework it is possible to easily find the configuration that minimizes the QoS requirements violations and the queued threads while maximizing the throughput of all the applications. Experimental results have been conducted on a real *Many-Core* system, to analyze the difference in system throughput and individual application performance when using the *Runtime Management System*. They show that it is introduced low overhead and that it is obtained a consistent performance improvement in the execution of multiple concurrent running applications.

The work done in this thesis can be the starting point for future developments. The first thing to consider more carefully is the power consumption. The runtime manager already supports the possibility to add an indicator of the dissipated power for every operating point, but this value is not used in the evaluation of the parallelization policies. With this improvement, the runtime manager would acquire energy saving capability. Another possible extension can be made over the interaction with the Operating System. It can be very interesting to improve the data exchange between the *Runtime Management System* and the OS, in order to obtain new possibilities for the optimization of the execution of the applications. An example of data that could be exchanged is the processor affinity. Also in this case, the runtime manager already supports the specification of a value that indicates the affinity between the parallel threads, but it is not used by the scheduler to improve the assignment of the cores. First, it would be necessary to study the different kind of affinity that the threads may have and, then, to evaluate the advantages of the different allocations. It could be also interesting

to study new policies for the applications to satisfy different necessities. For example, it may be considered a priority the minimization of the changes in the parallelization level, in order to reduce the overhead caused by the allocation/deallocation of the cores, by the possible thread migration and by the cold start latency of the caches.

Appendix A

OpenMP Extensions

In order to better understand how to use the extensions to the OpenMP specification 3.0 and the API for the profiled applications developed in this thesis, it is now shown an example of a benchmark application where the modifications needed to make the application profiled, are highlighted. This example can be very helpful for a programmer that wants to use the *Runtime Management System*, but it is also a way to show how few changes are needed to have a profiled application.

In Figure A.1 it is shown the code of the Pi benchmark, where the colored lines are related to the *Runtime Management System*. First of all, we start from the center of the code at line 16: in this line it is possible to see the OpenMP extensions described in Section 3.3. These are two clauses used to identify the section with a number and to specify a desired time, in milliseconds, in which the parallel section should be completed. The identification number can be any integer greater than zero, while the completion time can be also specified as a variable, initialized before the declaration of the section, that obviously must be always greater than zero. A section with a completion time not specified or with a value equal to zero will be executed as a best-effort application.

The orange lines of code (1, 3 and 4) are those needed to inform the runtime manager that the current application is profiled. The first line is the

```
1  #include "rtmgmt.h"
2  int main(int argc, char *argv[]) {
3    char* ops = "c_pi.ops";
4    rtmgmt_load_op(ops);
5    int prof[]={1};
6    rtmgmt_profile(prof,1,5);
7
8    double PI25DT = 3.14159265358979323846264 3;
9    double local, w, total_time, pi;
10   long i, N;    /* Precision */
11   N = OSCR_getarg_int(1);
12   w = 1.0 / N;
13   pi = 0.0;
14
15   #pragma omp parallel for default(shared)
16           private(i, local) reduction(+:pi)
17           section_id(1) completion_time(3000)
18   for(i = 0; i < N; i++) {
19     local = (i + 0.5) * w;
20     pi += 4.0 / (1.0 + local * local);
21   }
22   pi *= w;
23   rtmgmt_end_profile();
24   return 0;
25 }
```

Figure A.1: An example of a profiled application with highlighted modifications.

inclusion of the API methods definition, the second is the initialization of a string with the path to the file that contains the operating points, while the third is the call to the function of the runtime manager that will fetch the points from the specified file.

The violet lines (5, 6 and 23) are used to profile the application, and can be utilized only if the red and orange lines have been inserted, because it is necessary to identify all the sections and to load the operating points. These lines should be inserted only during the profiling phase, and then removed or commented. First, it is necessary to specify an array with the identification numbers of the sections that should be profiled. Then it is called the function `rtmgmt_profile`, with three parameters: the array defined before, the dimension of the array, or in other words the number of sections to be profiled (in this case 1), and the number of times that every operating point should be profiled (in this case 5). After this number of times the application will be terminated and all the results will be saved. If the application is not long enough to execute the specified number of times every operating point, it is necessary to insert the command `rtmgmt_end_profile` just before the end of the application. The method will save all the collected data, allowing a complete profiling of the application with few complete executions.

Bibliography

[1] OpenMP. The openmp api specification for parallel programming. *http://www.openmp.org/*.

[2] Jianian Yan, Jiangzhou He, Wentao Han, Wenguang Chen, and Weimin Zheng. How openmp applications get more benefit from many-core era. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 83–95. Springer Berlin / Heidelberg, 2010.

[3] Toshihiro Hanawa, Mitsuhisa Sato, Jinpil Lee, Takayuki Imada, Hideaki Kimura, and Taisuke Boku. Evaluation of multicore processors for embedded systems by parallel benchmark program using openmp. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 15–27, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384, New York, NY, USA, 2008. ACM.

[5] S. Illner, A. Pohl, H. Krumm, I. Luck, D. Manka, and T. Sparenberg. Automated runtime management of embedded service systems based on design-time modeling and model transformation. In *Industrial Informatics, 2005. INDIN '05*, pages 134 – 139, aug. 2005.

[6] Ch. Ykman-Couvreur, V. Nollet, Fr. Catthoor, and H. Corporaal. Fast multi-dimension multi-choice knapsack heuristic for mp-soc run-time management. In *System-on-Chip, 2006. International Symposium on*, pages 1 –4, nov. 2006.

[7] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Design-time application mapping and platform exploration for mp-soc customised run-time management. *Computers Digital Techniques, IET*, 1(2):120 –128, mar. 2007.

[8] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 917 –922, jul. 2009.

[9] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. A design space exploration methodology supporting run-time resource management for multi-processor systems-on-chip. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 21 –28, jul. 2009.

[10] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 196 –201, mar. 2010.

[11] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 5–5, Berkeley, CA, USA, 2000.

[12] Matthew Curtis-Maury, Xiaoning Ding, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. An evaluation of openmp on current and

emerging multithreaded/multicore processors. In *First International Workshop on OpenMP*, 2005.

[13] Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An efficient openmp runtime system for hierarchical architectures. In Barbara Chapman, Weiming Zheng, Guang Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 161–172. Springer Berlin / Heidelberg, 2008.

[14] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: An openmp runtime perspective. In Matthias Muller, Bronis de Supinski, and Barbara Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 79–92. Springer Berlin / Heidelberg, 2009.

[15] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling dynamic openmp applications over multicore architectures. In Rudolf Eigenmann and Bronis de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 170–180. Springer Berlin / Heidelberg, 2010.

[16] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and Perg Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 2(3):37, aug. 1994.

[17] R. Anane. Autonomic behaviour in qos management. In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, pages 57 –57, jun. 2007.

[18] Samuel Kounev, Ramon Nou, and Jordi Torres. Autonomic qos-aware resource management in grid computing using online performance models. In *ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[19] Xuan Chen and Shun Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 907 –912, dec. 2009.

[20] Alex Scherer, Thomas Gross, and Willy Zwaenepoel. Adaptive parallelism for openmp task parallel programs.

[21] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multicores: a machine learning based approach. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, New York, NY, USA, 2009. ACM.

[22] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC '10: Proceeding of the 7th international conference on Autonomic computing*, pages 79–88, New York, NY, USA, 2010. ACM.

[23] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[24] Multicube Explorer. Multi-objective design space exploration of multiprocessor-soc architectures for embedded multimedia applications. *http://home.dei.polimi.it/zaccaria/multicube_explorer_v1/Home.html.*

[25] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, and K.C. Sevcik. Processor saving scheduling policies for multiprocessor systems. *Computers, IEEE Transactions on*, 47(2):178 –189, feb. 1998.

[26] OmpSCR. Openmp source code repository. *http://ompscr.sourceforge.net*.