UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SELF-OPTIMIZING SKELETON EXECUTION USING EVENTS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

GUSTAVO ADOLFO PABÓN SÁNCHEZ

PROFESOR GUÍA:
JOSE MIGUEL PIQUER GARDNER

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULE
JOHAN FABRY
FERNANDO RANNOU FUENTES

SANTIAGO DE CHILE
2015

Resumen

Esta tesis propone una forma novedosa para introducir características autonómicas de auto-configuración y auto-optimización a Patrones de Paralelismo (o *Algorithmic Skeletons* en inglés) usando técnicas de la Programación Dirigida por Eventos (o *EDP* por sus siglas en inglés).

Gracias al uso de la programación dirigida por eventos, la solución propuesta en esta tesis presenta las siguientes contribuciones a las soluciones actuales:

- No está relacionada a una arquitectura de aplicaciones en particular, por ejemplo la arquitectura de componentes. De esta forma, presenta una solución con un mayor alcance y es independiente de la arquitectura usada para la implementación de los patrones de paralelismo.
- Provee un mayor nivel de adaptabilidad por permitir la introducción de cambios estructurales en tiempo de ejecución. A diferencia de otras soluciones que solo permiten la introducción de este tipo de cambios durante la compilación.
- Los estimados de trabajo futuro pueden ser calculados en tiempo de ejecución y no hay dependencia a estimados pre-calculados en ejecuciones anteriores.

Las características autonómicas presentadas en esta tesis se enfocan principalmente en garantizar un tiempo de ejecución a un patron de paralelismo por medio de la optimización de la cantidad de hilos de ejecución usados. Las calidades de servicio (*QoS* por sus siglas en inglés) usadas para este fin son: (1) tiempo de ejecución percibido por el usuario y (2) nivel de paralelismo.

Otras contribuciones de esta tesis son:

- Diseño e implementación de una novedosa solución a la separación de asuntos en los patrones de paralelismo usando técnicas de la programación dirigida por eventos. Esta solución novedosa permite la introducción de asuntos no-funcionales a los patrones de paralelismo sin disminuir sus abstracciones de alto nivel.
- Evaluación de diferentes estrategias de estimación de trabajo futuro con el fin de realizar recomendaciones relacionadas a responder a la pregunta ¿Qué estrategia produce mejores estimados bajo qué circunstancias?

Abstract

This thesis presents a novel way to introduce self-configuration and self-optimization autonomic characteristics to Algorithmic Skeletons using Event-Driven Programming (EDP) techniques.

Due to the use of event driven programming, the approach proposed in this thesis contributes to the current solutions as follows:

- It is not related to a specific application architecture, like the component architecture. Thus, it has a broader scope, and is independent of the architecture used to implement skeletons.
- It allows the introduction of structural changes during execution time. Therefore, it provides a higher level of adaptability compared with other solutions that only allows to introduce structural changes during compilation.
- The estimates of expected future work can be calculated on-the-fly, and it is not limited to pre-calculated estimates.

This thesis focuses mainly on guaranteeing a given execution time for a skeleton, by optimizing the number of threads allocated to execute an skeleton. The QoSs (Quality of Services) autonomically managed to be self-configured and self-optimized are: (1) Execution Wall Clock Time, and (2) Level of Parallelism.

Other contributions are:

- Design and implementation of a novel skeleton's separation-of-concerns based on event driven programing. This novel approach allows the introduction of non-functional concerns to skeletons without lowering its higher-level programming.
- Evaluation of different estimation strategies of future work was done. As result, a recommendation is presented to answer the question: Which strategy produces better estimates under which circumstances?.

To Mayi, Sofi, and Migue

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Prof. José M. Piquer, Dr. Mario Leyton and Prof. Ludovic Henrio who have been my mentors, for their continuous support, their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped throughout the research and writing of this thesis. I can not imagine having had better mentors for my M.Sc study.

My sincere thanks also go to NIC Chile Research Labs for its financial support and the scholarship granted to me during my M.Sc study. It was a pleasure to have worked for such great institution.

I thank my previous employer, IBM, and my current employer, SAB Miller, for its support and understanding during my studies, and for all the time that I have had to take from them to make this thesis a reality.

Last but not the least, I would like to thank my family: my wife Maria Cecilia, my daughter Sofia and my son Miguel, for their love and continuous support, and for understanding that I had to dedicate some of my time off to my studies and this thesis. Thank you very much.

GUSTAVO ADOLFO PABÓN SÁNCHEZ

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large-scale parallel environments allow the resolution of large-scale problems. However, as stated by Patterson in [32], parallel software development is hard, and currently, we are facing an increasing challenge due to the increasing number of cores available for a single computation.

According to Gartner [33], IT operations management costs are 36% of the total IT operation budget. As a solution, IBM in 2001, introduced the concept of Autonomic Computing (AC) [24]. It refers to the self-managing (self-configuration, self-optimization, self-healing, and self-protecting) characteristics of computing resources. In autonomic computing, each entity is able to adapt itself to changes in the runtime environment, or in the quality of service desired. The vision of autonomic computing is based on the idea of self-governing systems to reduce its management costs. These systems can manage themselves given high-level objectives from an administrator [26].

On another note, Algorithmic Skeletons [21] (skeletons for short) is a high-level parallel programming model introduced by Cole [10]. Skeletons take advantage of recurrent parallel programming patterns to hide the complexity of parallel and distributed applications. Lately, the use of skeletons has risen because of the increasing popularity of MapReduce pattern [8] for data-intensive processing.

This thesis presents a novel approach for introducing autonomic characteristics to skeletons based on event driven programming.

1.1 Problem

The introduction of autonomic characteristics to skeletons is an active research topic in the parallel and distributed computing area [3, 12, 22]. The benefits of this type of technology are:

1. reducing management costs of parallel systems, and
2. hiding the complexity of parallel programming to the developer.

As presented in more detail in chapter 2, State of the Art, the current solutions found in the literature suffer from at least one of the following issues:

- They are related to a specific application architecture, like the component architecture.
- Structural information is introduced only during compilation.
- The prediction mechanism at execution time uses only pre-calculated estimates to construct an execution plan.

1.2 Objectives and Contributions

The main objective of this thesis is the design and implementation of a novel approach to the introduction of autonomic characteristics to skeletons using event driven programming.

Due to the use of event driven programming, the approach proposed in this thesis contributes to the current solutions as follows:

- It is not related to a specific application architecture, like the component architecture. Thus, it has a broader scope, and is independent of the architecture used to implement skeletons.
- It allows for the introduction of structural changes during execution time. It, therefore provides a higher level of adaptability compared with other solutions that only allows to introduce structural changes during compilation.
- Estimates of expected future work can be calculated on-the-fly, and it is not limited to pre-calculated estimates.

Chapter 3 presents in detail why event driven programming makes these contributions possible.

To show the feasibility of the approach, self-configuring and self-optimizing characteristics were implemented on the Skandium framework. Skandium will be introduced in more detail in section 2.5.

This thesis focuses on guaranteeing a given execution time for a skeleton by optimizing the number of threads allocated. The QoSs (Quality of Services) autonomically managed to be self-configured and self-optimized are: (1) Execution Wall Clock Time, and (2) Level of Parallelism.

Other contributions are:

- Design and implementation of a novel skeleton's separation-of-concerns based on event driven programing. This novel approach allows the introduction of non-functional concerns to skeletons without lowering its higher-level abstractions. See chapter 3.

- Evaluation of different estimation strategies of future work was done. As result, a recommendation is presented to answer the question: Which strategy produces better estimates under which circumstances?. See chapter 5.

1.3 Overview

This document is organized as follows:

- Chapter 2 provides relevant background and a state of the art of the current approaches found in the literature to introduce autonomic characteristics to skeletons. The chapter begins by introducing Algorithmic Skeletons and Autonomic Computing. Then, the chapter provides a survey of current autonomic skeleton approaches. Then, skeleton's separation-of-concerns related work is presented. The chapter finishes by introducing Skandium, the skeleton framework used to demonstrate the feasibility of the novel approach proposed in this thesis.

- Chapter 3 presents a novel skeleton's separation-of-concerns based on event driven programing. The chapter includes a discussion about the "Inversion of Control" introduced by skeletons and how it is the main issue to be tackled when introducing separation-of-concerns. The chapter then shows how events provide an elegant solution. The chapter finishes with an example of a simple logger and a visualization-of-the-execution tool implemented using the proposed approach.

- Chapter 4 provides the actual description of the proposed approach to introduce self-configuration and self-optimization characteristics to skeletons using events. The chapter begins presenting the particular Quality of Services (QoSs) to be autonomically managed. Then, the chapter shows how to track the skeleton's execution and how to predict the remaining work. The chapter finishes by showing the viability of the proposed approach by presenting the evaluation of a case study.

- Chapter 5 presents the evaluation result of two estimation strategies to provide a recommendation of which strategy produces better estimates and in which circumstances. The chapter includes the analysis of two statistical strategies for estimating new values of a variable based on its previous values: Weighted Moving Average, and Exponential Moving Average; different parameters of each strategy are included in the analysis. The chapter finishes with the recommended strategy that should be used under certain circumstances.

- Chapter 6 concludes this thesis by providing future research perspectives and summarizes the contributions.

Chapter 2

State of the Art

This chapter provides a relevant background and state of the art of current approaches found in the literature to introduce autonomic characteristics to skeletons. The chapter begins with a general background on Skeletons. Section 2.2 introduces the concept "Autonomic Computing". The following section provides a survey of the current skeleton frameworks that have introduced autonomic characteristics, and demonstrates how this thesis contributes to each one. Finally, this chapter describes Skandium, the skeleton framework used to demonstrate the feasibility of the novel approach proposed in this thesis.

2.1 Algorithmic Skeletons

As stated by Mario Leyton in his Ph.D. Thesis [28], Algorithmic Skeletons (skeletons for short) are a high-level parallel programming model for parallel and distributed computing. Skeletons take advantage of recurrent parallel programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones.

The most outstanding feature of skeletons, which differentiates them from other high-level programming models, is that orchestration and synchronization of the parallel activities is implicitly defined by the skeleton patterns. Programmers do not have to specify the synchronizations between the application's sequential parts. This yields two implications. First, as the communication/data access patterns are known in advance, cost models can be applied to schedule skeletons programs [23]. Second, that algorithmic skeleton programming reduces the number of errors when compared to traditional lower-level parallel programming models (e.g. Threads, MPI).

Algorithmic skeletons were first introduced by Cole in 1989 [10]. Several frameworks have been proposed by different research groups using different techniques based on functional, imperative, custom and object oriented languages. A survey of algorithmic skeleton frameworks can be found

in González-Vélez & Leyton [21].

The following are some well-known skeleton patterns [1].

- ***FARM*** is also known as ***master-slave***. Farm represents task replication where the execution of different tasks in the same farm are replicated and executed in parallel.
- ***PIPE*** represents staged computation. Different tasks can be computed simultaneously on different pipe stages. A pipe can have a variable number of stages, each stage of a pipe may be a nested skeleton pattern. Note that an n-stage pipe can be composed by nesting n-1, 2-stage pipes.
- ***FOR*** represents fixed iteration, where a task is executed a fixed number of times. In some implementations the executions may take place in parallel.
- ***WHILE*** represents conditional iteration, where a given skeleton is executed until a condition is met.
- ***IF*** represents conditional branching, where the execution choice between two skeleton patterns is decided by a condition.
- ***MAP & REDUCE*** represents *split*, *execute*, *merge* computation. A task is divided into sub-tasks, sub-tasks are executed in parallel according to a given skeleton, and finally sub-task's results are merged to produce the original task's result.
- ***D&C*** represents divide and conquer parallelism. A task is recursively sub-divided until a condition is met, then the sub-task is executed and results are merged while the recursion is unwound.
- ***SEQ*** represents sequential execution and it is often used a convenient tool to wrap code as the leaves of the skeleton nesting tree.

Lately, the use of skeletons has risen due to the increasing popularity of the MapReduce pattern [8] for data-intensive processing.

2.2 Autonomic Computing

In 2001 IBM introduced the concept of Autonomic Computing (AC) [24]. It refers to the self-managing characteristics of computing resources. In autonomic computing, each entity is able to

[1] Taken from the "Algorithmic Skeleton´´ article of Wikipedia: http://en.wikipedia.org/wiki/Algorithmic_skeleton

adapt itself to changes in the runtime environment, or in the quality of service desired. The vision of autonomic computing is based on the idea of self-governing systems to reduce its management costs. These systems can manage themselves given high-level objectives from an administrator [26].

According to Horn [24], the current obstacle in keeping the very benefits information technology aims to provide, is complexity. "Dealing with it is the single most important challenge facing the I/T industry" he says. Horn supports it by arguing: "Follow the evolution of computers from single machines to modular systems to personal computers networked with larger machines and an unmistakable pattern emerges: incredible progress in almost every aspect of computing—microprocessor power up by a factor of 10.000, storage capacity by a factor of 45.000, communication speeds by a factor of 1.000.000—but at a price. Along with that growth has come increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions of lines of code. Up until now, we've relied mainly on human intervention and administration to manage this complexity. Unfortunately, we are starting to gunk up the works. Even if we could somehow come up with enough skilled people, the complexity is growing beyond human ability to manage it. Without new approaches, things will only get worse. Paradoxically, to solve the problem -make things simpler for administrators and users of I/T- we need to create more complex systems. By embedding the complexity in the system infrastructure itself -both hardware and software- then automating its management." [24].

"It's time to design and build computing systems capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there." [24].

In his argument, Horn presented eight characteristics that an autonomic system must have.

1. Self-awareness. To be autonomic, a computing system needs to "know itself" and "be aware" of the system-wide components and its relations. A system cannot monitor what it does not know exists, or control specific points if its domain of control remains undefined.

2. Self-configuration. An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions. System configuration or "setup" must occur automatically, just like the human autonomous nervous system.

3. Self-optimization. Horn argues that an autonomic computing system never settles for the status quo. It should always look for ways to optimize its workings. This consistent effort to

optimize itself is the only way a computing system will be able to meet the complex and often conflicting I/T demands of a business, its customers, suppliers and employees.

4. Self-healing. An autonomic computing system must be able to be aware of its malfunctioning and try to heal itself, just like the human immune system. It must be able to discover problems or potential problems, then find an alternative way of using resources or reconfiguring the system to keep functioning smoothly.

5. Self-protection. An autonomic computing system must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

6. Context-awareness. An autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly.

7. Use of open standards. Advances in autonomic computing systems will need a foundation of open standards for system identification, communication and negotiation.

8. Hidden complexity. About complexity, Horn says: this is the ultimate goal of autonomic computing, the marshaling of I/T resources to shrink the gap between the business or personal goals of our customers, and the I/T implementation necessary to achieve those goals-without involving the user in that implementation.

Horn concludes, "Realistically, such systems will be very difficult to build and will require significant exploration of new technologies and innovations. That's why we view this as a grand challenge for the entire I/T industry. We'll need to make progress along two tracks: making individual system components autonomic, and achieving autonomic behavior at the level of global enterprise I/T systems. Autonomic computing represents both this change and the inevitable evolution of I/T automation. This next era of computing will enable progress and abilities we can barely envision today." [24].

This thesis presents a novel approach for introducing self-configuration and self-optimization characteristics to skeletons based on Event-Driven Programming (EDP). EDP allows to improve self- and context- awareness while keeping the complexity hidden from the user of skeletons.

2.3 Autonomic Skeleton's Related work

Autonomic capabilities of skeletons have often been associated to component models [6]. Among them, the ParaPhrase project [22] is probably the most important project in the area. It aims

to produce a new design and implementation process based on adaptable parallel patterns. However, algorithmic skeletons are better adapted to express pure computational application patterns, compared to the structural pattern described by components. Thus the approach proposed in this thesis is better adapted to provide autonomic features for the computational aspects, while the ParaPhrase project gives those possibilities at the level of the application architecture. Also in the domain of components, several efforts have been made to give autonomic features in hierarchical structured component models like Fractal [7] and GCM [5] (Grid Component Model). Those works generally focus on the structure of the autonomic features and also on the application architecture [15, 34]. Again, the approach presented in this document is much more focused on the Computational Aspects, and thus is complementary with these component-oriented approaches.

The ASSIST framework [4] showed the complementarity of the two approaches by providing both structural support and computational support for autonomic Aspects. The approach proposed in this thesis only focuses on the Computational Aspect, and improves the state of the art on the autonomic adaptation of skeletons; consequently, it can be used in a framework like ASSIST to improve the autonomic adaptation capacities of the framework, and finally obtain large-scale complex and structured component applications with efficient autonomic adaptations.

Another important solution in the literature is part of the ASPARA project [20] lead by Murray Cole and Horacio Gónzales-Vélez. This work proposes a methodology to introduce adaptability in skeletons. In ASPARA, structural information is introduced during compilation. Compared to the ASPARA project, the solution proposed in this document allows the introduction of structural information during execution. This produces a higher level of adaptability because it is possible to react faster to mismatch in the quality of service (QoS) desired.

The next related work is the Auto-tuning SkePU [14]. Here the prediction mechanism of execution time uses online pre-calculated estimates to construct an execution plan for any desired configuration with minimal overhead. The proposed solution here does not need pre-calculated estimates, as it calculates estimates at runtime. Again from the dynamic estimation of runtime and the dynamic adaptation of the skeletons, the approach proposed in this thesis is able to react faster to unsatisfactory quality of service.

To summarize, the approach presented in this thesis: to introduce self-configuration and self-optimization characteristics to skeletons, is much more focused on the computational aspects. It allows the introduction of structural information during at execution time, it does not need pre-calculated estimates since it can calculate them at runtime, and thus is complementary with the current approaches to obtain large-scale complex and structured component applications with effi-

cient autonomic adaptations.

2.4 Skeleton's Separation-of-Concerns Related Work

One of the contributions of this thesis is the design and implementation of a novel skeleton's separation-of-concerns based on event driven programing. There are several related work On this topic. This section summarizes the closest ones with respect to the purpose of this thesis.

In [1], Aldinucci et al, introduces muskel, a full Java library providing a skeleton based parallel-and-distributed programming environment. Aldinucci et al propose annotations and AOP, to extend the possibility for users to control some non-functional features in a high-level way. The non-functional concerns discussed include autonomic managers (optimization), portability, security, parallelism exploitation properties, and expandability.

Another Skeleton Framework that incorporates AOP is SBASCO [16]. SBASCO includes the notion of aspect in conjunction with the original paradigms. Aspects are implemented as components and connectors, achieving in this way an homogeneous and clean implementation since skeletons are based on the same model. The proposed approach in this chapter provides an extension based on specific skeletons rather than a tertiary generic abstraction, which conforms a more generic solution independent of the application architecture.

The Join Calculus [18] introduced join-patterns, which are also another important type of parallelism pattern (skeleton). A join-pattern is like a super pipeline to match and join a set of messages available from different message queues, then handles them all simultaneously with one handler. In 2014, [36], Van Ham et al, introduces JEScala, a Join language based on EScala, [19], an advanced event driven OO language. JEScala language explores the synergy of Join-pattern skeleton and EDP just as proposed in this section. One of the key contributions of JEScala is the introduction of asynchronous execution of event handlers. On this thesis, it only supports synchronous execution of event handlers in order to guarantee that the "before" and "after" events are executed on the same muscle thread. Another important difference is the disjunction's consume. Where multiple join-patterns in a disjunction can share an event; when such event is raised it is non-deterministic consumed by one of the join-patterns. The approach presented in this section does not include disjunctions and all listeners registered to an event are executed once the event is raised. The main similarities with JEScala are: event listeners can be added and removed at runtime, event data is not sent to a single destination but to multiple destinations, support to implicit events to create a clear separation of concerns, support of declarative events by the use of Generic Listeners, and support to function operators by the use of listener guards.

2.5 Context: The Skandium Library

To show the feasibility of the approach proposed in this thesis, it was implemented as an extension of Skandium framework [29]. Skandium is a Java based Algorithmic Skeleton library for high-level parallel programming of multi-core architectures. Skandium provides basic nestable parallelism patterns, which can be composed to program more complex applications.

In Skandium, skeletons are provided as a Java Library. The library can nest task and data parallel skeletons according to the following syntax:

$$\begin{aligned}\Delta ::= \quad & seq(f_e)|farm(\Delta)|pipe(\Delta_1, \Delta_2)|while(f_c, \Delta)|if(f_c, \Delta_{true}, \Delta_{false})| \\ & for(n, \Delta)|map(f_s, \Delta, f_m)|fork(f_s, \{\Delta\}, f_m)|d\&c(f_c, f_s, \Delta, f_m)\end{aligned}$$

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern and hidden away from the programmer.

The task-parallel skeletons are:

- *seq* for wrapping execution functions;
- *farm* for task replication;
- *pipe* for staged computation;
- *while*/*for* for iteration;
- *if* for conditional branching.

The data-parallel skeletons are:

- *map* for single instruction multiple data;
- *fork* is like *map* but applies multiple instructions to multiple data;
- *d&c* for divide and conquer.

The nested skeleton pattern (Δ) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. These blocks are called "muscles", as they provide the real (non-parallel) functionality of the application.

In Skandium, muscles come in four flavors:

1. "Execution", $f_e : P \rightarrow R$;
2. "Split", $f_s : P \rightarrow \{R\}$;
3. "Merge", $f_m : \{P\} \rightarrow R$;
4. "Condition", $f_c : P \rightarrow boolean$

where P is the parameter type, R the result type, and $\{X\}$ represents a list of elements of type X. Muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined Δ. The result of a muscle is passed as a parameter to other muscle(s) following dependencies defined by the skeleton program. When no further muscles need to be executed, the final result is delivered.

Listing 2.1 shows an example of a simple Skandium program for counting words; it is a skeleton with the following structure: $map(f_s, f_e, f_m)$. Here `fs` takes as input an `String` and splits it by its tokens (e.g. words between spaces). In parallel, each token (word) is capitalized by `fe` to make the Word Count skeleton not case-sensitive. Finally, `fm` counts words by filling a `HashMap` structure that is returned.

```
// Muscle's definition
Split<String, String> fs = new Split<String, String>() {
    @Override
    public String[] split(String p) {
        return p.split("\\s+");
    }
};

Execute<String,String> fe = new Execute<String,String>() {
    @Override
    public String execute(String p) {
        return p.toUpperCase();
    }
};

Merge<String,HashMap<String, Integer>> fm =
    new Merge<String,HashMap<String, Integer>>() {
        @Override
        public HashMap<String, Integer> merge(String[] p) {
            HashMap<String,Integer> wc = new HashMap<>();
            for (String w:p) {
                if (!wc.containsKey(w)) wc.put(w,1);
                else wc.put(w,wc.get(w)+1);
            }
            return wc;
        }
    };

// Skeleton's definition
Map<String,HashMap<String,Integer>> skel = new Map<>(fs, fe, fm);

// Input parameter
String input =  "She sells sea-shells on the sea-shore" +
                "The shells she sells are sea-shells I'm sure" +
                "For if she sells sea-shells on the sea-shore" +
                "Then I'm sure she sells sea-shore shells";
Future<HashMap<String,Integer>> future = skel.input(input);

// do something else

// Wait for result
HashMap<String,Integer> output = future.get();
```

Listing 2.1: Word count, example of a skeleton processing

Chapter 3

Separation of Concerns using Events

This chapter shows one of the contributions of this thesis: the design and implementation of a novel skeleton's separation-of-concerns based on event driven programing. This novel approach allows the introduction of non-functional concerns to skeletons without lowering its higher-level programming.

There are different programming models or strategies to address separation of concerns (SoC). Aspect Oriented Programming (AOP) [27] is the preferred and widely used programming model. However, this thesis proposes a novel approach, based on Event-Driven Programming (EDP) [31], published on 2012, that does not require the introduction of another programming framework, like *AspectJ*, which could potentially introduce conceptual disruption to the programmer. The proposed approach still applies the main concepts of Aspect Oriented Programming (AOP).

3.1 Very brief summary of Aspect-oriented Programming

As mentioned before Aspect Oriented Programming (AOP) is the preferred and widely used programming model for the design of a clear separation of concerns. The main AOP concepts [37] are

- ***Cross-cutting concerns.*** Even though implementing *modular programming*, by means of *Object-oriented Programming* or *structured programming*, where a class or a module will perform a single, specific function, they often share common, secondary requirements with other classes or modules. For example, if it is necessary to add logging or security to classes or modules. Even though each class or module has a very different primary functionality, the code needed to perform the secondary functionality (or concern) is often identical.

- ***Advice code.*** This is the additional code that includes the secondary functionality (or concern). For example, the code to implement logging or security concerns.

- ***Pointcut.*** This is the term given to the point of execution in the application at which

cross-cutting concern needs to be applied. For example the beginning of a method execution for introducing logging concerns, or just before a module call when implementing security concerns.

- ***Aspect*** The combination of the pointcut and the advice code is termed an aspect. An aspect is normally related only to one concern.

Since the concerns are secondary functionality they are normally called non-functional concerns. Therefore, aspects, typically, implements non-functional concerns like autonomic characteristics.

3.2 Event Driven Programming benefits

Event-Driven Programming (EDP) is a programming model where the program flow is determined by events. EDP has been mainly used on interactive applications, like graphical user interfaces, and has been successfully used on real-time and embedded systems [17,35] thanks to its lightweight implementation and its signal (event) processing nature.

Main AOP concepts can be introduced into Algorithmic Skeletons using EDP as follows:

- Both, AOP and EDP allow the addition of custom code: Advice code in AOP and Event Handlers in EDP.
- Such custom code can be introduced in specific points on the execution flow of a program: Join Points in AOP and Event Hooks in EDP.

On this thesis, it has been chosen to apply AOP concepts using EDP instead of using AOP directly. First, because there is no need to weave non-functional code as we can statically create event hooks as part of the Skeleton Framework. And second, to minimize programmer conceptual disruption, one of the principles presented by Cole in its manifesto [11], by avoiding the necessity of adding another framework.

The use of Events for implementing SoC in summary offers two concrete benefits:

1. It allows for precise monitoring the status of the skeleton execution with a high level of adaptability by providing on-the-fly information about the run-time execution without lowering the skeleton's high-level of abstraction and

2. It improves the separation of concerns between functional code (muscles) and non-functional concerns, in a lightweight and efficient manner.

3.3 Inversion of control problem

Skeletons use inversion of control [25] to provide high-level programming patterns to hide the complexity of parallel and distributed programming to the user. At the same time, inversion of control hide the actual execution flow to the programmer, which is essential to the implementation of non-functional concerns like autonomic computing.

Inversion of control is a design pattern where the "main" code does not control the program's execution flow. Instead the framework (caller) receives the business code as parameter and decides when and where it is executed. This allows common and reusable code being developed independently from problem-specific code, producing valuable abstraction layers.

Algorithmic Skeleton Frameworks use these abstraction layers to provide high-level parallelism exploitation patterns, hiding the primitives and explicit scheduling of parallel and distributed applications.

The cost of this high-level parallel programming abstraction is paid in control over the execution flow. This produces batch-like processing where intermediate results and information about execution are hidden from programmers, which can make handling non-functional concerns difficult.

Listing 2.1 shows an example of a common Skandium program. Once execution starts there is no mechanism to get information of the current execution or to get partial results. Therefore, working with intermediate results and information about execution, to implement non-functional concerns, lowers the programming model's high-level abstraction.

3.4 Events for skeletons

It is proposed that by triggering events during skeleton execution, intermediate results and run-time information can be used to implement non-functional concerns using event listeners.

Events are triggered during a skeleton execution. Those events are statically defined during the skeleton's design and provide information on the actual skeleton execution (e.g., partial solutions and skeleton's trace).

By means of event listeners, the non-functional programmer can implement non-functional concerns without touching the business logic code, i.e., the muscles. For example, Seq skeleton has two events defined:

- the beginning of the skeleton: Seq Before, represented as $seq(f_e)@b(i)$
- the end of the skeleton: Seq After, represented as $seq(f_e)@a(i)$

Map skeleton has eight events defined:

- the beginning of the skeleton: $map(f_s, \Delta, f_m)@b(i)$
- before split muscle execution: $map(f_s, \Delta, f_m)@b_s(i)$
- after split muscle execution: $map(f_s, \Delta, f_m)@a_s(i, |f_s|)$
- before each nested skeleton: $map(f_s, \Delta, f_m)@b_\Delta(i)$
- after each nested skeleton: $map(f_s, \Delta, f_m)@a_\Delta(i)$
- before merge muscle execution: $map(f_s, \Delta, f_m)@b_m(i)$
- after merge muscle execution: $map(f_s, \Delta, f_m)@a_m(i)$
- the end of map: $map(f_s, \Delta, f_m)@a(i)$

All events provide partial solutions, skeleton's trace, and an event identification, i, which allows correlation between $Before$ and $After$ events. Events also provide additional run-time information related to the computation; e.g., "Map After Split" provides the number of sub-problems created when splitting, $|f_s|$.

Listing 3.1 shows an example of a simple logger implemented using a generic listener. A generic listener is registered on all events raised during a skeleton execution. As parameters of handler method, there is information of the event identification: skeleton trace, when (before or after), where (e.g. skeleton, split, merge), and i parameter. Additionally the partial solution, *param*, is sent and should be returned. This allows the possibility of modifying partial solutions which could be very useful on non-functional concerns like encryption during communication. It is guaranteed that the handler is executed on the same thread as the related muscle (i.e. the next muscle to be executed after a before event, and the previous muscle executed before an after event).

```
mainSkeleton.addListener(new GenericListener() {
    @Override
    public Object handler(Object param, Skeleton[] st,
      int i, When when, Where where) {

        String logText =
          "CURR_SKEL: " + st[st.length-1].getClass().getSimpleName() +
          "; WHEN/WHERE: " + when + "/" + where +
          "; INDEX: " + i +
          "; THREAD:" + Thread.currentThread().getId();

        logger.log(Level.INFO, logText);

        return param;
    }
});
```

Listing 3.1: Example of a simple logger implemented using a generic listener

Listing 3.2 shows the first and last five events captured by the logger defined in 3.1 of the word count program presented in previous chapter, listing 2.1.

```
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: BEFORE/SKELETON; INDEX: 1; THREAD:9
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: BEFORE/SPLIT; INDEX: 1; THREAD:9
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: AFTER/SPLIT; INDEX: 1; THREAD:9
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Seq; WHEN/WHERE: BEFORE/SKELETON; INDEX: 5; THREAD:10
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Seq; WHEN/WHERE: AFTER/SKELETON; INDEX: 5; THREAD:10
...
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Seq; WHEN/WHERE: AFTER/SKELETON; INDEX: 37; THREAD:19
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Seq; WHEN/WHERE: AFTER/SKELETON; INDEX: 53; THREAD:12
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: BEFORE/MERGE; INDEX: 1; THREAD:12
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: AFTER/MERGE; INDEX: 1; THREAD:12
Feb 13, 2015 6:41:15 PM wcount.Main$4 handler
INFO: CURR_SKEL: Map; WHEN/WHERE: AFTER/SKELETON; INDEX: 1; THREAD:12
```

Listing 3.2: Output of the example of a simple logger

3.5 Event Listeners

The non-functional programmer should implement non-functional concerns as Event listeners. Event listeners can be registered or removed from event hooks on-the-fly which allows a high level of flexibility to activate or deactivate non-functional concerns during execution.

In our proposed implementation there are four listener types as interfaces. Programmers must implement any of the four following interfaces to create a new event listener. Each listener type provides a different set of information, lowering the abstraction level on a need-to-know basis.

- `SimpleListener<P>` is the most high-level listener which specifies a guard/handler receiving a parameter `P` passed between skeletons and muscles at some point in the computation, and specifies the correlation parameter `i` between event hooks.
- `TraceListener<P>` additionally specifies a `Skeleton[]` trace containing as its first element the root skeleton of a nested skeleton composition, and as its last element the currently executed

	Parameters Received	Skeletons
Simple Listener	Parameters `P` and `i`	All skeletons
Trace Listener	+ Skeleton trace	All skeletons
Condition Listener	+ Condition result	if, $while$, $d\&c$
Generic Listener	+ Event hook identifiers	All skeletons

Table 3.1: Comparison of Listener's type

element. This is useful when the event listener is registered in more than one part of the skeleton nesting tree.

- `ConditionListener<P>` additionally specifies a boolean parameter as a result of Condition muscles in if, $while$ and $d\&c$ skeletons.

- `GenericListener` is a special listener that can be applied to any event hook and additionally specifies the event hook identification (when and where).

Table 3.1 shows the comparison of the different listener's types available and their parameters. It also shows the Skeletons supported by each type.

Programmers also have the possibility of registering/removing Generic Listeners. A Generic Listener can be registered in any event hook, and can be used to create common functionality for several events, just like point cuts in AOP, at the price of sacrificing type verification. The listener is applied to all the event hooks of nested skeletons. If the users needs to only apply the listener to some event hooks, he/she can use the following parameters to filter them:

- `Class pattern`. Filters by the Skeleton dimension (e.g. `Map.class`, so the listener will be registered/removed on all the Map events), the wild card is `Skeleton.class`.

- `When when` can take the values `When.BEFORE` or `When.AFTER`, the wild card is `null`.

- `Where where` filters event hooks related to a specific skeleton element, for example a condition muscle, split muscle, nested skeleton, etc; the wild card is `null`.

3.6 Event hooks

Event Hooks define specific location in the code of the skeletons where additional code could be integrated. Each pattern provides a different set of hooks, and each hook applies to a specific set of Event Listener types.

Hooks are presented in the following notation: $\Delta@when_{where}(params)$. For example, all skeletons provide before,$\Delta@b(i)$, and after,$\Delta@a(i)$ hooks. More specific hooks are also available depending on the pattern. For example, the if skeleton provides the before condition, $if(f_c, \Delta_t, \Delta_f)@b_c(i)$,

Skeleton	Before	After
Δ	$\Delta: \quad i, P$	$\Delta: \quad i, R$
$pipe(\Delta_1, \Delta_2)$	$\Delta_i: \quad i, P_i$	$\Delta_i: \quad i, R_i$
$if(f_c, \Delta_t, \Delta_f)$	$f_c: \quad i, P$ $\Delta_x: \quad i, P, x$	$f_c: \quad i, P, f_c(P)$ $\Delta_x: \quad i, R, x$
$while(f_c, \Delta)$	$f_c: \quad i, P, iteration$ $\Delta: \quad i, P, iteration$	$f_c: \quad i, P, f_c(P), iteration$ $\Delta: \quad i, R, iteration$
$for(n, \Delta)$	$\Delta: \quad i, P, iteration$	$\Delta_i: \quad i, R, iteration$
$map(f_s, \Delta, f_m)$	$f_s: \quad i, P$ $\Delta_i: \quad i, P_i$ $f_m: \quad i, \{R_j\}$	$f_s: \quad i, \{P_j\}$ $\Delta_i: \quad i, R_i$ $f_m: \quad i, R$
$fork(f_s, \Delta, f_m)$	$f_s: \quad i, P$ $\Delta_i: \quad i, P_i$ $f_m: \quad i, \{R_j\}$	$f_s: \quad i, \{P_j\}$ $\Delta_i: \quad i, R_i$ $f_m: \quad i, R$
$d\&c(f_c, f_s, \Delta, f_m)$	$f_c: \quad i, P$ $f_s: \quad i, P$ $\Delta_i: \quad i, P_i$ $f_m: \quad i, \{R_j\}$	$f_c: \quad i, f_c(P)$ $f_s: \quad i, \{P_j\}$ $\Delta_i: \quad i, R_i$ $f_m: \quad i, R$

Table 3.2: Event hooks

and after condition, $if(f_c, \Delta_t, \Delta_f)@a_c(i, return(f_c))$ hooks. Note that the "after condition" hook sends also as parameter the result of the execution of f_c.

Hooks for before and after nested skeletons are more specific on each pattern. For example, *pipe* has two subskeletons while if has one and the hooks definition differ accordingly. However, the only type unsafe hooks correspond to *pipe*'s nested parameter. This could be resolved, however, by extending *pipe*'s definition from `Pipe<P,R>` to `Pipe<P,X,R>` to account for the intermediary type.

Table 3.2 shows the current set of event hooks implemented in Skandium. The second and third columns show what parameters are sent for each stage of the skeleton's execution. The first row implies that all skeletons have a before and- after event hook. The parameters P and R correspond to the input and output of each stage.

3.7 Roles

Separation of concerns is essentially a software engineering matter. The proposed separation of concerns recognizes three different roles:

- **Skeleton programmer** whose job is to materialize parallelism patterns into skeletons.
- **Non-functional programmer** who implements other, non-parallel, non-functional features, and
- **Business programmer** who uses both skeleton patterns and non-functional features to implement the final business application.

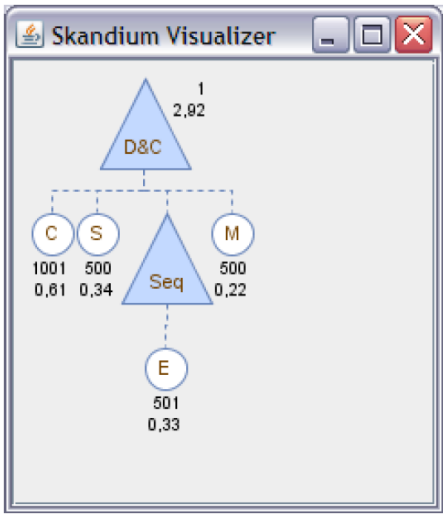

Figure 3.1: Online Performance Monitor

3.8 Logger and Online Performance Monitoring

As proof of concept of the approach proposed in this chapter, a logger and a visual performance monitor were developed.

The logger, similar to the example in listing 3.1 is a generic tool that can be activated and deactivated on the fly during a skeleton's execution. The logger traces relevant information about the execution which can be used to identify performance bottlenecks.

The visual execution monitor is an extension of the work in [9] which now provides online performance monitoring rather than post-mortem. As shown in Figure 3.1, for each muscle and skeleton, the number of invocations and processing time spent is updated continuously.

3.9 Overhead analysis

An overhead analysis was made using a parallel QuickSort algorithm implemented as one divide and conquer skeleton. The implementation is based on the QuickSort definition found in [29] and can be seen in listing 3.3. Here `Range` is a simple class that envelop the array to be sorted and the left and right indexes defining a sub-problem (listing 3.4). The split condition is based on a `THRESHOLD` that controls the size of the range to be sorted by a Java native sorting. The partition algorithm used during split muscle is defined in listing 3.5. The `rarray` algorithm is a simple algorithm that produces an array of size `SIZE` of random integers. The variable `THREADS` defines the number of threads to be used for sorting.

The Divide and Conquer skeleton generates a variable number of event hooks that is linearly proportional to the size of the input. All tests were run on a dual core processor of 2.40 GHz with

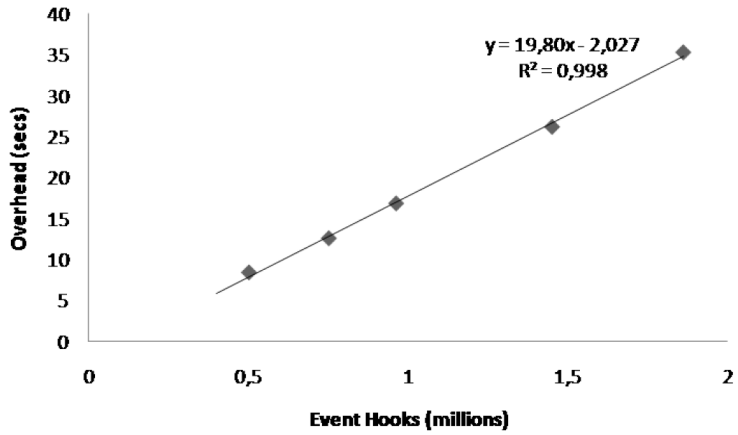

Figure 3.2: Absolute overhead processing time

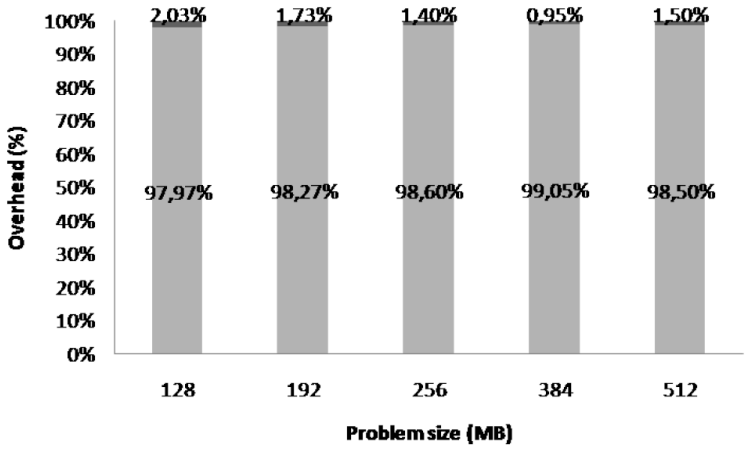

Figure 3.3: Relative overhead processing time

2GB of RAM.

The analysis compares the execution of the same problem with two different versions of Skandium, the one before the events support, and the one that includes events. This analysis calculates the overhead caused only by the inclusion of the event instructions (event hooks) during the transformation and reduction steps. Time presented is an approximation of CPU time.

Figure 3.2 shows the result of the analysis; The x-axis corresponds to the number (in millions) of event hooks, while y-axis shows the calculated overhead (CPU time with events minus CPU time without events). As result we see a linear increase with a slope of 19.80 seconds for each million of event hooks created. *Therefore the overhead is 19.80 microseconds per event hook.*

Figure 3.3 shows the relative overhead processing time where x-axis corresponds to the problem size in MB. The light gray part on the bottom of each bar shows the percentage of total CPU time used to solve the problem, and the dark gray part of the top of each bar shows the percentage used

```
// Muscle's definition
Condition<Range> fc = new Condition<Range>() {
    @Override
    public boolean condition(Range r) throws Exception {
        return r.right - r.left > THRESHOLD;
    }
};
Split<Range, Range> fs = new Split<Range, Range>() {
    @Override
    public Range[] split(Range r)  {
        int i = partition(r.array, r.left, r.right);
        Range[] intervals = {   new Range(r.array, r.left, i-1),
                                new Range(r.array, i+1, r.right)};
        return intervals;
    }
};
Execute<Range,Range> fe = new Execute<Range,Range>() {
    @Override
    public Range execute(Range r) {
        if (r.right <= r.left) return r;
        Arrays.sort(r.array, r.left, r.right+1);
        return r;
    }
};
Merge<Range,Range> fm = new Merge<Range,Range>() {
    @Override
    public Range merge(Range[] r) throws Exception {
        Range result = new Range( r[0].array, r[0].left, r[1].right);
        return result;
    }
};


// Skeleton's definition
Skeleton<Range, Range> sort = new DaC<Range, Range>(fc,fs,fe,fm);

// Input parameters with the defaults singleton Skandium object
Skandium skandium = new Skandium(THREADS);
Stream<Range, Range> stream = skandium.newStream(sort);
Future<Range> future = stream.input(new Range(rarray(SIZE),0, SIZE-1));

// Do something else here.

// Get results
Range result = future.get();
```

Listing 3.3: Quick Sort implementation

```
public class Range {

        int array[];
        int left;
        int right;

        public Range(int array[], int left, int right){
                this.array=array;
                this.left=left;
                this.right=right;
        }
}
```

Listing 3.4: Range class definition

```
public static int partition(int[] a, int left, int right) {
        int i = left - 1;
        int j = right;
        while (true) {
                while (less(a[++i], a[right]))
                        ;
                while (less(a[right], a[--j]))
                        if (j == left) break;
                if (i >= j) break;
                exch(a, i, j);
        }
        exch(a, i, right);
        return i;
}

private static boolean less(int x,  int y) {
        return (x < y);
}

private static void exch(int[] a, int i, int j) {
        int swap = a[i];
        a[i] = a[j];
        a[j] = swap;
}
```

Listing 3.5: Partition algorithm definition

to interpret the empty event hooks.

This is the worst case behavior expected since DaC is the skeleton that generates the most event hooks in proportion to the input size. Furthermore the implementation of events in Skandium is itself a parallel problem and thus in actual parallel execution this overhead is linearly parallelized with respect to the wallclock overhead time.

3.10 Conclusions

This chapter proposed a novel separation of concern approach extending the skeletons model based on Event-Driven Programming, by means of event hooks and event listeners. This novel approach improves on a clear separation of concerns between the non-functional aspects, implemented using event handlers, and functional code (muscles). Secondly, it provides access to users of run-time execution information on the fly without lowering the skeleton's high-level abstraction.

The proposed approach has been verified by implementing a logger and a visual performance tool, and our implementation has been measured to have a negligent performance overhead. This separation of concerns (SoC) can be used to implement several other non-functional concerns like:

- *Audit, control, and reporting.* As well as the logger already implemented, another controlling and reporting tools could be built in order to address an online quality goals analysis (e.g. processor speed, memory, disk space, network bandwidth, etc.)
- *Error analysis.* Using events, a full skeleton debugger could be implemented. Another tool that can be implemented using the ability of registering and removing listeners dynamically, is a tracer that increases/decreases trace level online.
- *Security.* Using the ability to manipulate intermediate results, algorithms of encryption/decryption could be used during communication.

Next chapter describes the use of such events to provide a framework for the execution of skeletons with a high level of adaptability by introducing autonomic characteristics. Which is the main purpose of this thesis. The introduction of autonomic characteristics to skeletons is an active research topic in the parallel and distributed computing area [3, 12, 22]. The benefits of this type of technology are: (1) reduction of management costs of parallel systems, and (2) hiding the complexity of parallel programming to the developer. Without events, the introduction of autonomic characteristics to skeletons is a very challenging problem because of the skeleton's inversion of control.

Chapter 4

Autonomic Skeletons

This chapter is the core of this thesis because it presents the proposed approach to introduce self-configuration and self-optimization characteristics to skeletons using events. This chapter begins by defining the particular Quality of Services (QoSs) implemented. Then, it shows how to track the skeleton's execution and how to predict the remaining work. This chapter finishes by presenting an example of the execution.

Autonomic Computing (AC) is often used to achieve a given QoS that the system has to guarantee as much as possible. For example, a type of QoS is Wall Clock Time (WCT, time needed to complete the task). On an AC system that supports the WCT QoS, it is possible to ask for a WCT of 100 seconds for the completion of a specific task. This means that the system should try to finish the job within 100 seconds by means of self-managing activities (e.g., modifying process priorities or number of threads).

This thesis focuses on guaranteeing a given execution time for a skeleton by optimizing the number of threads allocated. The QoSs autonomically managed to be self-configured and self-optimized are: (1) Execution Wall Clock Time, and (2) Level of Parallelism.

4.1 Wall Clock time, and Level of Parallelism

The proposed autonomic characteristics related to the QoSs WCT and LP works as follows: if the system realizes that it will not reach the WCT goal with the current LP, but it will reach it, if the LP is increased, it autonomically increases the LP. However, if the system realizes that it will target the WCT goal even if it decreases the LP, it autonomically decreases the LP. To avoid potential overloading of the system, it is possible to define a maximum LP.

Why would it not always use the maximum number of threads in order to get the fastest possible WCT? There are several reasons that drive the decision to not do so. First, energy consumption and heat production. The more work a processor does, the more energy it consumes, and the more

heat it produces, implying even more energy needed for the cooling system. Another reason is to improve the overall system performance over the performance of a single application, when it is possible to free resources that could be used by other processes.

It is not always true that the WCT decreases if the number of active threads increases. The hardware cache system could lead to situations where higher performance is achieved with a number of threads even smaller than the available hardware threads. For simplicity, it was assumed that the LP produces a non-strictly increasing speedup. This simplification obeys the complexity of inferring the memory access pattern of the functional code.

The principles of the proposed autonomic adaptation framework are the following. Using events, it is possible to observe the computation and monitor its progress, find out when tasks start and finish, and how many tasks are currently running. Thanks to the use of events, it is possible to monitor this information without modifying the business code, and in a very dynamic way: the execution of each individual skeleton is monitored. Consequently, the execution can be adapted as soon as it is "detected" that the quality of service expected will not be achieved.

In practice, Skandium uses functions to estimate the size and the duration of the problem to be solved, and, if necessary, to allocate more resources to the resolution of a given skeleton. This way, the autonomic adaptation targets the currently evaluated instance, and not the next execution of the whole problem as in most other approaches.

Skandium uses events both to build up an estimation of the execution time, and to react instantly if the current execution may not be achieved in time. In the last case, more resources are allocated to improve the execution time. On the contrary, if the execution is fast enough or if the degree of parallelism must be limited, the number of threads allocated for the task can be decreased.

The algorithm to calculate the optimal WCT is a greedy one, while the algorithm to calculate the minimal number of threads to guarantee a WCT goal is NP-Complete. Therefore Skandium does not reduce the LP as fast as it increases it. The algorithm implemented for decreasing the number of threads first checks if the goal could be targeted using half the number of threads; if it can, it decreases the number of threads to the half.

4.2 Evaluating the remaining execution time

This section shows how it is possible to estimate in advance the WCT. Let $t(m)$ be the estimated execution time of the muscle m and let $|m|$ be the estimated cardinality of the muscle m. The estimated cardinality is only defined to the muscles of type Split and Condition. The cardinality of a muscle of type Split is the estimated size of the sub-problem set returned after its execution; the

cardinality of a Condition muscle is the estimated number of times the muscle will return "true" over the execution of a While skeleton, or the estimated depth of the recursion tree of a Divide & Conquer skeleton.

Assume that the values of the functions $t(m)$ and $|m|$ are known in advance. It is then possible to draw an Activity Dependency Graph (ADG) like the one shown in Figure 4.1.

Figure 4.1 shows an example of an ADG related to an actual skeleton execution of two nested Map skeletons, $map(f_s, map(f_s, seq(f_e), f_m), f_m)$. Let's assume that $t(f_s) = 10$, $t(f_e) = 15$, $t(f_m) = 5$, and $|f_s| = 3$. Each activity is shown as a rectangle with three columns and corresponds to a muscle execution. The middle column shows the muscle related to the activity. The first and third columns represent the start and end time respectively. If the activity has been executed, the first and third columns contain the actual time measured after execution, in a light-gray box. If the activity has not yet been executed, estimations are shown for the start or for the end time: the top box shows the "best effort" estimated time, and the bottom box shows the "limited LP" estimated time. The skeleton of Figure 4.1 has been executed using an LP of 2 threads, and the ADG has been taken at WCT= 70.

For example, the ADG of figure 4.1 shows that the execution started at time 0 where the first *split* muscle was executed, and finished its execution at time 10, producing 3 sub problems. Two of them started at time 10, but one started at time 65. As the ADG represents the situation at time 70, it is shown that the *split* that started at 65 has not yet finished, but it is expected to finish at 75 in either the *Best Effort* case or in the $LP(2)$ case.

The best effort strategy for estimating t_i uses the following formula: $t_i = max_{a \in A}(a_{t_f})$, where A is the set of predecessor activities and a_{t_f} is the end time of activity a. If $max_{a \in A}(a_{t_f})$ is in the past, $ti = currentTime$. Best effort strategy assumes an infinite LP; it calculates the best WCT possible, i.e. the end time of the last activity with a best-effort strategy.

Optimal LP is calculated using a time-line like the one presented on Figure 4.2. Figure 4.1 and figure 4.2 shows the same situation but in different ways. Figure 4.2 shows the estimated LP changes during the skeleton execution. It is possible to build the timetable on the *Best Effort* case using the estimated start and end times and the above formulas. Figure 4.2 shows a maximum requirement of 3 active threads during the interval $[75, 90)$. Therefore the optimal LP for this example is 3 threads.

Limited LP strategy is used to calculate the total WCT under a limit of LP. In this case LP is not infinite, therefore the t_i calculation has an extra constraint: at any point of time, LP should not exceed the limit. As you can see on Figure 4.2, the LP for the Limited LP case never exceeds 2 threads, and the total WCT will be 115.

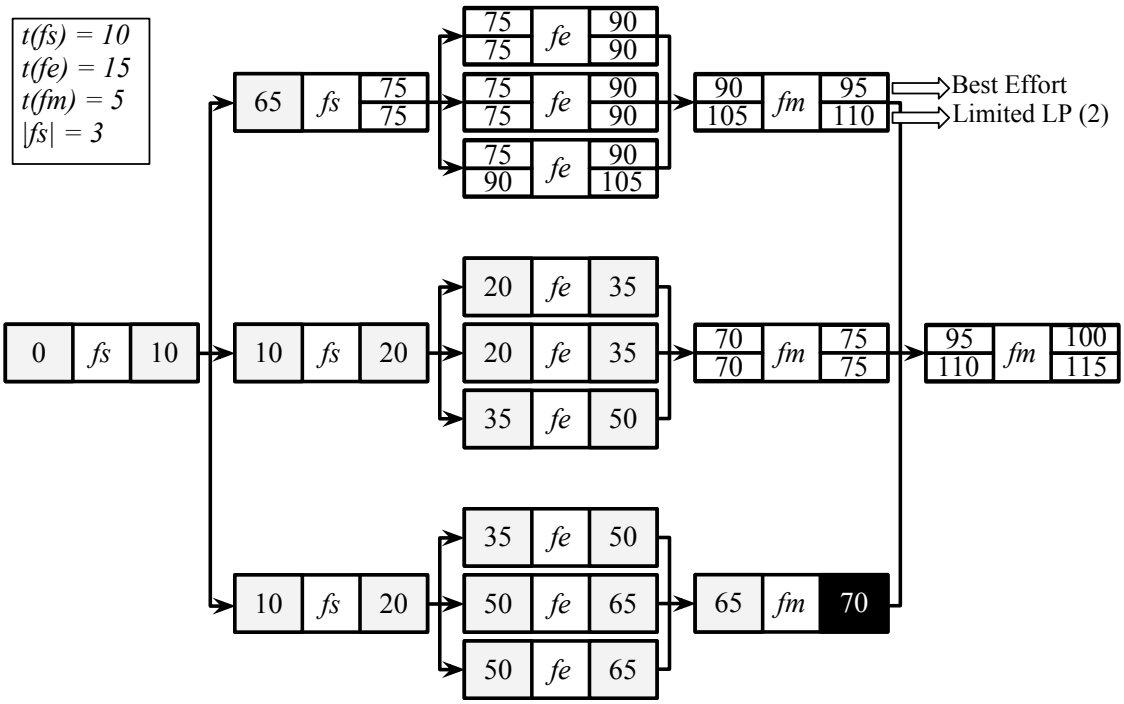

Figure 4.1: Example of an Activity Dependency Graph

If we set the WCT QoS goal to 100, Skandium will autonomically increase LP to 3 in order to achieve the goal.

Notice that this estimation algorithm implies that the system has to wait until all muscles have been executed at least once. In the example, Figure 4.1, it is possible to estimate the work left of the skeletons still running because all muscles have been executed at least once at the moment of ADG analysis (black box). However, Skandium also supports the initialization of $t(m)$ and $|m|$ values.

4.3 Event-based monitoring

All these previous analyses have been conducted under the assumption that we already know the values of $t(m)$ and $|m|$. We will explain below how to estimate these values. The estimation algorithm implemented on Skandium is based on history: "the best predictor of the future behavior is past behavior".

This section shows how to monitor execution time and skeleton cardinality using events. Chapter 5 presents the evaluation of different strategies for time and cardinality prediction.

Using events make it possible to trace a skeleton execution without touching functional code. The natural way to design a system based on events is by state machines. We use state machines for tracking the Skeleton execution to create or update the ADG. Figures 4.3 and 4.4 show the

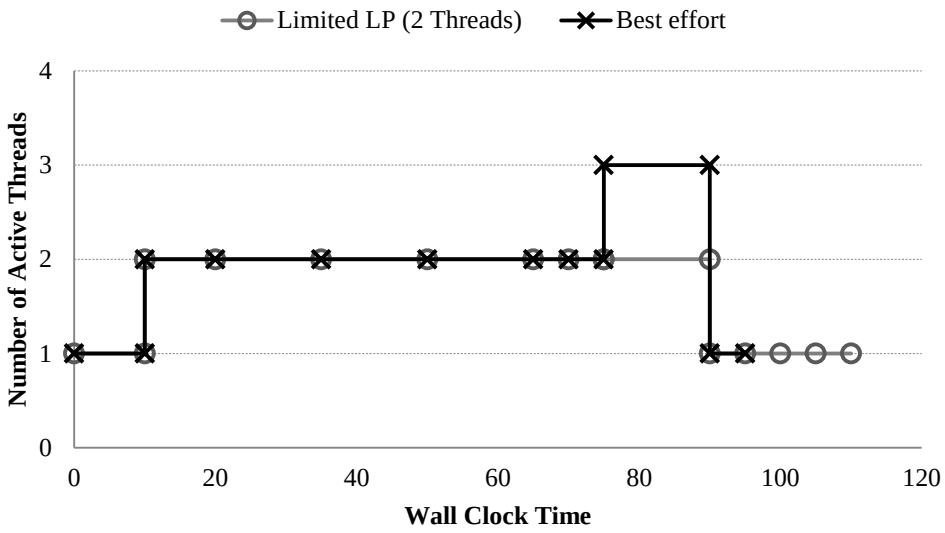

Figure 4.2: Example of timeline used to estimate the total WCT and the optimal level of parallelism

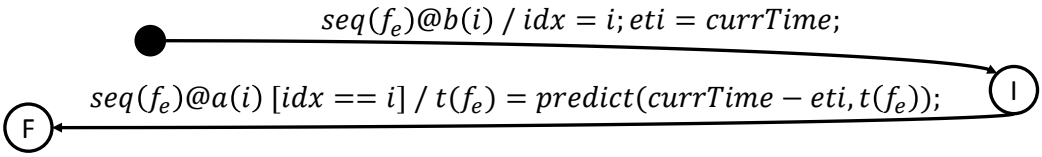

Figure 4.3: $StateMachine(seq(f_e))$ definition

state machines implemented for Seq and Map skeletons respectively. On those figures, *predict* is a function that takes the previous prediction and the current value of the last execution and predicts the new value of $t(m)$ and $|m|$. Different predict functions are investigated in Chapter 5.

Each type of skeleton included on Skandium has its corresponding state machine except if and $fork$ skeletons which are not yet supported. If skeleton produces a duplication of the whole ADG that could lead to performance issues, and $fork$ skeleton produces a non-deterministic state machine. The support for those types of skeletons are under construction.

The state machine for *Seq* skeleton is shown in Figure 4.3. It is activated once the *seq before* event, $seq(f_e)@b(i)$, have been triggered. *seq before* event has a parameter, i, that identifies the executed skeleton. Its value is stored on the local variable *idx*. Another local variable, *eti*, holds the time-stamp of the start of the muscle execution; once *seq after* event, $seq(f_e)@a(i)$, of the corresponding *idx* is triggered, the $t(f_e)$ value is computed and updated according to the evaluation strategy presented in Chapter 5.

The state machine for Map skeleton, Figure 4.4, is a little more complex but its general idea is the same- to trace the skeleton's execution, and to update the values of $t(f_s)$, $t(f_m)$, and $|f_s|$ as follows.

Map State Machine starts when a *Map Before Split* event, $map(f_s, \Delta, f_m)@bs(i)$ is triggered. Similar to Seq case, it has an identification parameter, i, that is stored in a local variable idx that serves as guard for the following state transitions. The time-stamp of the start of split muscle is stored in a local variable sti. The transition from state I to S occurs when the *Map After Split* event,$map(f_s, \Delta, f_m)@as(i, fsCard)$, is triggered, where the $t(f_s)$ and $|f_s|$ estimations are updated. At this point all the children State Machines, $StateMachine(\Delta)$, are on hold waiting to start. When all children State Machines are in F state, the Map State Machine is waiting for the *Map Before Merge* event, $map(f_s, \Delta, f_m)@bm(i)$. Once it is raised, the mti local variable stores the time-stamp just before the execution of merge muscle. The transition from M to F state occurs when the *Map After Merge* event, $map(f_s, \Delta, f_m)@am(i)$, occurs, where the $t(f_m)$ estimate is updated.

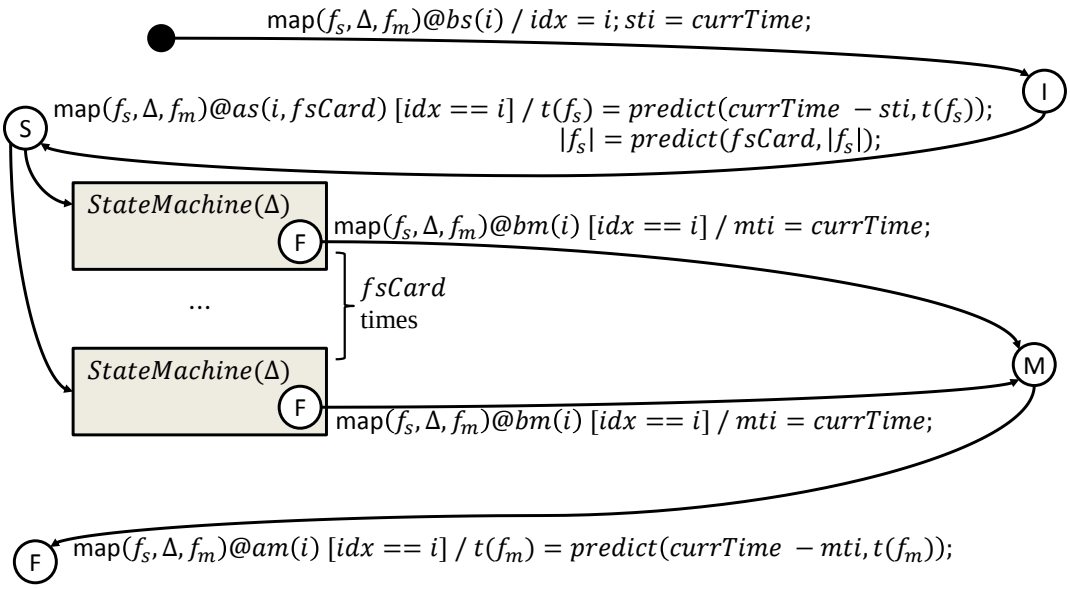

Figure 4.4: $StateMachine(map(f_s, \Delta, f_m))$ definition

As presented in this section, a given execution time is guaranteed for a skeleton by optimizing autonomically the number of threads allocated to its execution and estimating the remaining execution time while the skeleton is running. The next section shows an execution example of this approach, demonstrating that it is indeed light-weight and allows autonomic adaptation to occur while the skeleton is being evaluated.

4.4 Execution example

This section presents an execution example to show the feasibility of our approach.

The example is an implementation of a Hashtag and Commented-Users count of 1.2 million Colombian Tweets from July 25th to August 5th of 2013 [13]. The problem was modeled as two

nested Map skeletons: $map(f_s, map(f_s, seq(f_e), f_m), f_m)$, where f_s splits the input file on smaller chunks; f_e produces a Java HashMap of words (Hashtags and Commented-Users) and its corresponding partial count; and finally f_m merges partial counts into a global count.

The executions were done on an Intel(R) Xeon(R) E5645 at 2.4 GHz each, with a total of 12 cores and 24 CPU Threads, 64 GB RAM Memory, running Skandium v1.1b1.

To show the feasibility of our approach we present the comparison of three different execution scenarios:

1. *Goal Without Initialization* - autonomic execution with a WCT QoS set at 9.5 secs without initializing the estimation functions;

2. *Goal With Initialization* - autonomic execution with a WCT QoS goal set at 9.5 secs with initialization of estimation functions.

3. *WCT Goal of 10.5 secs* - autonomic execution with WCT QoS goal of 10.5 secs.

Figures 4.5, 4.6 and 4.7 show the change on number of active threads during execution and the WCT goal is shown as a vertical red line.

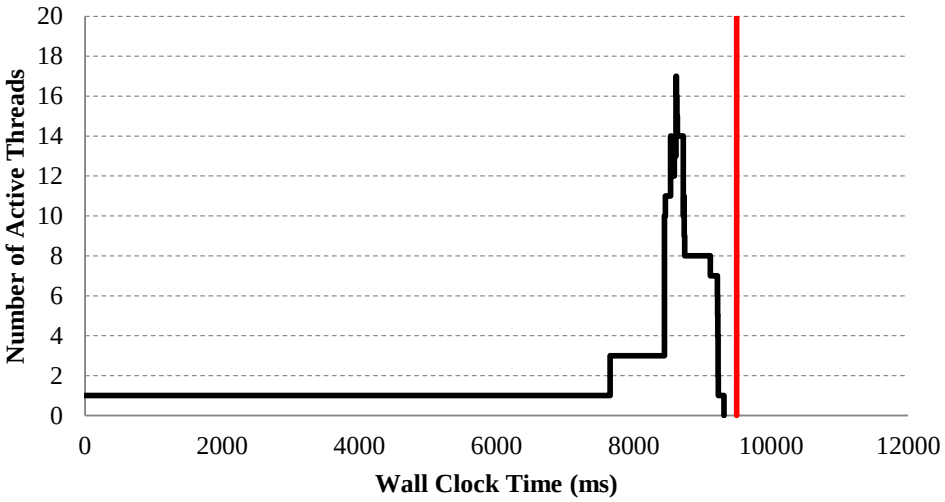

Figure 4.5: "Goal Without Initialization" execution

Goal Without Initialization (figure 4.5)

The purpose of the first scenario is to show the behavior of the autonomic properties without any previous information with an achievable goal.

In this scenario Skandium waits until the first Merge muscle is executed in order to have all the information necessary to build the ADG. This occurs at 7.6 secs.

At that point, the first estimation analysis occurs and Skandium increases to 3 threads. It reaches its maximum number of active threads, 17, at 8.6 secs when almost all the f_e muscles can be executed in parallel. This scenario finishes its execution at a WCT of 9.3 secs reaching its targeted goal.

But why was a goal of 9.5 secs chosen? The total sequential work (WCT of the execution with 1 thread) takes 12.5 secs, therefore any goal greater than 12.5 secs does not produce the necessity of an LP increase. On the other hand, Skandium took 7.6 secs to execute at least the first Split, one other Split, all the execution muscles of the second Split, and one Merge. The first split took 6.4 secs (as we will see in scenario 2), and the second level split is expected to be 7 times faster than the first one, and 0.04 secs per Execution and Merge muscles. Therefore in the best case it is expected that the system could finish at 8.63 secs. What could occur is that the increase of threads happened after any of the left splits have already started their execution. In such case, Skandium cannot achieve the maximum LP. Therefore, in the worst case, it is necessary to wait an extra split execution. A WCT of 9.54 secs is expected in this situation.

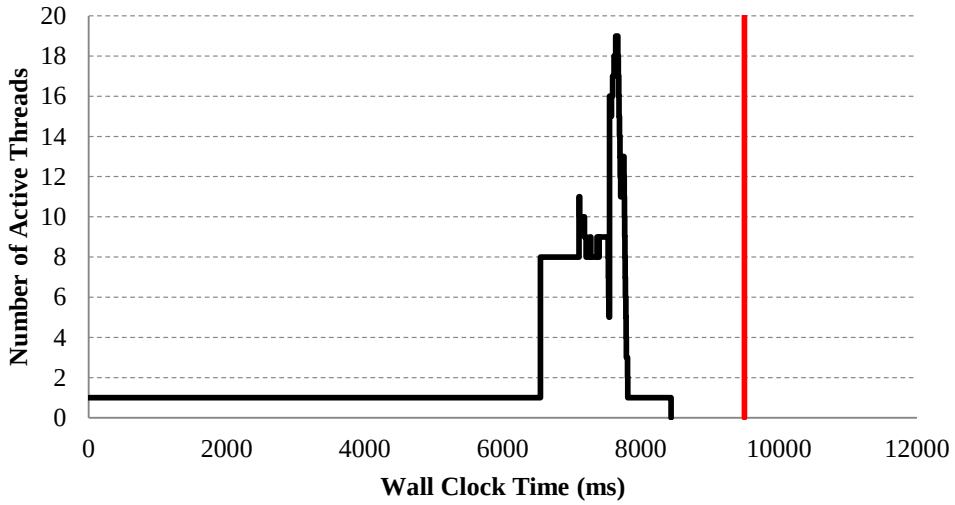

Figure 4.6: "Goal With Initialization" execution

Goal With Initialization (figure 4.6)

The purpose of the second scenario is to show how a previous execution could offer important information that can be used to improve the estimations. In this scenario we chose a goal of 9.5 secs to compare with the same parameters of the first scenario except the initialization of variables.

Here the $t(m)$ and $|m|$ functions are initialized with their corresponding final value of a previous execution. Figure 4.6 shows that Skandium increases the number of threads to 8 at 6.4 secs of WCT execution. As you can notice, this increase has to occurred before the first Merge muscle has been

executed. Skandium does not increase the number of threads before because it is performing I/O tasks, i.e., reading the input file stream on the first split muscle where there is no need for more than one thread. The execution reaches its maximum LP, 19 active threads, at 7.6 secs. This scenario finishes its execution at a WCT of 8.4 secs. It shows a better execution time than experiment (1) where Skandium needed some time to detect that the WCT could not be achieved. One can notice that experiment (1) shows the additional cost paid during the first execution in order to initialize the cost functions.

This experiment finishes at 1.1 secs earlier than targeted goal. The reason is that Skandium does not reduce the LP as fast as it increases. In fact, Skandium increased the number of threads from 1 to 8 in a single step, but, as described in previous section, Skandium decreases the number of threads by a maximum factor of 2. This produces the early WCT.

It could be expected that this execution uses all 24 threads in its maximum LP. Theoretically all the execution muscles should have been executed in parallel and therefore all physical threads should be used. However, in practice some execution muscles took less time than others, and therefore the same thread is used to solve more than one execution muscle. As consequence, the parallelism is reduced.

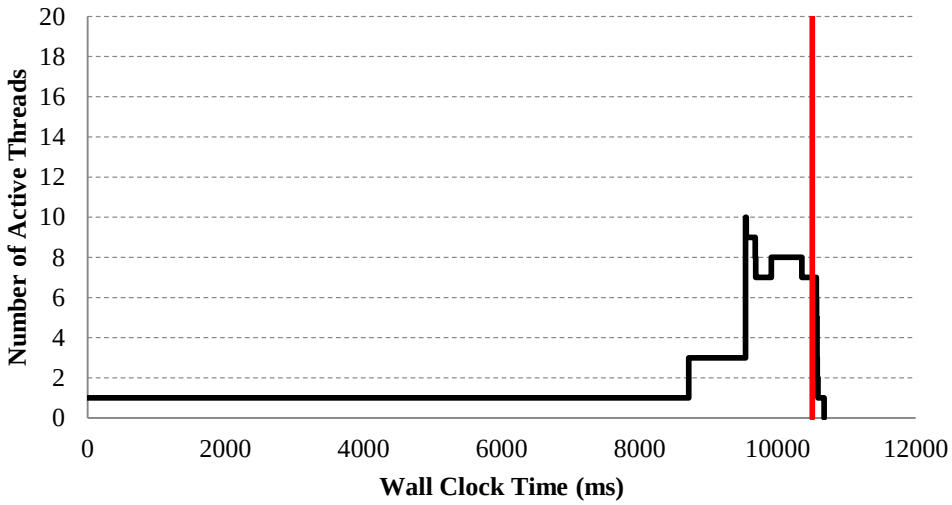

Figure 4.7: "WCT Goal of 10.5 secs" execution

WCT Goal of 10.5 secs (figure 4.7)

The purpose of this scenario is to show the behavior of the LP when it is needed to increase the number of threads but not as much as in the first scenario. In this scenario Skandium has more clearance. Therefore, a lower LP is expected.

Figure 4.7 shows that Skandium, at 8.7 secs of execution, realizes that it won't reach its goal

with the current LP, therefore it increases the LP to a maximum of 10 active threads.

Note that the maximum LP of this execution is lower than the used on the two previous executions because the WCT goal has more room to allocate activities with less number of threads. It finishes at 10.6 secs.

These examples have shown the feasibility of the proposed solution. It has been illustrated how instrumenting skeletons with events allows to discover that the execution of a skeleton might be too slow or too fast and to adapt the degree of parallelism dynamically, during the execution of this same skeleton. Not only this methodology allows to adapt the degree of parallelism faster, but it is also adapted to cases where the execution time is dependent on the input data, while strategies using the execution time of the previous instance to adapt the degree of parallelism of the next skeleton are unable to achieve this.

Chapter 5

Estimating the muscle's execution time and cardinality

As stated in chapter 4, it is possible to monitor execution time and skeleton cardinality using events and state machines. This chapter presents the result of the evaluation of two statistical strategies to estimate the muscle's execution time and cardinality. The goal of this chapter is to provide a recommendation of which strategy produces better estimates under which circumstances.

There is other work in the literature that needs to predict execution time of parallel applications. One of most relevant is [30], where the authors introduce an estimation method for parallel execution times, based on identifying separate "parts" of the work done by parallel programs. The time of parallel program execution is expressed in terms of the sequential work (muscles) and of the parallel penalty. The parallel work estimation proposed in this thesis uses a different approach based on Activity Dependency Graphs (ADGs), chapter 4, which models the work estimation as an activity scheduling problem. The sequential work estimation, presented in this chapter, uses a lightweight history-based estimation algorithm that allows on-the-fly estimation tuning, and is therefore complementary.

The estimating strategies used in this thesis should fulfill the next two characteristics:

- they should allow online prediction,
- and they should be based on previous values, but they should not store all of them. Therefore they should be lightweight.

Moving average [39] calculations are simple metrics that fit with those requirements. Specifically, two algorithms were evaluated: *Exponential Moving Average (EMA)* [38] and *Weighted Moving Average (WMA)* [41].

5.1 Exponential Moving Average (EMA)

EMA has the form:

$$E_{n+1} = \rho \times A_n + (1 - \rho) \times E_n$$

where E_{n+1} is new estimated value, A_n is the last actual value, E_n is the previous estimated value, and ρ is a system parameter between 0 and 1.

ρ defines an exponential decrease of the weight of the samples: the weight of the last actual value is ρ, the weight of the next-to-the-last value is $\rho \times (1 - \rho)$, and so on. Finally, the weight of the first value is $\rho \times (1 - \rho)^{(n-1)}$. A proper value for ρ depends on the relation among previous values and the new expected value. For example, if ρ is set to 1, then only the last measure will be taken into account; but, if ρ is set to 0, then only the first value will be taken into account. Overall, if ρ is closed to 0 then the value will not be too sensitive to recent variations and the adaptation will be triggered slowly, following a stable tendency of results. If ρ is close to 1, the framework will quickly react to recent values. This measure should be adapted to cases where the last measure is a good prediction for the next one, but can still be slightly stabilised in case one of these measures is erratic.

5.2 Weighted Moving Average (WMA)

WMA is defined as follows:

$$E_{n+1} = \frac{\alpha \times A_n + n \times E_n}{\alpha + n}$$

Here, n, the number of samples, is taken into account producing a smoother distribution of weights. The parameter α allows for changing the weight factor. If $\alpha = 1$, we will get a constant distribution of weights, therefore E_{n+1} will be the mean; if $\alpha = 2$, we will get a linear distribution of weights; and if $\alpha > 2$, we will get an exponential distribution of weights, similar to EMA with a $\rho = \frac{\alpha}{\alpha+n}$, but it has an important difference: here, ρ depends on n. Therefore for large enough n, WMA will be less sensitive to recent variations. This measure should work fine if variables are stable enough, though some of the measures might be erratic: if a stable result has been produced enough times, one erratic measure will not influence very much the prediction.

5.3 Evaluation

The accuracy of the estimations made by several prediction algorithms was evaluated:

- EMA with $\rho \in \{0.5, 0.25, 0.125\}$,

- WMA with $\alpha \in \{2, 5, 10\}$.

The evaluation was performed on six benchmarks:

- Bubblesort, ordering an array of $65,536$ random elements.
- Mergesort, ordering an array of $65,536$ random elements.
- Naive NQueens for a board of 15×15 board size.
- Pi calculation with $3,000$ decimals.
- Quicksort, ordering an array of $65,536$ random elements.
- Strassen matrix multiplication of two 2048×2048 random matrixes.
- Word count for 1.2 million twitter posts.

In order to evaluate the accuracy of the estimations and to compare different algorithms with different parameters on different benchmarks we used *Normalized root-mean-square deviation (NRMSD)* [40] as a measure of prediction accuracy:

$$RMSD = \sqrt{\frac{\sum_{k=1}^{n}(E_k - A_k)^2}{n}}$$

$$NRMSD = \frac{RMSD}{max_{k=1}^{n}(A_k) - min_{k=1}^{n}(A_k)}$$

As introduced above, A_k is the actual measure for sample k and E_k is the prediction for sample k.

Figures 5.1 and 5.2 show two views of the result of the estimation algorithm evaluation. Figure 5.1 shows the different *NRMSD* grouped by benchmark, while figure 5.2 shows a drill down of the different *NRMSD* grouped by type of variable: execution time or cardinality. Only variables that change their value over time are taken into account. For example, $|f_s|$ in Quicksort and Mergesort is always 2, and the NRMSD will be 0 for all estimation algorithms. Therefore $|f_s|$ in Quicksort and Mergesort is not included in the analysis.

As shown in figure 5.1, there is not much discrepancy on the use of the different estimation algorithms for the benchmarks Mergesort and Strassen, which means that the execution time and cardinality of the estimated variables do not change too much.

Bubblesort, PI, and Quicksort show a similar distribution of NRMSDs. Here EMA with an $\alpha = 0.5$ have a slightly better behavior. Among the estimation algorithms, $EMA(0.5)$ is the one that gives more weight to the last value with respect to the previous ones. This means that these benchmarks behave accordingly. This is evident in Bubblesort, since each iteration reduces the number of comparisons in a constant factor. This means that the expected value of the execution

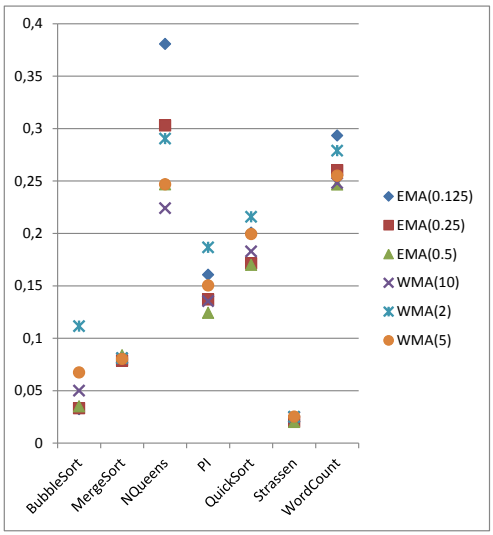

Figure 5.1: Normalized deviation grouped by benchmark

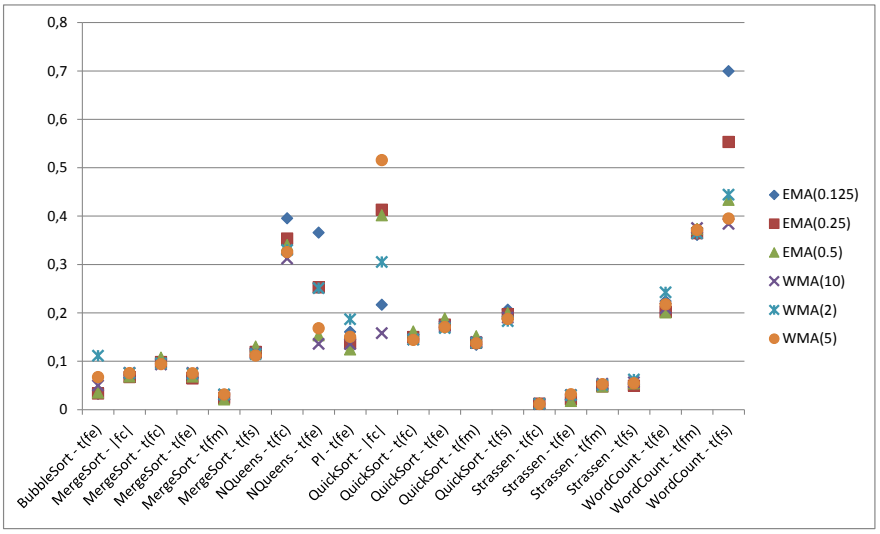

Figure 5.2: Normalized deviation grouped by type variable

time of the next iteration should be closest to execution time of the last iteration and farthest from the first one.

Similar behavior is expected in PI and QuickSort due to their D&C nature. During the split phase, the execution time of the deeper muscles should be closer to the one of the other deeper muscles than the outer ones. Similarly, during the merge phase the execution time of the outer muscles should be closer to the execution time of other outer muscles rather than the deeper ones. $|f_c|$ has a different behavior as shown in figure 5.2. $|f_c|$ is the depth of the recursion tree in a D&C skeleton. Here, $WMA(10)$ offers better prediction, because the average value of $|f_c|$ is quite stable but the EMA prediction with higher α values is too sensitive to the last observation that might be too erratic.

Naive NQueens has a similar behavior. In NQueens, some of the explored states are much shorter, because they lead to no solution (or few solutions). Having too much weight on the last elapsed time is harmful. This is why WMA behaves considerably better than the other algorithms.

As stated before, the case of WordCount is seriously affected by the execution time of the first split that loads ∼100MB to main memory. However, this case shows a slight tendency to behave better with the algorithms that gives not too much relevance to the last value. Here, WMA behaves better than the EMA, because this benchmark produces a lot of IO exceptions with irregular waiting time, and WMA is less sensitive to irregular values than EMA.

5.4 Conclusions

From the analysis we can draw the following conclusions:

- EMA behaves better than the other algorithms evaluated on computation intensive muscles.
- WMA behaves better then the other algorithms evaluated on muscles with erratic behavior (e.g., IO intensive muscles).
- When using EMA, $\rho = 0.5$ and offers an average good behavior.
- When using WMA, $\alpha = 10$ and offers an average good behavior.
- Comparing WMA(10) and EMA(0.5), the first one offers a better result on average because the second performs badly on irregular use-cases, however, most of the time, EMA(0.5) performs slightly better than WMA(10).

Chapter 6

Perspectives and Conclusions

This chapter summarises the conclusions of this thesis and presents research perspectives that this research could open.

6.1 Conclusions

In this thesis I have shown how skeletons together with autonomic computing present a promising solution for the autonomic adaptation of parallel applications by the design and implementation of a novel approach using event driven programming.

It has been shown that the proposed approach contributes to the current solutions as follows:

- It is not related to a specific application architecture, like the component architecture. Thus, it has a broader scope, and is independent of the architecture used to implement skeletons.
- It allows for the introduction of structural changes during execution time. Therefore, it provides a higher level of adaptability compared with other solutions that only allows to introduce structural changes during compilation.
- The estimates of expected future work can be calculated on-the-fly, and it is not limited to pre-calculated estimates.

The feasibility of the approach has been shown by designing and implementing self-configuring and self-optimizing characteristics on the Skandium framework to guarantee a given execution time for a skeleton, by optimizing the number of threads allocated. The QoSs implemented as part of this thesis are: (1) Execution Wall Clock Time, and (2) Level of Parallelism.

This thesis has shown that the use of events allows for precise monitoring of the status of skeleton execution and permits the estimation of the remaining computation time.

Other contributions of this thesis are:

- a novel skeleton's separation-of-concerns based on event-driven programing. This approach allows the introduction of non-functional concerns to skeletons without lowering its higher-level programming.
- the evaluation of two different estimation strategies and a recommendation of which of them better estimates under what circumstances. Despite their simplicity the predictors analyzed are reliable enough and can be chosen easily depending on the application.

6.2 Research perspectives

This thesis discusses the QoSs level of parallelism and Wall Clock Time. As discussed by Aldinucci et al in [2], there are different QoS and non-functional concerns that are widely studied and incorporated as autonomic characteristics: e.g. dynamic load balancing, adaptation of parallelism exploitation pattern to varying features of the target architecture and/or application, among others. This research creates the basis for the design and implementation of different QoS in skeletons to improve their scalability and maintainability.

Another future research project based on this work will consist in designing autonomic decision procedures for deciding whether local execution with increased number of threads, distributed evaluation or a mix of the two will be the best for improving the performance or achieving a required QoS. This thesis shows that the framework proposed is convenient for this research, but the design of such a complex autonomic adaptation procedure requires additional work.

The proposed solution here is independent of the platform chosen for executing the skeleton. It has been illustrated in a multi-core setting, but it could also be adapted to a distributed execution environment by a centralized distribution of tasks to a distributed set of workers, adding or removing workers like adding or removing threads in a centralized manner. Taking decisions in a distributed manner would require more work. It is likely that a hierarchical distributed algorithm would be more feasible than a purely distributed one.

Bibliography

[1] M. Aldinucci and M. Danelutto. Securing skeletal systems with limited performance penalty: The muskel. *Journal of Systems Architecture - Embedded Systems Design*, 54(9):868–876, 2008.

[2] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, 2009.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, C. Montangero, and L. Semini. Managing adaptivity in parallel systems. In B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 199–217. Springer Berlin Heidelberg, 2013.

[4] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic qos in assist grid-aware components. In *14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2006)*.

[5] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1-2):5–24, 2009.

[6] F. Baude, L. Henrio, and P. Naoumenko. A Component Platform for Experimenting with Autonomic Composition. In *First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007). Invited Paper*. ACM Digital Library, Oct. 2007.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[8] D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010. ¡ce:title¿ICCS 2010¡/ce:title¿.

[9] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer Berlin Heidelberg, 2007.

[10] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[11] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[12] A. Collins, C. Fensch, and H. Leather. Auto-tuning parallel skeletons. *Parallel Processing Letters*, 22(2), 2012.

[13] D. created using Twitter4j (http://twitter4j.org). Source raw data used for the example on section 5: 1.2 million colombian twits from july 25th to august 5th of 2013. https://drive.google.com/file/d/0B_KljwYYwPn0S0Nob3NTX29XcHc, August 2013. [Online; accessed 7-January-2014].

[14] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[15] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006. 10.1007/11821946 6.

[16] M. Diaz, S. Romero, B. Rubio, E. Soler, and J. Troya. Adding aspect-oriented concepts to the high-performance component model of sbasco. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 21–27, Feb 2009.

[17] B. Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Doing hard time : developing real-time systems with UML, objects, frameworks and patterns / Bruce Powel Douglass. Addison-Wesley, 1999.

[18] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.

[19] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. Escala: Modular event-driven object interactions in scala. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[20] H. González-Vélez and M. Cole. Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. *Concurr. Comput. : Pract. Exper.*, 22(15):2073–2094, Oct. 2010.

[21] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[22] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 218–236. Springer Berlin Heidelberg, 2013.

[23] K. Hammond and G. Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, UK, 2000.

[24] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, 2001.

[25] R. Johnson, E. Gamma, J. Vlissides, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[26] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, Jan. 2003.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[28] M. Leyton. *Advanced Features for Algorithmic Skeleton Programming*. PhD thesis, l'Université de Nice - Sophia Antipolis, 2008.

[29] M. Leyton and J. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, 2010.

[30] O. Lobachev, M. Guthe, and R. Loogen. Estimating parallel performance. *Journal of Parallel and Distributed Computing*, 73(6):876 – 887, 2013.

[31] G. Pabon and M. Leyton. Tackling algorithmic skeleton's inversion of control. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 42–46, 2012.

[32] D. Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7):28–32, 2010.

[33] K. Potter, M. Smith, J. K. Guevara, L. Hall, and E. Stegman. It metrics: It spending and staffing report, 2011. Technical Report G00210146, Gartner, Inc., January 2011.

[34] C. Ruz, F. Baude, and B. Sauvan. Using components to provide a flexible adaptation loop to component-based soa applications. *International Journal on Advances in Intelligent Systems*, 5(1 2):32–50, July 2012. ISSN: 1942-2679.

[35] M. Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Electronics & Electrical. Taylor & Francis, 2009.

[36] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. Jescala: Modular coordination with declarative events and joins. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 205–216, New York, NY, USA, 2014. ACM.

[37] Wikipedia. Aspect-oriented programming - terminology, 2015. http://en.wikipedia.org/wiki/Aspect-oriented_programming#Terminology [Online; accessed 15-May-2015].

[38] Wikipedia. Exponential moving average, 2015. http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average [Online; accessed 3-May-2015].

[39] Wikipedia. Moving average, 2015. http://en.wikipedia.org/wiki/Moving_average [Online; accessed 3-May-2015].

[40] Wikipedia. Root-mean-square deviation, 2015. http://en.wikipedia.org/wiki/Root-mean-square_deviation [Online; accessed 3-May-2015].

[41] Wikipedia. Weighted moving average, 2015. http://en.wikipedia.org/wiki/Moving_average#Weighted_moving_average [Online; accessed 3-May-2015].