Politecnico di Milano
Dipartimento di Elettronica,Informatica e Bioingegneria
Master thesis

A distributed framework supporting runtime autotuning for HPC applications

M.Sc Thesis of:
Cristiano Di Marco

Advisor:
Prof. Cristina Silvano

Co-Advisors:
Prof. Gianluca Palermo
Eng. Davide Gadioli

Academic year 2016/2017

Abstract

Compute and data intensive problems, such as universe or microbiological studies, are pushing High Performance Computing architectures to achieve the Exascale level, that is the capability to process a billion billion calculations per second.

These applications manage huge inputs and they can be set up by multiple parameters that influence execution; since power consumption issues and energy efficiency have become essential, there exist various techniques that try to improve these aspects, keeping quality of results however acceptable: for instance, Approximate Computing strategies, both at software or hardware level, balance computation quality and expended effort, in order to respect applications goals and requirements and to maximize overall efficiency.

The Design Space of all possible configurations, for these kind of applications, is very wide and, therefore, it can't be explored exhaustively: having full knowledge about both parameter values and corresponding metric of interest results (such as, for instance, throughput, power consumption or output precision) is almost impracticable.

This thesis has focused on the development of a framework, Agorà,

that is able to drive online Design Space Exploration through an initial subset of configurations, with the aim to predict application complete model through Machine Learning techniques; the result is used by an autotuner to dynamically adapt program execution with the best configuration that satisfies current goals and requirements. Main Agorà advantages are the elimination of any offline phase nor design-time knowledge and the capability to manage multiple running applications at the same time.

Sommario

Applicazioni che si interessano di campi di ricerca complessi come, ad esempio, gli studi sull'universo o sulla microbiologia, stanno proiettando le architetture ad elevate prestazioni di calcolo (*High Performance Computing*) verso il traguardo di un miliardo di miliardi di operazioni effettuate al secondo.

Queste intricate ricerche sono caratterizzate da una moltitudine di dati in ingresso e dalla presenza di parametri che ne influenzano l'esecuzione; poichè problematiche inerenti il consumo di potenza e l'efficienza energetica hanno assunto un'importanza rilevante, esistono varie tecniche che tentano di migliorare questi aspetti, mantenendo comunque accettabile il valore dei risultati ottenuti: ad esempio, strategie di approssimazione (*Approximate Computing*), applicabili sia a livello hardware sia a livello software, ricercano un equilibrio tra la qualità della computazione e lo sforzo richiesto, al fine di adempiere ai vincoli e agli obiettivi dell'applicazione e di massimizzare, al contempo, l'efficienza generale.

Per questo tipo di applicazioni, lo spazio delle possibili configurazioni è molto ampio e, pertanto, non è possibile esplorarlo esaus-

tivamente: avere completa conoscenza riguardo le combinazioni dei parametri e i corrispondenti risultati delle metriche prese in esame (come, ad esempio, il numero di operazioni completate al secondo, il consumo di potenza oppure la precisione dei dati in uscita) è irrealizzabile.

Questa tesi ha sviluppato un sistema, Agorà, in grado di gestire l'esplorazione dello spazio delle configurazioni attraverso un sottoinsieme di esso, al fine di predire, attraverso tecniche di apprendimento automatico (*Machine Learning*), il modello completo dei programmi; questo risultato è usato da un cosiddetto *autotuner* per scegliere dinamicamente la migliore combinazione di parametri che soddisfa vincoli e obiettivi correnti dell'applicazione in gestione. I principali benefici apportati da Agorà sono l'eliminazione di ogni fase antecedente l'esecuzione dei programmi e la capacità di gestire molteplici applicazioni contemporaneamente.

Contents

CHAPTER 1

Introduction

1.1 Problem and Motivations

Nowadays, increasingly accurate climate forecasts, biomedical researches, business analytics, astrophysics studies or, in general, Big Data related problems require huge computing performance in order to obtain significant results; for these reasons, High Performance Computing technologies have been continuously sped up and, now, their next target is the Exascale level, that is the capability of at least one exaFLOPS, equal to a billion billion FLOPS (FLoating Point Operations Per Second), which is the order of processing power of the human brain at neural level, based on H. Markram et al. study ([30]).

To raise computing performances, multicore scaling era has pro-

duced, over time, frequency and power increase; it's impossible to follow this trend anymore, due to the beginning of the Dark Silicon era ([22]) and the failure of the Dennard Scaling ([21]): in 1974, Robert H. Dennard provided a more granular view of the famous Moore's Law ([33]) about the doubling of the number of transistors in a dense integrated circuit approximately every two years; this trend produced faster processors because, as transistors got smaller, their power density was constant, so that the power use was proportional with area: as transistors got smaller, so necessarily did voltage and current, giving the possibility to increase frequency. The ability to drop voltage and current, in order to let the transistors operate reliably, has broken down; static power (the power consumed when the transistor is not switching) has increased its contribution in the overall power consumption, with an importance similar to the dynamic power (the power consumed when the transistor changes its value), producing serious thermal issues and realistic breakdowns. This scenario implicates the impossibility to power-on all the components of a circuit at the nominal operating voltage due to Thermal Design Power (TDP) constraints.

Due to these physical problems, systems energy efficiency has become essential; even if, every approximately 1.5 years computation per kilowatt-hour have doubled over time ([28]), Subramaniam et al. investigation ([42]) suggests that there will be the need of an energy efficiency improvement at a higher rate than current trends, in order to reach the target consumed power of 20 MW for Exascale systems, established by DARPA report ([12]).

Efficiency has become very important also for electricity consumption of data centers: in fact, just the US data centers are expected to consume 140 billion kWh in 2020, from 61 billion kWh in 2006 and 91 billion kWh in 2013 ([1]), since the amount of information managed by worldwide data centers will grow 50-fold while the number

of total servers will increase only 10-fold ([24]).

It is straightforward therefore that green and heterogeneous approaches have to be applied in order to reach all these achievements where, for instance, multiple Central Processing Units (CPUs), General-Purpose computing on Graphics Processing Units (GPGPUs) and Many Integrated Cores accelerators (MICs) coexist and work together in a parallel system architecture; for these reasons, *TOP500 Green Supercomputers* ([4]) demonstrates the large interest in green architectures, ranking and updating the world top 500 supercomputers by energy efficiency.

There exists various approaches in order to deal with these problems, from both hardware and software point of view; for instance, Power Consumption Management consists of various techniques such as Dynamic Voltage and Frequency Scaling (DVFS) and Thermal-aware or Thermal-management technologies in order to deal with the Dark Silicon issue ([31]); another important concept is Approximate Computing ([32]), that focuses on balancing computation quality and expended effort, both at programming level and in different processing units; this thesis deals with this last technique, exploited at software level, oriented to tunable High Performance Computing applications that follow the so called *Autonomic Computing* research project ([26]).

The underlying structure on which this theory is based is the Monitor-Analyze-Plan-Execute feedback loop: these systems have the capability to manage themselves, given high-level objectives and application knowledge in terms of possible configurations, made by parameters values (such as, for instance, the number of threads or processes involved, possible various kind of compiler optimization levels, application-specific variables values, etc.) and the corresponding metrics of interest values (such as, for instance, power consumption, throughput, output error with respect to a target value, etc.); this

information assembles application Design Space.

Since, for these kind of programs, the multitude of parameters and their corresponding set of values makes Design Space huge and since, in general, computation time is not negligible, an exhaustive search is practically unfeasible, so there are Design Space Exploration techniques that aim at provide approximated Pareto points, namely those configurations that solve better than others a typical multi-objective optimization problem (for instance, keeping throughput above some value while limiting output error and power consumption within a prearranged interval).

Typically, this knowledge is built at design-time; after that, it is passed to a dynamic autotuner that has the ability to choose, from time to time, the best possible set of parameters values (also called *software knobs*) with respect to current goals and requirements that might change during execution.

One of the leading researches in this area is the ANTAREX project ([39]) that aspires to express by LARA ([15, 16]), a Domain Specific Language (DSL), the application self-adaptivity and to autotune, at runtime, applications oriented to green and heterogeneous HPC systems, up to the Exascale level.

1.2 Objectives

The main objective of this work is to fulfil applications Design Space Exploration at runtime, collecting all the information about used parameters values and related performances in terms of associated metrics of interest; through this data and Machine Learning techniques, we aim to predict applications complete model, made by the whole possible configurations list; finally the model is transmitted to the online dynamic autotuner, that can finally set up related program execution in the best way, according to current goals and requirements.

Agorà purpose is the ability to wisely manage multiple tunable applications that run inside a parallel architecture; we want not only to simultaneously superve different programs, but also to share Design Space Exploration phase among all those nodes that run the same application, hence speeding up this process.

Any kind of prior offline phase and design-time knowledge, that separates applications from their execution in runtime autotuning mode, is therefore avoided; Agorà makes autotuning completely independent from both application type and technical specifications about the machine in which program is executed; in fact, Agorà doesn't have to know this information before it starts and the final complete model will be suitable regardless autotuner objectives.

1.3 Thesis structure

[Da fare in ultimo, quando tutta la struttura della tesi è chiara]

CHAPTER 2

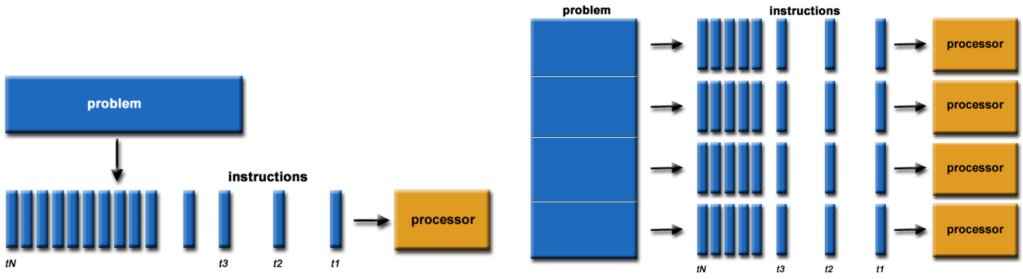
Background

In this chapter we explain the main used concepts and tools, related to tesiCris; as it has been revealed in the Introduction, this work has been thought for HPC applications that run in parallel architectures, so we start clarifying what are these target systems and how they are built; after that, we explain the concepts of Design Space Exploration and Design of Experiments, that represents the kernel of this work, in union with the idea of application dynamic online autotuning; finally, we present the used tools for the various communications among tesiCris components and for applications complete model prediction, highlighting their principal characteristics and features: the Message Queue Telemetry Transport (MQTT) messaging protocol and the Generalized Linear Regression interface by Apache Spark™ Machine Learning MLlib library.

2.1 Target architecture

High Performance Computing applications have complex and massive tasks to do, so they need huge computing power in order to accomplish their objectives.

Parallel computing ([10]) simultaneously uses multiple resources in order to solve a computational problem, taking advantage of concurrency, that is the possibility to execute, at the same time, those parts that are independent each other; in figure 2.1a we can see an example of serial computation, with a single processor that executes problem instructions sequentially, one after another; figure 2.1b instead represents a possible parallel computation, where problem is split in four parts that can be executed concurrently on different processors:



(a) Serial computation **(b)** Parallel computation

Figure 2.1: *Serial and parallel computation of a problem*

A typical parallel architecture is composed by an arbitrary number of computers, called *nodes*, each of them with multiple processors, cores, functional units, ect. connected all together by a network:

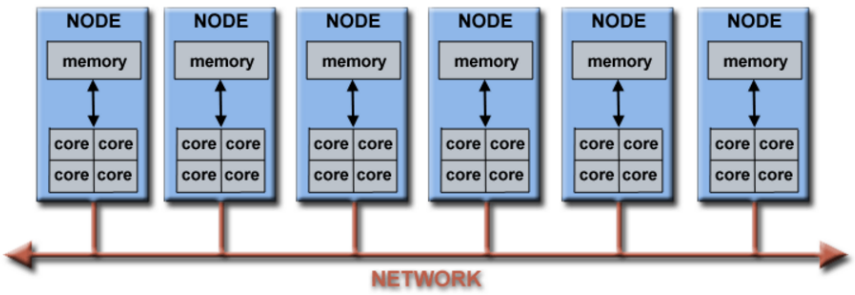


Figure 2.2: *Parallel architecture schema*

Inside a parallel architecture there can be heterogeneous nodes, with different computing techniques and configurations; according to Flynn's classical taxonomy, we can identify four different ways to classify computing architectures:

1. *Single Instruction, Single Data (SISD)*: this is the original kind of computer, in which there is only one instruction stream that is managed by the Central Processing Unit (CPU) during clock cycles; this is serial, so non-parallel, computing. Figure 2.3 shows a SISD execution example, in which every instruction is executed after the previous one:

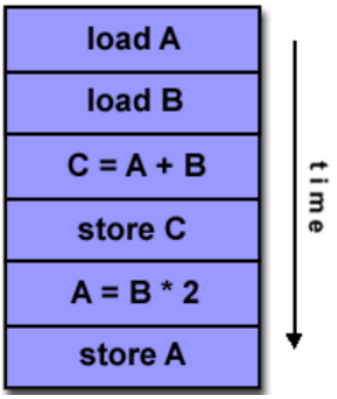


Figure 2.3: *Instruction execution in a SISD architecture example*

2. *Single Instruction, Multiple Data (SIMD)*: at any clock cycle, there is one instruction that is executed, but the processing units can operate on different data elements of the instruction; in figure 2.4 a classical SIMD execution is shown, where vector A and

B data are used to simultaneously modify vector *C*:

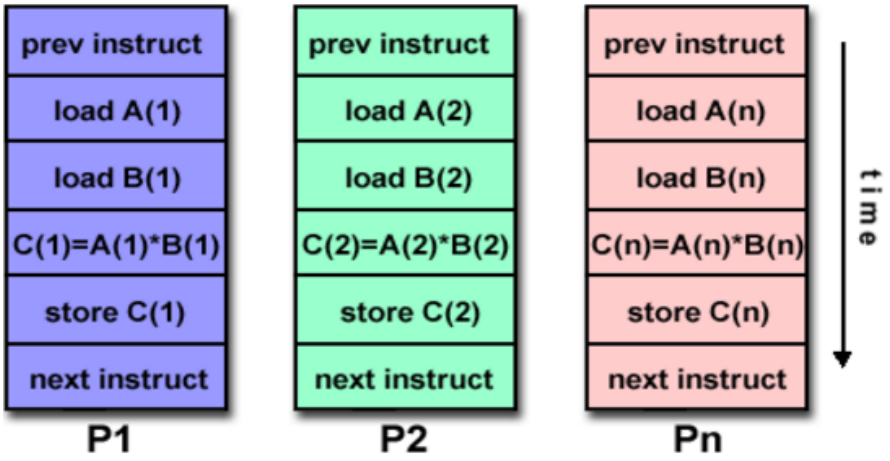


Figure 2.4: *Instruction execution in a SIMD architecture example*

3. *Multiple Instruction, Single Data (MISD)*: a single data stream is managed by multiple processing units, that can independently operate on data through separate instruction streams; figure 2.5 shows a MISD execution, where *A(1)* is simultaneously used to compute different tasks:

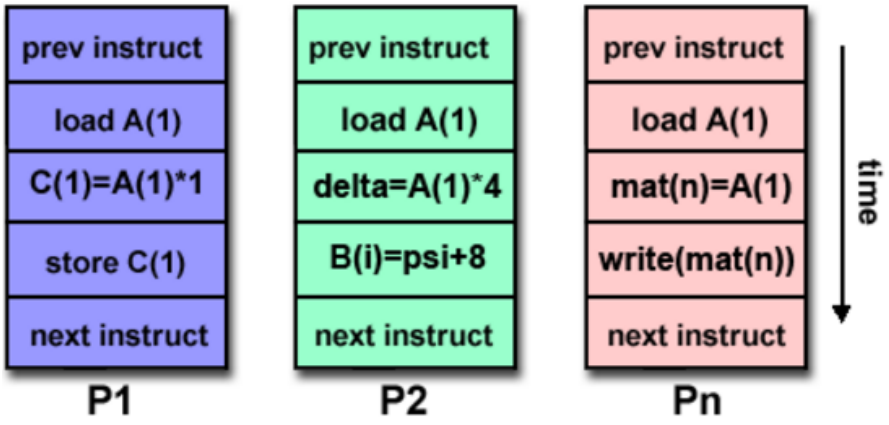


Figure 2.5: *Instruction execution in a MISD architecture example*

4. *Multiple Instruction, Multiple Data (MIMD)*: every processing unit can execute different data and instruction streams; in figure 2.6, processors concurrently elaborate different tasks that involve different elements:

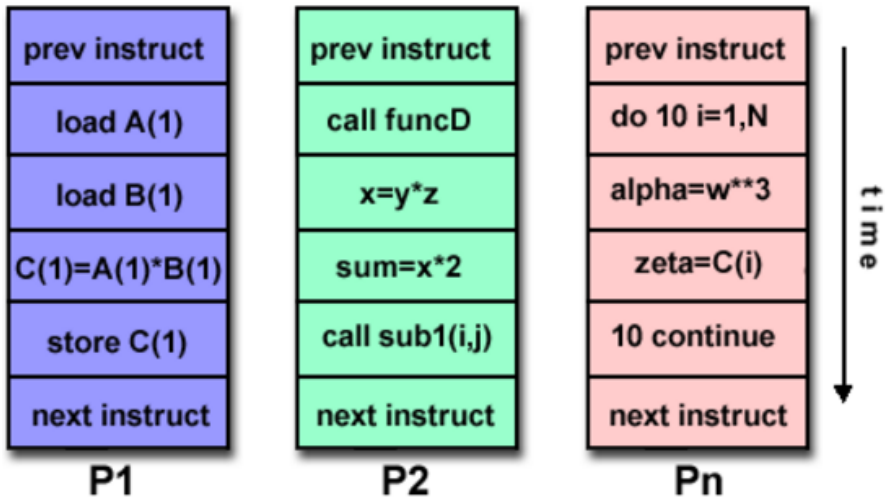


Figure 2.6: *Instruction execution in a MIMD architecture example*

Nowadays, among the introduced alternatives, MIMD architectures are the most used in parallel computing, especially in HPC systems.

2.2 Design Space Exploration

In many engineering problems there are several objectives that have to be obtained, with the presence of certain constraints and with the possibility to manage some customizable parameters; objectives can involve various metrics of interest, for instance connected to application throughput, system consumed power or overall cost.

Design Space Exploration (DSE) analyzes, in a systematic way, the space of possible parameters combinations that constitute application Design Space, with the objective to find the best design point that fulfills problem goals and requirements.

HPC applications have almost always a lot of parameters and related set of values, making the list of program configurations very long; in these cases the corresponding Design Space literally explodes, making impossible to analyze it in an exhaustive way.

2.2.1 Multi-Objective Optimization (MOO) problem

When there is more than one objective function that has to be accomplished, DSE consists of a Multi-Objective Optimization (MOO) problem; taking [14] as reference, in mathematical terms a MOO problem can be defined as:

$$min(f_1(x), f_2(x), ..., f_k(x)), \quad k \geq 2 \qquad s.t. \quad x \in X \tag{2.1}$$

where k denotes the number of objective functions and X represents the feasible region, defined by some constraints functions (for instance, in an embedded architecture, the total area should not exceed a predetermined value). If some of the objective functions have to be maximized, they can be attributed to minimizing its negation.

Multi-Objective Optimization problem has a lot of analogies into a wide variety of situations and domains, even the most common ones: formerly the ancients, given a set of seeds and a plot of land, had to choose what, how and how much farm, in order to maximize the harvest profit and to minimize the required effort, simultaneously not exceeding a cost limit; airline companies want to augment the number of passengers on their airplanes, to increase safety, to enlarge autonomy of their vehicles by means of various technical, strategic and commercial choices; an undergraduate student would minimize his/her university career duration and maximize his/her exam average, with a predetermined time limit and with the possibility to choose some coursers than others.

As it can be understood, since some objectives are in contrast with other objectives, a unique solution doesn't exist; for instance, in a microcontroller, an objective focused on performance would definitely confront against a power consumption one: the best solution for one of them would be the worst for the other and conversely. So, the aim of Design Space Exploration is almost always to search for a meeting

point, between all the goals and requirements, that fulfills the overall problem.

Since therefore the concept of unique optimal solution can't be applied, it is useful to introduce the notion of Pareto optimality; pareto optimal solutions are, essentially, those ones that can't be improved without degrading at least one objective function. So, a solution x^1 is said to *(Pareto) dominate* a solution x^2 if:

$$\begin{cases} f_i(x^1) \leq f_i(x^2) \quad \forall i \in \{1, 2, ..., k\} \\ f_j(x^1) < f_j(x^2) \quad \textit{for at least one } j \in \{1, 2, ..., k\} \end{cases} \tag{2.2}$$

The set of Pareto optimal solutions is often called Pareto front, Pareto frontier or Pareto boundary.

2.3 Design of Experiments

When a Design Space of an application is huge and, consequently, there is no possibility to do an exhaustive search over all possible configurations, there is the need to take a subset of points of interest that represent as closely as possible system behavior.

Therefore, on the one hand there is the quality of representation, that should be reliable enough; on the other, the number of simulations to do, that should be small.

Taking [34] as reference, among various DoE techniques that generate the initial set of design points to be analyzed, we can mention:

- *Full-Factorial DoE*: it is made by all possible combinations among parameters values, so all possible application configurations are picked up;

- *2-Level Full-Factorial DoE*: suitable for designs with two or more parameters, this DoE picks up all possible combinations among the extreme values of all parameters.

If, for instance, there are three tunable parameters:

1. Number of Processors $\in$ { 2, 4, 8, 16, 32 };
2. Number of Threads $\in$ { 1, 2, 3, 4, 5, 6, 7, 8 };
3. Cache size $\in$ { 2K, 4K, 8K, 16K, 32K }.

the design points will be: $\langle$ #processors, #threads, cache size $\rangle$ $\in$

{ $\langle$ 2, 1, 2K $\rangle$, $\langle$ 32, 1, 2K $\rangle$, $\langle$ 2, 8, 2K $\rangle$, $\langle$ 32, 8, 2K $\rangle$, $\langle$ 2, 1, 32K $\rangle$, $\langle$ 32, 1, 32K $\rangle$, $\langle$ 2, 8, 32K $\rangle$, $\langle$ 32, 8, 32K $\rangle$ };

- *Face Centered Central Composite DoE with one Center Point*: also this DoE is appropriate for designs with two or more parameters; the list of its design points can be split in three sets:

  1. A 2-Level Full-Factorial Design set;
  2. A Center Point, in which each value is the median value of the corresponding parameter;
  3. An Axial Points set, in which all the median and extreme values of each parameter are combined.

  Considering the example in the previous DoE, the final design points list would be: $\langle$ #processors, #threads, cache size $\rangle$ $\in$

  { $\langle$ 2, 1, 2K $\rangle$, $\langle$ 32, 1, 2K $\rangle$, $\langle$ 2, 8, 2K $\rangle$, $\langle$ 32, 8, 2K $\rangle$, $\langle$ 2, 1, 32K $\rangle$, $\langle$ 32, 1, 32K $\rangle$, $\langle$ 2, 8, 32K $\rangle$, $\langle$ 32, 8, 32K $\rangle$ } $\cup$

  { $\langle$ 8, 4, 8K $\rangle$ } $\cup$

  { $\langle$ 2, 4, 8K $\rangle$, $\langle$ 32, 4, 8K $\rangle$, $\langle$ 8, 1, 8K $\rangle$, $\langle$ 8, 8, 8K $\rangle$, $\langle$ 8, 4, 2K $\rangle$, $\langle$ 8, 4, 32K $\rangle$ };

- *Plackett-Burman DoE*: this DoE might be useful to analyze, more eonomically, a larger number of variables; in fact, Plackett-Burman

reduces the number of potential factors, constructing very economical designs with the number of points multiple of 4 (rather than power of 2, as in the 2-Level Full-Factorial DoE).

Concerning the example above, in this case the final design points list would be: $\langle$ #processors, #threads, cache size $\rangle \in$

{ $\langle$ 2, 1, 32K $\rangle$, $\langle$ 32, 1, 2K $\rangle$, $\langle$ 2, 8, 2K $\rangle$, $\langle$ 32, 8, 32K $\rangle$ };

- *Latin-Hypercube DoE*: this DoE randomly chooses parameters values for each design point; the number of final configurations can be set up in advance.

2.4 Dynamic autotuning

When applications expose some configurable parameters (also known as *dynamic knobs*), the concept of dynamic autotuning is defined as the ability to find the best set of knobs values, in an automatic and systematic way, that satisfies application goals and requirements at runtime, properly reacting to possible objective functions change. For instance, a web video streaming application would be able to manage video quality according to the overload of its servers: in some situations, it could set up itself for the best possible resolution; sometimes, it should reduce quality in order to make still available its services to all connected clients.

IBM research studies on *Autonomic Computing* ([26], [20]) made a breakthrough on the concept of self-adapting systems that are able to manage themselves and to dynamically optimize their execution configuration at runtime; *Autonomic Computing* is based of the MAPE-K control loop, showed in figure 2.7:

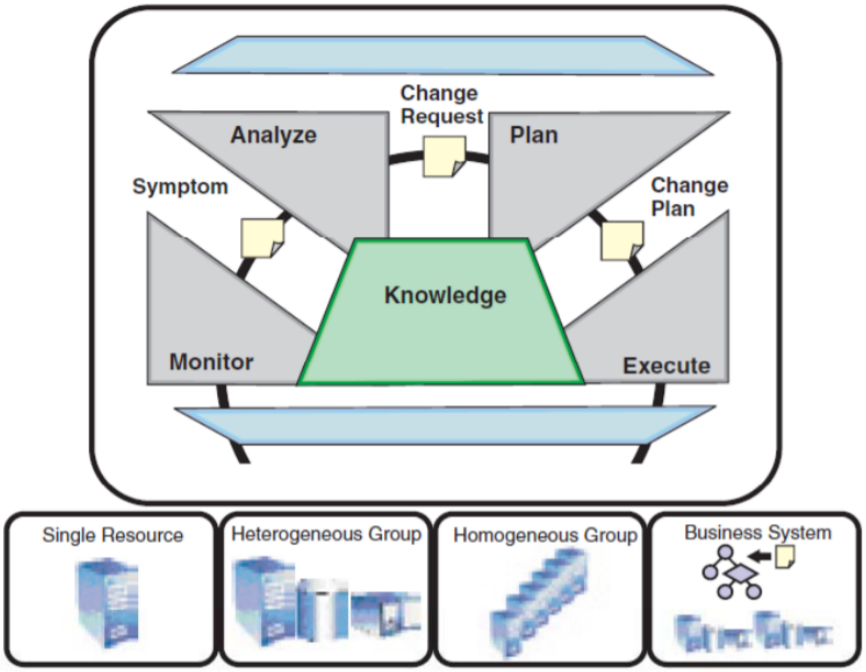


Figure 2.7: *MAPE-K control loop design*

In this control loop we can identify four principal functions:

- *Monitor*: it gathers application information about knobs setting and the associated metrics of interest values as, for instance, throughput and consumed power;
- *Analyze*: it performs data analysis and reasoning on the information provided by the monitor function;
- *Plan*: it reacts to a possible application objectives change during execution and it structures the actions needed to achieve this new state;
- *Execute*: it modifies the behavior of the managed resource, according to the actions recommended by the plan function.

Finally, *Knowledge* source is composed by all data that is used by the autonomic manager's four functions; it includes information such as, for instance, topology structure, historical logs, metrics and policies.

Online autotuning therefore entrusts systems management from people to technology, achieving self-configuration and self-optimization objectives; applications requirements may change during execution and the overall system is able to react properly and to re-adapt itself.

2.5 MQTT messaging protocol

MQTT (Message Queue Telemetry Transport, [9]) is a lightweight messaging protocol that gives the possibility to establish remote communications among subjects. Its main characteristics are the minimization of network bandwidth and devices requirements but also the attention to the assurance of delivery; this features made MQTT ideal for machine-to-machine (M2M) or Internet of Things (IoT) world of connected devices, but in general this protocol have had a large use in different projects: for instance, the famous Facebook Messenger is built on top of it ([47]).

MQTT uses a publish/subscribe pattern, instead of the classical client/server model: a client has the possibility to subscribe to topics and to publish messages on them (both topics and messages are strings); another component, called *broker*, deals with the dispatch of messages to only those clients that have subscribed to corresponding topics; so, publishers (those clients that sends messages) and subscribers (those clients that receive messages) don't know about the existence of one another: the broker, which is known by every client, will distribute messages accordingly. Figure 2.8 shows a possible MQTT scenario with a sensor and two devices:

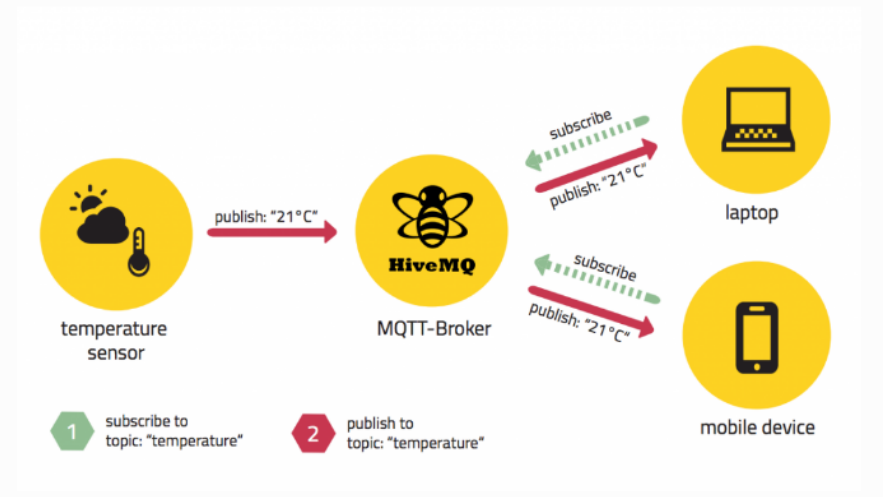


Figure 2.8: *MQTT publish/subscribe example, taken from [2]*

Topics are used by the broker to filter messages and manage them in a correct way; they are made up of one or more levels, separated by a forward slash, as shown in figure 2.9:

Figure 2.9: *MQTT topic example, taken from [2]*

There is the possibility to subscribe to more topics at once through the use of wildcards: the single-level one (denoted with the symbol +) and the multi-level one (indicated with the symbol #).

Single-level wildcard substitutes an arbitrary level in a topic, so all topics that matches the same structure will be associated to the one with single-level wildcard:

Figure 2.10: *MQTT topic with single-level wildcard example, taken from [2]*

for instance, *myhome/groundfloor/kitchen/temperature* and *myhome/
groundfloor/livingroom/temperature* will match the topic in figure 2.10, while *myhome/groundfloor/kitchen/humidity* won't.

Multi-level wildcard is placed at the end of a topic and it covers an arbitrary number of topic levels:

Figure 2.11: *MQTT topic with multi-level wildcard example, taken from [2]*

in this case, for instance, *myhome/groundfloor/kitchen/temperature* and *myhome/groundfloor/kitchen/humidity* will match the topic in figure 2.11, while *myhome/firstfloor/livingroom/temperature* won't.

Another interesting MQTT feature is the Last Will and Testament (LWT): each client can specify a normal MQTT message with topic and payload. When it connects to the broker, this message is stored; if client abruptly disconnects, the broker sends the corresponding LWT to all subscribed clients on related topic, notifying the occurred disconnection.

2.6 Apache Spark^{TM} MLlib library

tesiCris takes advantage of the Machine Learning MLlib library by Apache Spark™ ([41]) in order to predict the complete model of running applications; in particular, it has been focused on the Generalized Linear Regression interface.

2.6.1 (Generalized) Linear Regression

Taking [3] as reference, Linear Regression tries to model the relationship between a variable *y* and one or more variables x_1, x_2, ... x_n, $n \geq 1$; more rigorously, given a set of statistical units $\{y_i, x_{i1}, x_{i2}, ..., x_{ip}\}_{i=1}^{n}$, in which y_i is the variable that depends on the p-vector $[x_{i1}, x_{i2}, ..., x_{ip}]$, linear regression assumes that this relationship is linear:

$$y_i = \beta_0 \mathbf{1} + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \epsilon_i, \quad i = 1, 2, ...n \tag{2.3}$$

y_i is the *regressand*, *endogenous variable*, *response variable*, *measured variable*, *criterion variable* or *dependent variable*;

$x_{i1}, x_{i2}, ..., x_{ip}$ are called *regressors*, *exogenous variables*, *explanatory variables*, *covariates*, *input variables*, *predictor variables* or *independent variables*;

$\beta_0, \beta_1, ..., \beta_p$ are the *effects* or *regression coefficients*, whose values establish the connection between the regressand and the regressors; β_0 is also called *intercept*; linear regression mainly focuses on the estimation of these parameters;

ϵ_i is called the *error term*, *disturbance term* or *noise*. It represents all other factors that influence y_i other than the predictor variables.

Linear Regression assumes that the response variable follows a Gaussian distribution; Generalized Linear Regression (GLR) gives the possibility to specify other distributions taken from the exponential family; it is useful for several kinds of prediction problems including Linear Regression, Poisson Regression (for count data) and

Logistic Regression (for binary data); moreover, GLR gives the possibility to specify a link function g that relates the mean of the measured variable to the exogenous variables. In the case of Gaussian distribution for Linear Regression, the link function can be equal to *Identity*, *Log* and *Inverse*, with an explicit meaning.

2.6.2 Regressors transformations

In order to use Linear Regression even if the relationship between the dependent variable and the independent variables is not linear, there is the necessity to modify the input variables; tesiCris implements two possible regressors transformations:

1. it transforms parameters values with inverse, square root and natural logarithmic functions; in union with the unaltered predictor variables, it tries to find the best model combining these transformations and the available link functions;

2. it transforms parameters values through the polynomial expansion of second order: their cross-products and square values are added to the set of regressors, evaluating the best model with the available link functions.

tesiCris chooses the model with the smallest Akaike Information Criterion (AIC) value, that is a measure of the quality, in terms of lost information, of statistical models for a given set of data; at the same AIC value, the chosen model is the one with the smallest mean of the sum of coefficient standard errors, that measure how precisely the model has estimated the regression coefficients with the given training set of data.

2.7 Thesis structure

[Da fare in ultimo, quando tutta la struttura della tesi è chiara]

CHAPTER 3

State of The Art

In this chapter we introduce all those works that have been a starting point for the design and implementation of tesiCris; we divide them in two subcategories: the former is oriented to Design Space Exploration, the latter on autotuning.

3.1 Design Space Exploration related works

ReSPIR ([36]) proposes a DSE methodology for application-specific multiprocessor systems-on-chip (MPSoCs). First, a Design of Experiments phase is used to capture an initial plan of experiments that represent the entire target Design Space to be explored by simulations; after that, Response Surface Methodology techniques ([27]) identify the feasible solution area with respect to the system-level

constraints, in order to refine the Pareto configurations until a predetermined criterion is satisfied.

MULTICUBE Explorer ([40]) is an open source, portable, automatic Multi-Objective Design Space Exploration framework for tuning multi-core architectures; a designer, through an existing executable model (use case simulator) and a Design Space definition XML file, can easily explore his parametric architecture.

ϵ-Pareto Active Learning (ϵ-PAL, [48]) aims at efficiently localize an ϵ-accurate Pareto frontier in Multi-Objective Optimization problems; it models objectives as Gaussian process models, in order to guide the iterative design evaluation and, therefore, to maximize progress on those configurations that are likely to be Pareto optimal.

ReSPIR and MULTICUBE are researches oriented on application-specific architectures design, while ϵ-PAL deals with the MOO problem in a general way; all these three works aims to obtain a Pareto front and their execution is done offline; tesiCris uses the concept of Design Space Exploration but it is focused on Approximate Computing software strategies in executing applications; tesiCris doesn't calculate Pareto frontier, but its goal is to provide complete applications model through Machine Learning techniques; moreover, with respect to ReSPIR, MULTICUBE and ϵ-PAL, we want to avoid any offline DSE phase, driving it during programs execution.

Furthermore, at the best of our knowledge, tesiCris is the first framework that is able to fulfill application Design Space Exploration in a shared way, among simultaneously running applications; with this improvement, DSE executing time is considerably reduced.

3.2 Autotuning related works

SiblingRivalry ([7]) proposes an always online autotuning framework that uses evolutionary tuning techniques ([19]) in order to adapt parallel programs. It eliminates the offline learning step: it

divides available processor resources in half and it selects two configurations, a "safe" one and an "experimental" one, according to an internal fitness function value; after that, the online learner handles the identical current request in parallel on each half and, according to the candidate that finishes first and meets target goals and requirements, it updates its internal knowledge about configurations just used. This technique is used until a convergence is reached or when the context changes and, therefore, new objectives have to be achieved. When objectives change, SiblingRivalry restarts its procedure until new result is obtained; tesiCris doesn't ground its worklow on predetermined objectives: it is completely uninteresting about application goals and requirements, since it predicts the complete model for all metrics under examination; furthermore, computational power of the machine in which the tunable program is executed is not kept busy by tesiCris, since application information gathering and model prediction are done remotely, on a different computer.

Capri framework ([43]) focuses on control problem for tunable applications which mostly use Approximate Computing ([32]) as improvement technique; given an acceptable output error, it tries to minimize computational costs, such as running time or energy consumption. There are two main phases: training, done offline, estimates error function and cost function, using machine learning techniques with a training set of inputs; the control algorithm, done online, finds the best knobs setting that fulfills objectives. Also tesiCris is oriented in tuning applications based on Approximate Computing techniques; it is focused on everything that precedes the control phase, supplying, to the dynamic online autotuner, metrics of interest estimations for each possible application configuration; tesiCris wants to eliminate any offline phase, giving the possibility to simply run an application, driving its execution in order to collect a training set for model prediction; finally, the result is transmitted to applica-

tion autotuner, that is in charge of managing the control phase.

mARGOt ([23]) proposes a lightweight approach to application runtime autotuning for multicore architectures; it is based on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop ([26]), made up of a complex and varied monitors infrastructure and an Application-Specific RunTime Manager (AS-RTM), based on Barbeque work ([11]): the first captures runtime information that is used by the latter in order to tune application knobs, together with design-time knowledge and application multi-objective requirements. The user has to provide XML configuration files in which a list of mARGOt Operating Points, desired monitors and application objectives are expressed; the AS-RTM module, starting from this design-time knowledge and evaluating runtime information, has the task of choosing, from time to time, the best application configuration that satisfies application goals and requirements as best as possible. This work has been the starting point for the conception of tesiCris framework; mARGOt has to have the list of program configurations before execution: we want to produce this information while applications are running, removing to users the burden to make an offline step before taking advantage of autotuner capabilities.

There are other different autotuning frameworks in HPC context, yet based on design-time knowledge: OpenTuner ([6]) is able to build up domain-specific program autotuners in which users can specify multiple objectives; ATune-IL ([38]) is an offline autotuner that gives the possibility to specify a wide range of tunable parameters for a general-purpose parallel program; PetaBricks ([5]) is oriented on the definition of multiple algorithms implementations to solve a problem; [44] proposes a scalable and general-purpose framework for autotuning compiler-generated code, combining Active Harmony's parallel search backend ([18]) and the CHiLL compiler transformation framework ([17]); Green ([8]) is focused on the energy-conscious

programming using controlled approximation; PowerDial ([25]) is a system that transforms static configuration parameters into dynamic knobs in order to adapt applications behavior with respect to the accuracy of computation and the amount of resources. tesiCris main difference is the absence of any kind of prior information about application features and the indifference on quality and quantity of metrics and objectives, in contrast with the last two mentioned works ([8] and [25]), focused on energy and accuracy goals.

Finally, among libraries for specific tasks, OSKI ([45]) provides a collection of low-level primitives that automatically tunes computational kernels on sparse matrices; SPIRAL ([37]) generates fast software implementations of linear signal processing transforms; ATLAS ([46]) builds up a methodology for the automatic generation of highly efficient basic linear algebra operations, focusing on matrix multiplications. tesiCris aims to generalize autotuning process.

CHAPTER 4

Proposed methodology

A tunable application is characterized by the presence of specific parameters, also known as *knobs*, that influence program execution; their change produces different application results in terms of metric of interest values, as, for instance, throughput or power consumption. Figure 4.1 shows a typical parallel architecture in witch several tunable applications are running:

Figure 4.1: *An example of a parallel architecture with several executing tunable applications*

Very often, High Performance Computing applications expose a large set of parameters, making related Design Space huge and, consequently, unrealistic to explore it in an exhaustive way; in order to choose, from time to time, best program setting with the aim to improve energy efficiency with respect to power consumption and current input data, the concept of runtime autotuning is used: a class of online autotuners is able to choose, from time to time, best possible parameter values that fullfil application goals and requirements, starting from a design-time knowledge that gives information about parameter values and corresponding metric of interest values, built off-line. Figure 4.2 shows an application interconnected with mARGOt [23], a dynamic autotuner developed by our research group:

Figure 4.2: *Tunable application with the assistance of mARGOt autotuner schema*

As we can understand, this kind of autotuner needs application knowledge, so there have to be a preceding phase, before program start of computation, in which it is made; our improvement is to avoid this off-line step, building, managing and updating application knowledge during execution itself. A local module mainly takes care of properly setting application knowledge, while a remote one manages collected information during execution, in order to predict complete application model. Figure 4.3 shows an application interconnected with Agorà and mARGOt autotuner:

Figure 4.3: *Tunable application with the assistance of Agorà plus mARGOt autotuner schema*

This work, therefore, wants to address the problem of managing possible multiple applications that run, at the same time, inside a parallel architecture; the objective is to initially drive programs execution with a subset of parameters configurations taken from their Design Space, in order to gather all metrics of interest values associated to them; this list will compose the training set for the prediction of the complete model through Machine Learning techniques. Agorà can also correctly manage possible features; a feature is a particular application element than can not be set up like software knobs, but it contributes to the estimation of complete model; during DSE phase, feature values are observed like metric values while, during model prediction, they are considered as parameters, so their observations

take part to the estimation of metric of interest values.

The typical architecture in which tesiCris works is a parallel one, where there are multiple nodes, potentially heterogeneous, that execute applications; principal tesiCris strenghts are:

1. the ability to drive Design Space Exploration in a distributed way, among all those nodes that are running the same program, in order to considerably reduce DSE phase and to speed up overall workflow;

2. the ability to manage multiple kinds of applications, each of them separately organized by a dedicated tesiCris module that is in charge of all the nodes that execute the same program;

3. the out-of-band activity from the parallel architecture data streams: the computation of Design of Experiments configurations, the collection of associated metrics of interest values and the complete models prediction are done in a separate node with respect to the ones that run applications inside the architecture, while the exchange of information is done using the lightweight MQTT protocol (discussed in chapter 2.5);

4. the persistence of generated knowledge: once the complete model of an application is predicted, it is stored so, at any time, it can be reloaded and it is sent to new nodes that have started running the same application, without repeating all the workflow through which the complete model has been previously predicted;

5. the capability of being fault-tolerant, with respect both to a running node crash and to the interruption of the machine that has the objective to predict applications complete model: if the former situation happens, tesiCris has to properly handle the remaining running nodes; if the latter situation happens, running nodes inside the parallel architecture doesn't have to stop their

execution but they react properly, according to their internal state at that moment.

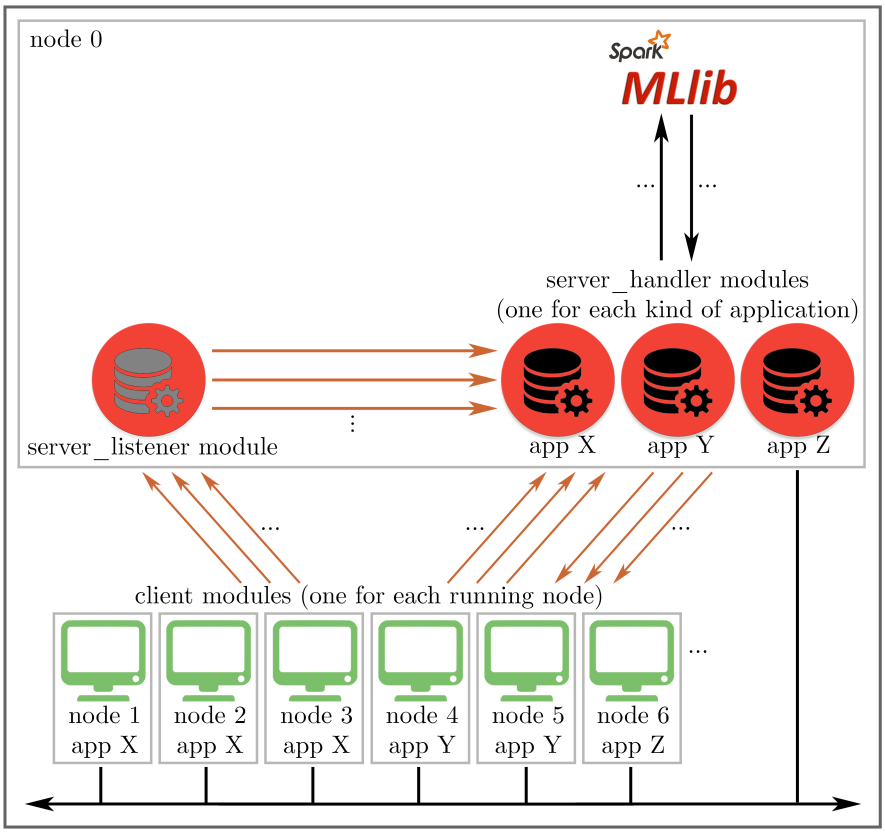


Figure 4.4: *Agorà overview in a parallel architecture*

Figure 4.4 shows all tesiCris components and a possible scenario with a parallel architecture in which six nodes are running three different applications; for every kind of application there exists a dedicated server_handler module that manages it; orange arrows represent all possible communications among modules, made possible through MQTT subscriptions and publications on predetermined topics.

tesiCris main components are:

- the *server_listener* module, written in Python: it keeps waiting for programs arrival, in order to properly manage them;

- the *server_handler* module, written in Python: it is created by the server_listener for every kind of application; it asks for application information such as, for instance, all parameters name and values; it computes application configurations that compose the Design of Experiments; it drives Design Space Exploration phase, distributing DoE configurations among all the nodes it manages; it collects parameters values and the observed metrics of interest sent by running programs; it makes use of Machine Learning techniques in order to build the complete application model; it sends the result to connected nodes;

- the *client* module, written in C++: it is set up in every executing program; it communicates with the autotuner that manages application behavior; it notifies the existence of the running machine to the server_listener module; it replies to possible information request made by the related server_handler module; during Design Space Exploration phase, it receives configurations from the server_handler module, it pass them to the autotuner and, after the application has done computation, it sends back all the obtained information, regarding parameters values and the associated metrics values; it saves the complete model received from the server_handler module, in order to properly set up the autotuner.

Principal use cases that define framework workflow and the interaction among components are the following:

1. application start of execution: when programs start running, the related client module notifies their existence to the server_listener module; if the application is unknown, the server_listener creates a dedicated server_handler module that is in charge of managing it, otherwise it communicates the new node to the corresponding existing server_handler;

2. Design of Experiments computation: the server_handler module computes the subset of configurations, from the entire application Design Space, that compose the Design of Experiments; after that, it is ready to drive Design Space Exploration phase, distributing these configurations to requesting nodes;

3. configuration reception: client module communicates to the autotuner all the configurations that, from time to time, are sent by the server_handler module; when computation is finished, it sends back to the server_handler module a list of values, made by the configuration just used with the observed metrics of interest;

4. application configurations and related metrics values collection: the server_handler module collects all the information it receives from running nodes; when it has all the necessary data, it uses Machine Learning techniques in order to predict the application complete model, made by all possible configurations associated with the predicted metrics values;

5. predicted model dispatch: the complete model is sent by the server_handler to the nodes; the client_module set the autotuner with this information, so the application can be sets up with the best configuration that fulfils current goals and requirements.

The interaction among tesiCris components have been implemented in an asynchronous way: programs executions are independent from the exchange of MQTT messages and all modules properly react to these events, in order to not condition application workflow and to not steal execution time, making all process as flowing as possible.

CHAPTER 5

Agorà: proposed framework

In this chapter, technical implementation of the proposed framework is illustrated, dividing the overall workflow in all possible use cases; in the last part, a sketch application shows how tesiCris client module is integrated and how it works.

5.1 Introduction

tesiCris implements all possible situations that can happen in a scenario as the one described in the previous chapter (**??**).

The MQTT protocol ([9]) makes possible all the communications among components: tesiCris uses Eclipse Paho MQTT Python Client and Eclipse Paho MQTT C Client for managing messages exchange ([35]), while it uses Eclipse Mosquitto as broker ([29]). The Machine

Learning library MLlib by Apache Spark™ ([41]) is used to predict the applications complete models.

The next technical use cases implementation makes use of application *Swaptions* as reference, taken from the PARSEC benchmark suite ([13]). This application is a workload which prices a portfolio of swaptions through Monte Carlo simulations; it has two tunable parameters, the number of threads (variable *num_threads*, from 1 to 8) and the number of trials for the simulation (variable *num_trials*, from 100.000 to 1.000.000 with a step of 100.000), while metrics of interest are throughput (variable *avg_throughput*), as the number of priced swaptions per second, and error (variable *avg_error*), computed as:

$$avg\_error = \frac{\sum_{s \in pricedSwaptions} |StandDevRef(s) - StandDev(s)|}{|pricedSwaptions|}$$

where $StandDevRef(s)$ is the reference standard deviation for swaption s, $StandDev(s)$ is the evaluated one and $pricedSwaptions$ represents all swaptions that are priced every computing cycle; so, metric avg_error stands for the average of differences between standard deviation of priced swaptions using evaluated configuration with respect to the reference one (standard deviation for 1.000.000 trials).

tesiCris has been interconnected to the mARGOt autotuner ([23]), that exploits design-time knowledge to dynamically adapt applications behavior during execution; mARGOt represents this information as a list of Operating Points (OPs): an OP is made by a set of parameters values, also called software knobs, in union with the associated performance (metrics of interest values), profiled at design-time; tesiCris improvement is to build application knowledge at run-time, with an online distributed Design Space Exploration phase in which a subset of OPs are collected, in order to predict the complete

model, made by the entire Operating Point list.

tesiCris could work in union with other autotuners that, using application knowledge in terms of configurations and associated performances, have the capability to dynamically adapt applications behavior during execution.

5.2 Use case implementation

5.2.1 server_listener module creation

The starting point is the creation of the server_listener module, that is in charge of managing the arrival of applications; it connects to the MQTT broker and it subscribes to topic "tesiCris/apps":

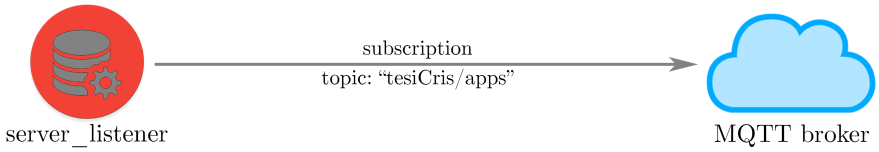


Figure 5.1: *server_listener subscription*

5.2.2 Application arrival

An application can be already known by the server_listener module or a program is executed by a machine for the first time.

Unknown application

A node starts running a program; the related client module notifies this event, publishing on topic "tesiCris/apps" a string composed of the application name and the machine hostname plus the Process IDentifier (PID), with format "*[appName] [hostname]_[PID]*", so that the client can be univocally recognized in the future; the message is dispatched to the server_listener, that creates a dedicated server_handler for that application:

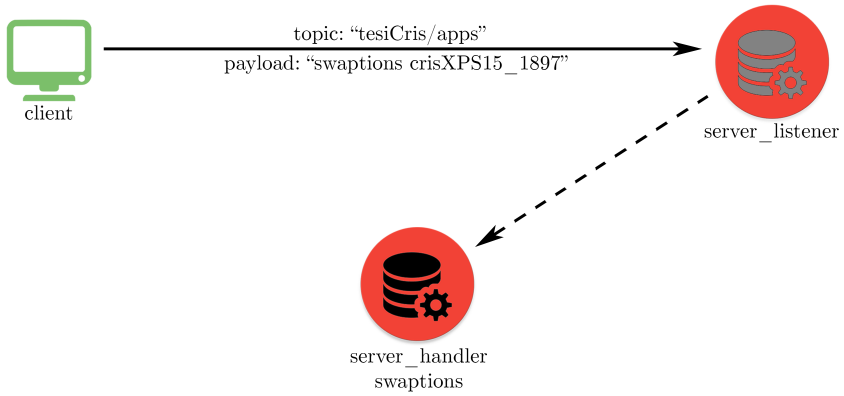


Figure 5.2: *New unknown application arrival example; a dedicated server_handler is created by the server_listener*

At the beginning, the client module subscribes to some topics that are needed to receive communications from the related server_handler:

1. "tesiCris/*[appName]*", in order to understand if the server_handler has asked application information and, therefore, to reply (see 5.2.3); this topic is also used to understand if the server_handler has crashed and, so, to react properly (see 5.2.7);

2. "tesiCris/*[appName]*/*[hostname]_[PID]*/conf", in order to receive configurations from the server_handler during Design Space Exploration phase (see 5.2.3);

3. "tesiCris/*[appName]*/*[hostname]_[PID]*/model", in order to receive a partial OP list (see 5.2.3) and the complete predicted model from the server_handler (see 5.2.3).

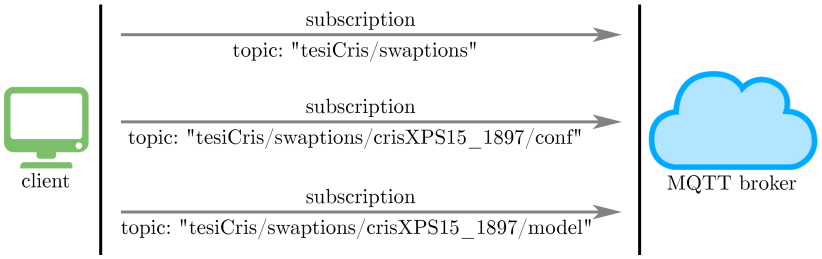


Figure 5.3: *client MQTT subscriptions example*

The server_handler subscribes to some topics in order to manage correctly all the various situations that happens:

1. "tesiCris/*[appName]*/newHostpid", in order to manage the hypothetical arrival of other clients that are running the supervised application (see 5.2.2);

2. "tesiCris/*[appName]*/req", in order to manage all the requests made by clients during program execution (see 5.2.3);

3. "tesiCris/*[appName]*/info/#", in order to receive all the available application information, such as parameters name and values (see 5.2.4); the real topic will be with format "tesiCris/*[appName]*/info/*[hostname]_ [PID]*", therefore the server_handler can store the ID of the node that is sending application information, in order to react properly to a possible client crash during this phase (see 5.2.6);

4. "tesiCris/*[appName]*/disconnection", in order to correctly react to a possible node disconnection (see 5.2.6);

5. "tesiCris/*[appName]*/OPs", in order to receive Operating Points from clients during Design Space Exploration phase (see 5.2.5).

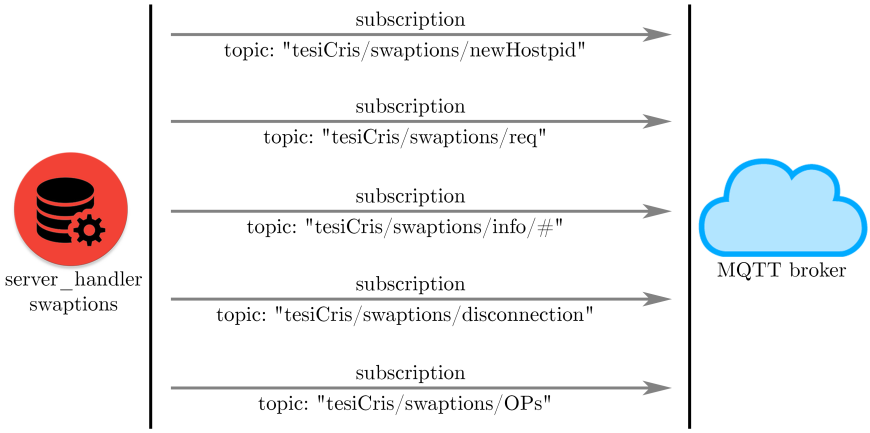


Figure 5.4: *server_handler MQTT subscriptions example*

Known application

When the server_listener is informed that a new node has started running an application but there is already a server_handler that is managing that program, it publishes on topic "tesiCris/*[appName]*/newHostpid" the new machine hostname plus PID, so that the corresponding server_handler can add the node to the pool of machines that are running the application it is supervising:

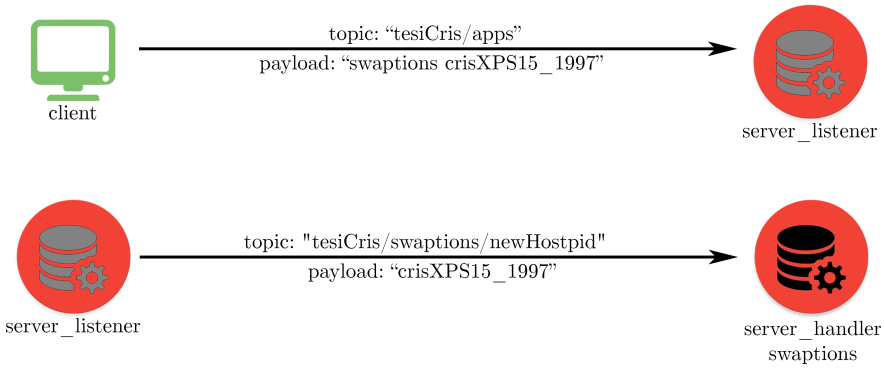


Figure 5.5: *New known application arrival example; client ID is sent by the server_listener to the corresponding server_handler*

5.2.3 Client request

At each predetermined time interval, clients make a request to their application server_handler, publishing their hostname plus PID on topic "tesiCris/*[appName]*/req":

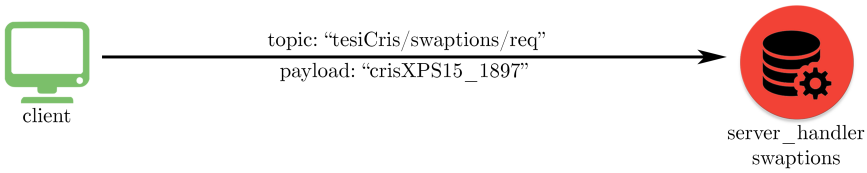


Figure 5.6: *Client request example*

This kind of publication is repeated until the node receives the predicted complete Operating Point list; server_handler replies to these requests according to its internal state, that can be one of the

following:

1. *unknown*;
2. *receivingInfo*;
3. *buildingDoE*;
4. *DSE*;
5. *buildingTheModel*;
6. *autotuning*.

server_handler *unknown* internal state

The server_handler doesn't know anything about the application it is managing, except its name; it asks to clients all the available information, making a publication with payload "info" on topic "tesiCris/*[appName]*":

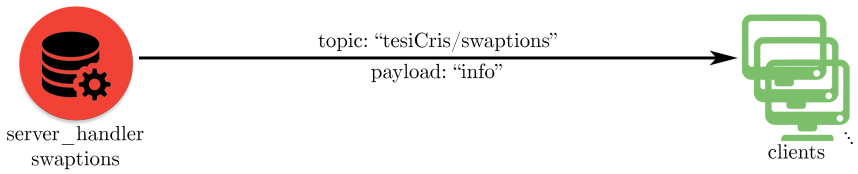


Figure 5.7: *Application information request by AgoràRemoteAppHandler example*

server_handler *receivingInfo* internal state

The server_handler is receiving application information, so in this case it discards all possible requests made by clients.

server_handler *buildingDoE* internal state

The server_handler, according to the received Design of Experiments type (see 5.2.4), is building the set of configurations that will be distributed to clients during Design Space Exploration phase; also in this case it discards all possible clients requests.

server_handler *DSE* internal state

The server_handler has computed DoE configurations, so it is driving Design Space Exploration phase; it picks up the first element on top of the available configurations list and it sends, in lexicographic order, the associated software knobs values on topic "tesiCris/*[appName]*/*[hostname]_[PID]*/conf", relative to the client that made the request; the configuration just sent is reinserted at the end of the mentioned list:

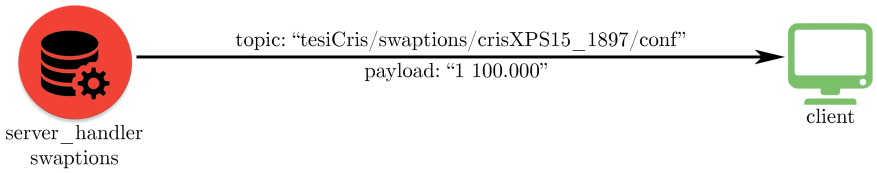


Figure 5.8: *Configuration dispatch by AgoràRemoteAppHandler example*

As shown in figure 5.8, client receives a configuration with $num\_threads = 1$ and $num\_trials = 100.000$; the next computation will be done with these parameters values.

server_handler *buildingTheModel* internal state

The server_handler module has gathered all the needed OPs related to DoE configurations and it is computing the complete Operating Points list through Machine Learning techniques; from the gathered Operating Points, a partial model is built, assembling to every DoE configuration the mean of metrics values, taken from the related OPs; this partial model in sent to the client that made the request: every publication is done on topic "tesiCris/*[appName]*/*[hostname]_[PID]*/model" with payload equal to the computed OP, with format "*[configuration] [metrics values]*"; both configurations and metrics values follow lexicographic order; finally, the server_handler makes a final publication on the same topic with payload "DoEModelDone":

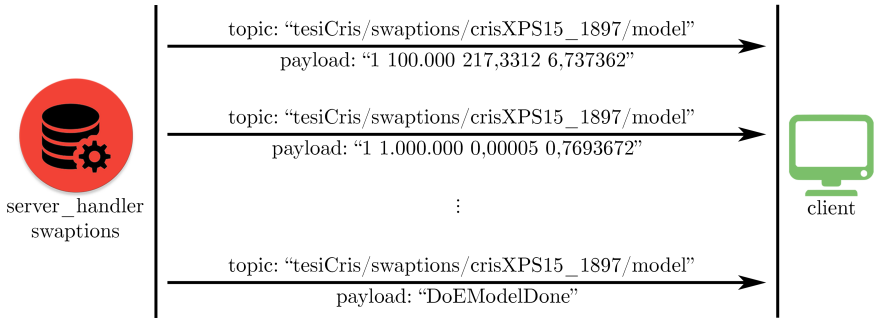


Figure 5.9: *Partial model dispatch by AgoràRemoteAppHandler example*

Taking figure 5.9 as reference, the first sent OP has parameters $num\_threads = 1$ and $num\_trials = 100.000$, with metrics $avg\_error = 217,3312$ and $avg\_throughput = 6,737362$; the client module sets up mARGOt autotuner with this OP list, so the application is executed with the best Operating Point that fulfils current goals and requirements.

server_handler *autotuning* internal state

The server_handler owns the complete OP list, obtained through the Generalized Linear Regression interface by Apache Spark™ MLlib library; similarly to the previous case (5.2.3), every Operating Point is sent to the client on topic "tesiCris/*[appName]*/*[hostname]_[PID]*/model" with format "*[configuration] [metrics values]*", respecting lexicographic order for both parameters and metrics values; the final publication has payload "modelDone":

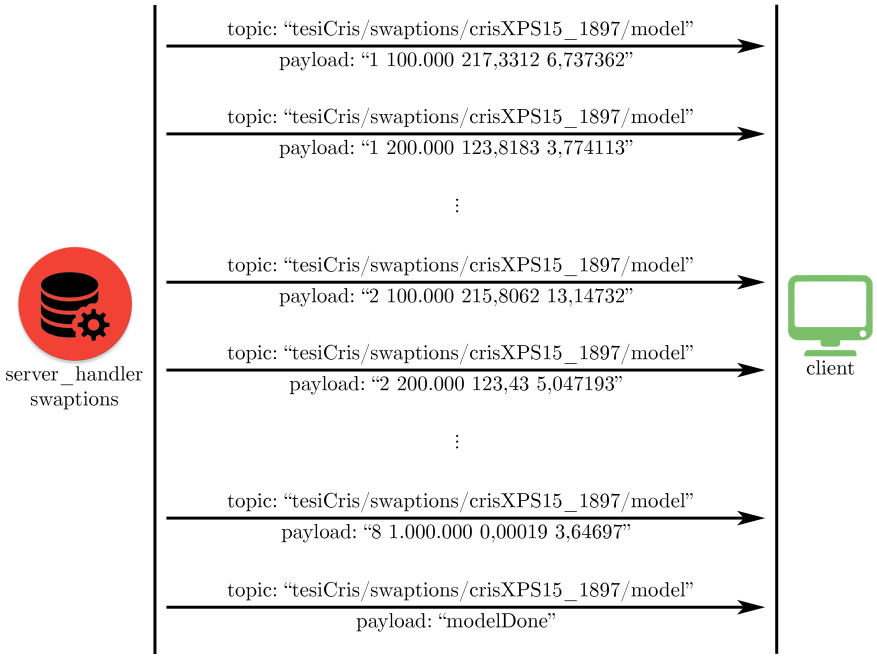


Figure 5.10: *Complete model dispatch by AgoràRemoteApplicationHandler example*

In swaptions application, variable num_threads can assume 8 different values, while variable num_trials 10 ones, so the complete model is composed by all the 80 OPs; after mARGOt autotuner has received all the predictions, it can set application knobs according to current objectives.

From this point on, client stops making requests to the server_handler module.

5.2.4 Client application information dispatch

It has been shown that, if a server_handler receives a request from a node but its internal state is unknown, it requests application information (see 5.2.3); the server_handler saves both the identifier of the first client that replies and all data it receives; other possible replies from other clients are discarded.

Mandatory information that cilent modules have to send is:

1. metrics under examination: the keyword is *metric*, followed by

metric name; there is a publication for each metric; publications have to be in lexicographic order with respect to metrics name;

e.g. payload: "metric avg_throughput"

2. application parameters: the keyword is *param*, followed by parameter name, the way in which it is transmitted and the corresponding values; there is a publication for each parameter; also in this case, publications must follow lexicographic order with respect to parameters name; tesiCris makes available two ways to send values:

   (a) by list: the keyword is *enum* and, in this case, all possible values are listed;

   (b) by extreme values and step: the keyword is *range* and, in this case, minimum value, maximum value and step are sent; the server_handler, from this information, computes all possible parameter values.

   e.g. payload: "param num_threads enum 1 2 3 4 5 6 7 8"

   e.g. payload: "param num_trials range 100.000 1.000.000 100.000"

There are some optional information that clients can send to the server_handler:

1. number of required repetitions for each Operating Point: the keyword is *numReps*, followed by a number; this value represents the number of Operating Points that the server_handler has to gather for each Design of Experiments configuration, during Design Space Exploration phase;

   e.g. payload: "numReps 5"

2. Design of Experiments type: the keyword is *DoE*, followed by the term that indicates the kind of Design of Experiments that has to be used; it can be:

(a) *fcccd*: it corresponds to the Face Centered Central Composite DoE with one Center Point;

(b) *ff2l*: it corresponds to the 2-Level Full-Factorial DoE;

(c) *pbd*: it corresponds to the Plackett-Burman DoE;

(d) *lhd*: it corresponds to the Latin-Hypercube DoE; in this case, there is another optional information that can be sent to the server_handler: the number of configurations that have to be produced, with the word *lhdSamples* followed by the desired value; if this information is not sent, the number of random configurations is equal to the number of application parameters;

(e) *fcccdExtra*: it corresponds to the Face Centered Central Composite DoE with one Center Point plus the addition of other configurations through the Latin-Hypercube DoE; as in the previous case, the optional keyword *lhdSamples* is used to express the number of extra configurations, otherwise they are equal to the number of parameters;

(f) *fullFact*: it corresponds to the Full-Factorial DoE.

Design of Experiments concepts are explained in chapter 2.3.

e.g. payload: "DoE fcccdExtra"

e.g. payload: "lhdSamples 6"

3. Response Surface Modeling technique: the keyword is *RSM*, followed by the term that indicates the Machine Learning technique that has to be used in order to predict the complete OP list; tesiCris has implemented two versions of the Apache Spark™ Generalized Linear Regression RSM, explained in detail in 2.6.2:

   (a) the first, that uses parameters values transformations, with associated word *sparkGenLinRegrTransforms*;

(b) the second, that uses parameters polynomial expansion of second order, with associated word *sparkGenLinRegr2ndPolyExp*

e.g. payload: "RSM sparkGenLinRegr2ndPolyExp"

4. parameters values transformations for the first implemented version of Apache Spark™ Generalized Linear Regression RSM: the keyword is *paramsTransforms*, followed by the involved metric name and the terms that indicate the kind of parameters transformations, the family distribution and the link function.

   Transformations must follow the same order of parameters information dispatch; they can be:

   (a) *inv*: in this case, to the corresponding parameter values in the OPs, the inverse function is applied;

   (b) *ln*: in this case, to the corresponding parameter values in the OPs, the natural logarithmic function is applied;

   (c) *sqrt*: in this case, to the corresponding parameter values in the OPs, the square root function is applied;

   (d) *id*: in this case, the corresponding parameter values in the OPs are not transformed.

   tesiCris has focused on the prediction of continuous functions with normal distribution: the corresponding family is the Gaussian one, indicated with word *gaussian*.

   For the Gaussian family, the link function can be:

   (a) *identity*;

   (b) *log*;

   (c) *inverse*.

   If this kind of information is available, it must exist for each metric of interest.

e.g. payload: "paramsTransforms avg_error id sqrt gaussian log"

5. number of application features: the keyword is *numFeats*, followed by the corresponding number; in this case, a minimum number *n* of features values observations can be specified through the keyword *minNumObsFeatsValues*, followed by *n*: only those features values that are observed at least *n* times, during Design Space Exploration phase, contribute to the prediction of the complete OP list; if no features value reaches *n* observations, *n* becomes the number of observations of the most observed features value; if this last information is missing, $n = 1$.

   e.g. payload: "numFeats 1"

   e.g. payload: "minNumObsFeatsValues 5"

If not specified, the default Design of Experiments type is Face Centered Central Composite DoE with one Center Point, the default number of repetitions for each OP is 1, the default number of program features is 0 and the default RSM technique is the implemented second version of the Apache Spark™ Generalized Linear Regression. If the chosen RSM technique is the GLR first version but there is no information about parameters values transformations, tesiCris tries every possible combination in union with all possible link functions, choosing the best result according to Akaike Information Criterion value and the mean of the sum of coefficient standard errors (see 2.6.2).

All the information is sent by clients on topic "tesiCris/*[appName]*/info/*[hostname]_[PID]*"; a final publication with message "done" specifies to the server_handler that application information is finished:

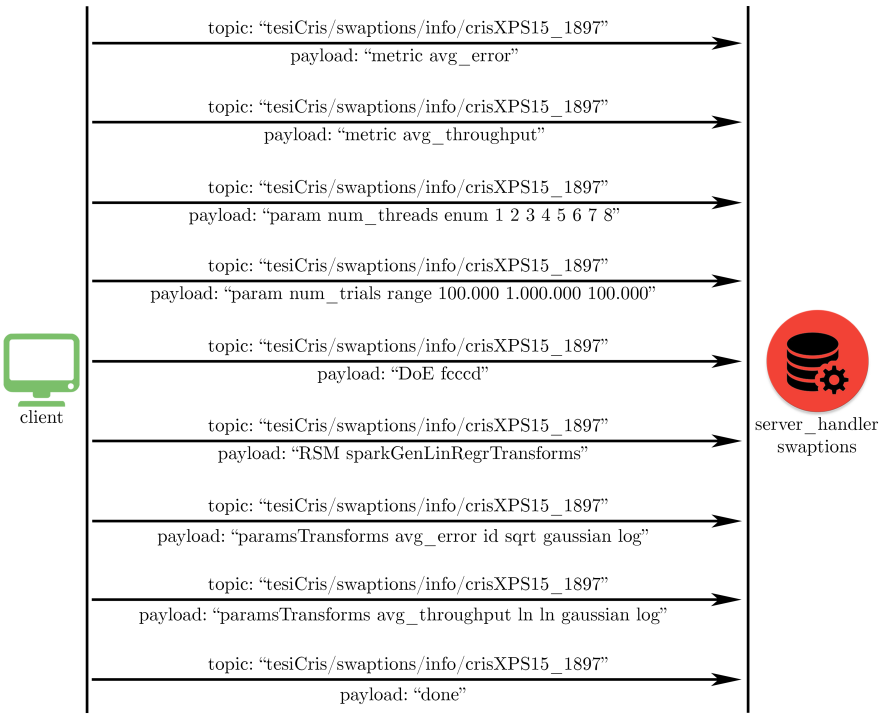


Figure 5.11: *Application information dispatch by AgoràLocalAppHandler example*

Taking figure 5.11 as reference, the server_handler pulls out, from topic, *crisXPS15_1897* and it saves this identifier as the client that is sending program information; this application focuses on two metrics, *avg_error* and *avg_throughput*; it has two parameters, *num_threads* and *num_trials*; the first is sent with the complete list of values, while the latter is sent with the keyword *range*, specifying its minimum value (100.000), its maximum one (1.000.000) and the step (100.000): the server_handler computes all values, that are therefore $100.000, 200.000, 300.000, ..., 1.000$ the Design of Experiments that has to be used is the Face Centered DoE with one Center Point and the RSM technique is the first implemented version of the Apache Spark™ Generalized Linear Regression; features transformations are also specified so, e.g. for metric *avg_error*, the first parameter (*num_threads*) will not be transformed, while the second parameter (*num_trials*) has to be transformed with the square root function, applying *gaussian* as family

distribution and *log* as link function; it can be noticed that there is no information about the number of Operating Points repetitions to gather during DSE phase: in this case the default value is used (one repetition for each OP); finally, there is no payload with keyword *numFeats* either: the application does not have features.

The server_handler is now ready to compute Design of Experiments configurations; after that, it can start distributing them to the nodes, driving the subsequent Design Space Exploration phase.

5.2.5 Client Operating Point dispatch

During DSE, clients store the configurations that receive from the server_handler (see 5.2.3); every time a configuration is sent, if the current one differs from the new one, the client module communicates to mARGOt autotuner new parameters values; software knobs are set up, therefore the next program computation is executed with new parameters values.

After the execution, client module arranges the obtained Operating Point, composed by the set of parameters values and the observed metrics of interest; it publishes on topic "tesiCris/*[appName]*/OPs" a message in the form "*[configuration]*:*[metrics values]*", in which both values lists have to follow lexicographic order with respect to parameters and metrics name:

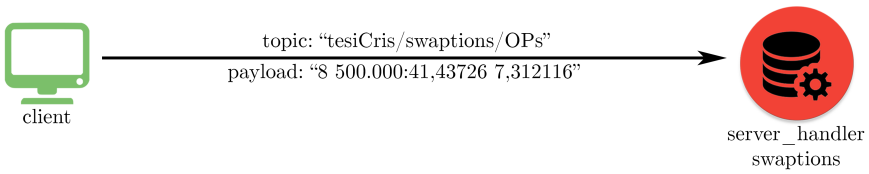


Figure 5.12: *Operating Point dispatch by AgoràLocalAppHandler example*

In the example above, the application has been just executed with $num\_threads = 8$ and $num\_trials = 500.000$; monitored metrics values are $avg\_error = 41,43726$ and $avg\_throughput = 7,312116$.

When the server_handler receives an OP, it decrements the corresponding number of needed Operating Points repetitions: if the updated value is equal to zero, it means that there is no need of other related OPs, so the configuration is moved from the set of available ones to the set of accomplished ones.

The model prediction just starts when the last needed Operating Point repetition for the last available configuration is received from a node.

5.2.6 Client disconnection

If a client disconnects for some reason, the server_handler receives a message on topic "tesiCris/*[appName]*/disconnection" with payload "*[hostname]_[PID]*", relative to the disconnected client; the server_handler removes the node identifier from the list of machines that it is managing:

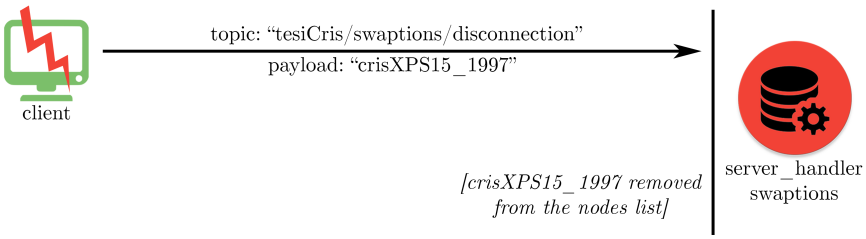


Figure 5.13: *client disconnection example*

Particular attention has to be taken if the disconnected client is the one that, at the beginning, is sending application information (see 5.2.4): in this case, the server_handler has to remove the disconnected machine, it has to reset partial data (received up to that moment) and it asks again all the available application information to remaining connected clients:

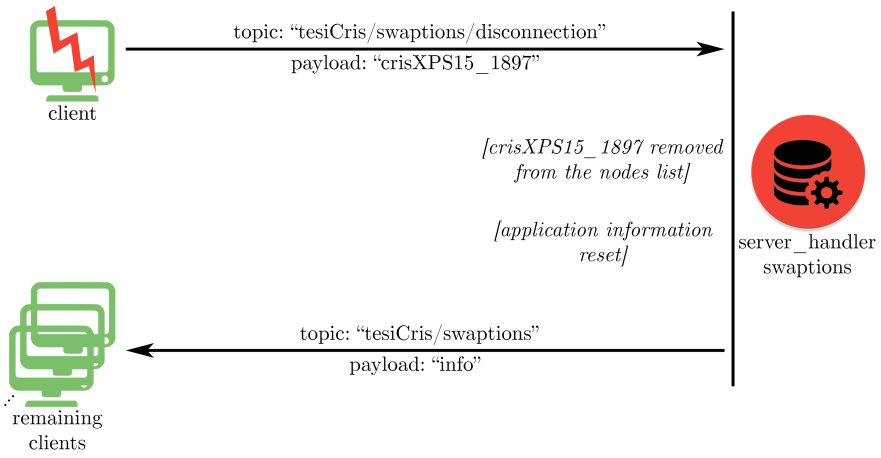


Figure 5.14: *Disconnection of AgoràLocalAppHandler that was sending application information example*

5.2.7 server_handler disconnection

If the server_handler disconnects, clients receive a message on topic "tesiCris/*[appName]*" with payload "disconnection":

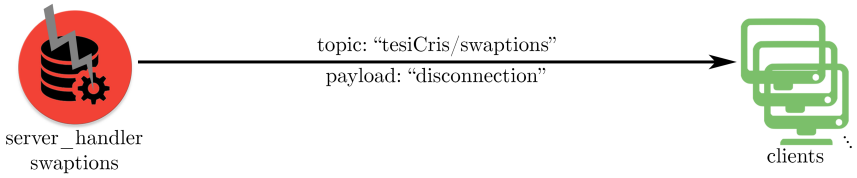


Figure 5.15: *server_handler disconnection example*

Each client reacts to this event according to its internal state, that can be:

1. *defaultStatus*;
2. *DSE*;
3. *DoEModel*;
4. *autotuning*.

client *defaultStatus* internal state

When a node starts running, the autotuner sets up application parameters values with a predetermined default configuration; if any

Design Space Exploration phase has not been started yet, the server_handler disconnection doesn't affect application behavior.

client *DSE* internal state

The server_handler is driving Design Space Exploration phase, sending configurations to clients; in this case, the default configuration is restored and the application is executed with the corresponding parameters values:

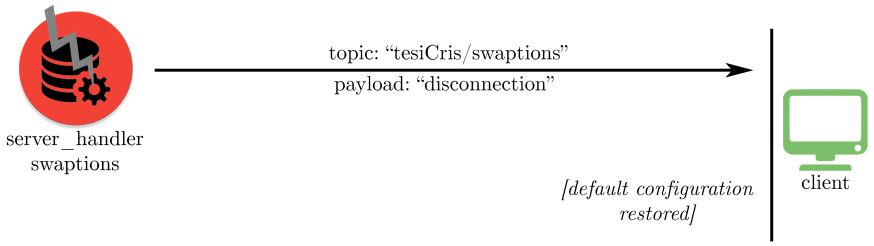


Figure 5.16: *server_handler disconnection with client internal state equal to DSE example*

client *DoEModel* internal state

Client has received a partial OP list, related to Design of Experiments configurations (see 5.2.3); in this case, the available OP list is not deleted, therefore the autotuner continues to work with this information.

client *autotuning* internal state

Client has already received the predicted complete OP list from the server_handler, therefore nothing changes.

5.3 Client module integration

```
1  #include "tesiCris_margot_manager.hpp"
2
3  int param1;
4  int param2;
5  int param3;
6
7  int main()
8  {
9    tesiCris_margot_manager tmm;
10   tmm.init();
11
12   while( loop_condition() )
13   {
14       tmm.update_OPs();
15
16       margot::margot_block
17       {
18           do_computation( param1, param2, param3 );
19       }
20
21       tmm.sendResult( { param1, param2, param3 },
22                       { margot::margot_block::metric1, margot::margot_block::metric2 } );
23       }
24 }
```

Figure 5.17: *Sketch application with AgoràLocalAppHandler plus mARGOt autotuner integration*

Figure 5.17 shows a sketch application that, until *loop_condition()* is verified (line 12), is executed; computation depends on three parameters (*param1, param2, param3*) that are set up by mARGOt autotuner at the beginning of every cycle (line 16), while two metrics of interest (*metric1, metric2*) are monitored (for mARGOt details, see the related scientific publication [23]).

The integration code required to use tesiCris framework with mARGOt autotuner is written in bold red; three are the main steps during program execution:

1. tesiCris client module and mARGOt autotuner instantiation and initialization (lines 9-10): the client module saves all application information and sets up mARGOt autotuner with a default Operating Point; if nothing happens, the application is executed with this configuration;

2. Operating Points knowledge update (line 14): tesiCris module updates, from time to time, its internal knowledge about application configurations that are sent by the server_handler; before mARGOt autotuner sets up application parameters (line 16), tesiCris checks mARGOt knowledge with respect to its internal one: if they are different, mARGOt configurations are updated. In this way, for instance, if the client module receives a new configuration during Design Space Exploration phase (see 5.2.3), mARGOt knowledge is set up with this data, so the application is forced to be executed with the corresponding parameters values; if, for instance, the complete model is received (see 5.2.3), mARGOt internal knowledge is set up with all this information, so the autotuner can choose the best Operating Point that fulfils application current goals and requirements;

3. Operating Point dispatch (line 21-22): after the computation has done, parameters values just used with the corresponding monitored metrics of interest are published on a predetermined MQTT topic, so the server_handler that is in charge of this application can receive this information (see 5.2.5).

CHAPTER 6

Experimental results

In order to demonstrate tesiCris validity, we have done some tests on various applications and scenarios. Firstly, we present the structure of programs we have used to do experiments; after that, we show test types and corresponding results.

6.1 Experimental setup

We have tested tesiCris in three different scenarios: two versions of a synthetic application and a real one. This work has been coupled with mARGOt autotuner ([23]); from now on, we use the concept of Operating Point (OP) concerning applications configurations in terms of parameters values and associated performance (metrics of interest values).

Synthetic application has three parameters: $param_1$, $param_2$ and $param_3$; a function of these three variables calculates the amount of milliseconds of an execution cycle (*executionTime* variable), while another function sets up an error measure; this last variable is considered as metric, together with application throughput as number of jobs per second (so, approximately equal to $\frac{1000}{executionTime}$).

For the real scenario, *Swaptions* application has been used, taken from the PARSEC benchmark suite ([13]). This application solves partial differential equations through Monte Carlo simulations in order to price a portfolio of swaptions; its tunable parameters are: the number of threads (variable *num_threads*, from 1 to 8) and the number of trials for the simulation at every cycle (variable *num_trials*, from 100.000 to 1.000.000 with a step of 100.000); observed metrics of interest are: throughput (variable *avg_throughput*) as the number of priced swaptions per second and error (variable *avg_error*), computed as:

$$avg\_error = \frac{\sum_{s \in pricedSwaptions} |StandDevRef(s) - StandDev(s)|}{|pricedSwaptions|}$$

where $StandDevRef(s)$ is the reference standard deviation for swaption s, $StandDev(s)$ is the evaluated one and $pricedSwaptions$ represents all swaptions that are priced every computing cycle; so, metric avg_error stands for the average of differences between standard deviation of priced swaptions using evaluated configuration with respect to the reference one (standard deviation for 1.000.000 trials).

The parallel architecture has been simulated by a Dell XPS 15 9550 with 4 core / 8 threads Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 processor.

6.1.1 Synthetic application version 1

In the first version of synthetic application, the amount of execution time is calculated as:

$$executionTime = 7.35 \cdot \ln(param_1) + 38.1 \cdot param_2 + 52.96 \cdot \sqrt{param_3} + noise$$

where *noise* simulates a disturbance; it is computed as:

$$noise = executionTime \cdot randomNumber \cdot noisePercentage$$

randomNumber is a generated random value according to an exponential distribution with mean 0.3; *noisePercentage* affects noise weight and it can be equal to *1%, 5%, 10%, 15%, 25%* or *50%*.

Error metric is calculated as:

$$error = \frac{1}{0.015 \cdot \sqrt{param_1} + 0.033 \cdot \ln(param_2) + 0.028 \cdot \ln(param_3)}$$

Since *randomNumber* tends to 0.3, there is a reference model for each application version with a prearranged *noisePercentage* value, where *randomNumber* is replaced with the mean of the chosen exponential distribution, so 0.3.

Application parameters can assume these values:

Parameters	Values
$param_1$	1, 50, 150, 300, 450, 700, 800
$param_2$	1, 50, 100, 150, 200
$param_3$	10, 80, 150, 220, 290, 360, 430, 500, 570, 640, 710, 780, 850

This application has 455 Operating Points; figure 6.1 shows complete OPs distribution for $noisePercentage = 15\%$: every point represents a particular configuration with an error value (on x-axis) and a throughput one (on y-axis). Since $noise$ in $executionTime$ is affected

by $randomNumber$, this plot consider its expected value, equal to the mean of the chosen exponential distribution (0.3); the obtained OPs list represents the reference model for this predetermined $noisePercentage$ value; of course, other reference models are similar, since the only difference is the *noisePercentage* value for the calculation of *executionTime* variable.

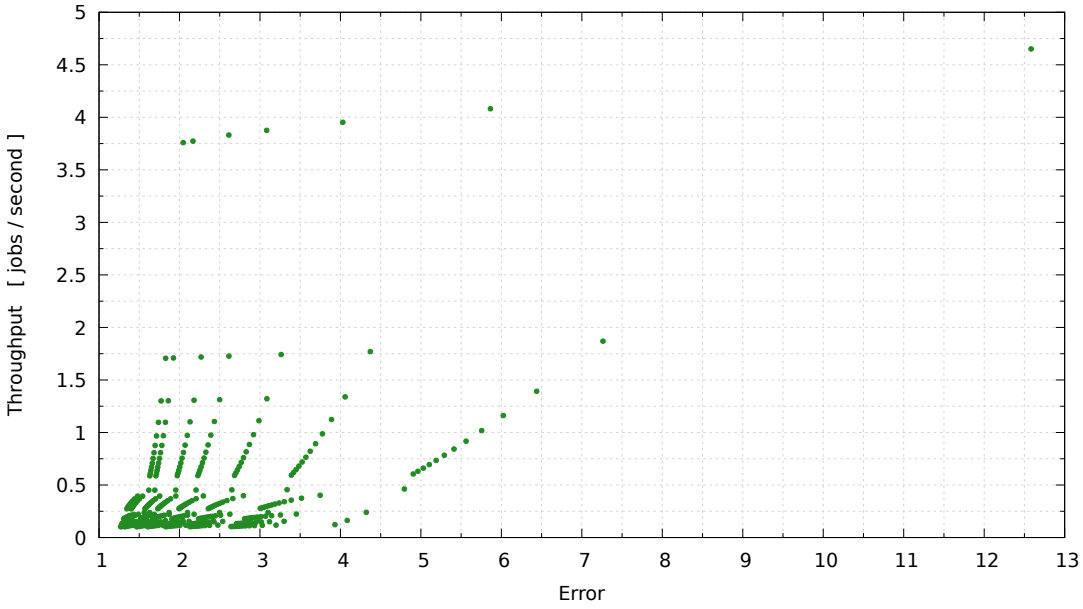


Figure 6.1: *Complete OP distribution of synthtetic application version 1 (reference model with $noisePercentage = 15\%$); every point stands for a particular configuration with associated observed metric values; on x-axis: error metric values, on y-axis: throughput metric values [jobs / second]*

Concerning model prediction quality, therefore, for every application setting with a fixed *noisePercentage*, estimated OPs list is compared with the corresponding reference one.

6.1.2 Synthetic application version 2

In the second synthetic application version, execution time is equal to:

$$executionTime = 7.4 \cdot param_1 \cdot param_2 + 2.1 \cdot (param_3)^2 + noise$$

where *noise* is simulated as in the previous application version, while the error is:

$$error = \frac{1}{0.01 \cdot param_1 + 0.7 \cdot \ln(param_2) + 0.019 \cdot param_3}$$

Parameters values are:

Parameters	Values
$param_1$	1, 10, 15, 25, 40, 65, 80
$param_2$	1, 5, 10, 20
$param_3$	10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46

In this application the number of Operating Points is 364; figure 6.2 shows complete OPs distribution of reference model with $noisePercentage = 5\%$:

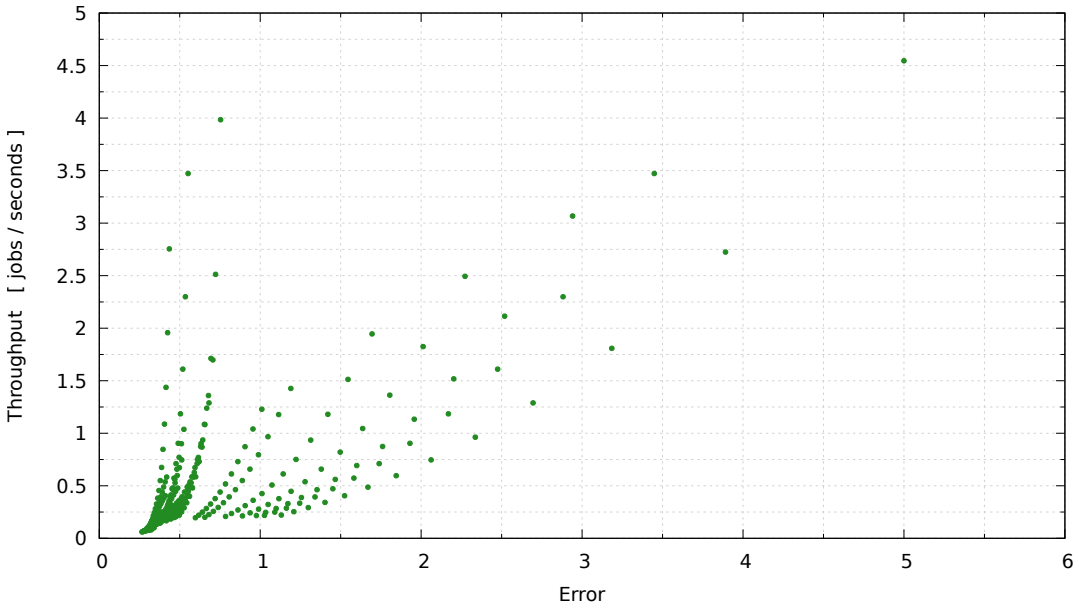


Figure 6.2: *Complete OP distribution of synthtetic application version 2 (reference model with $noisePercentage = 5\%$); every point stands for a particular configuration with associated observed metric values; on x-axis: error metric values, on y-axis: throughput metric values [jobs / second]*

Of course, regarding model prediction goodness, same reasoning as previous application version is applied.

6.1.3 Swaptions

As already stated in 6.1, this application has two parameters: *num_threads* and *num_trials*; their values are:

Parameters	Values
num_threads	1, 2, 3, 4, 5, 6, 7, 8
num_trials ($\cdot 10^3$)	100, 200, 300, 400, 500, 600, 700, 800, 900, 1000

The number of Operating Points is therefore 80; figure 6.3 shows complete OPs distribution with respect *avg_error* and *avg_throughput* metrics of interest:

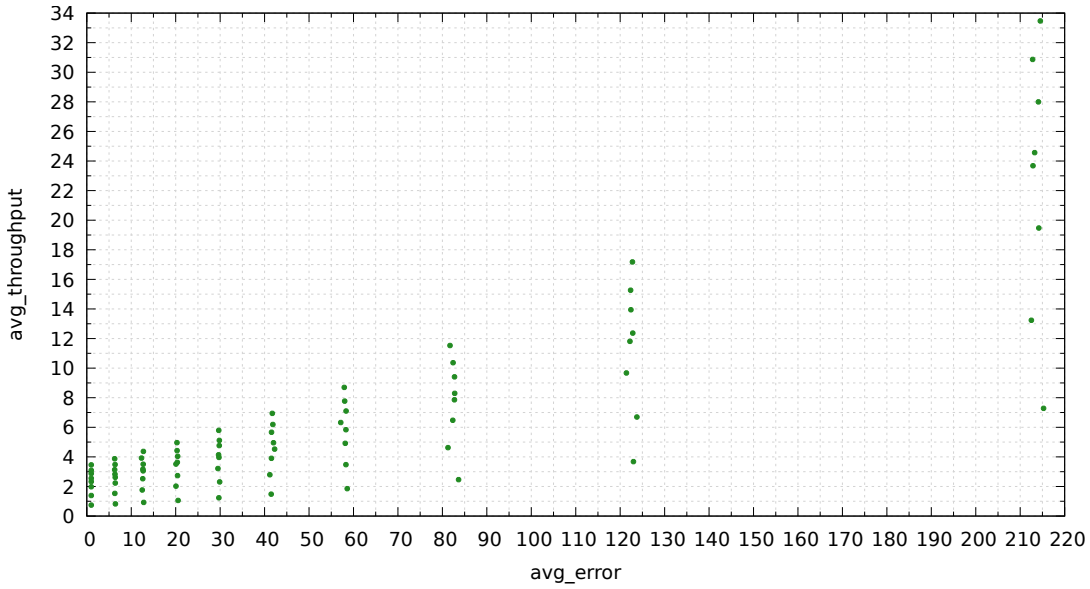


Figure 6.3: *Swaptions complete OP distribution; every point stands for a particular configuration with associated observed metric values; on x-axis: avg_error metric values, on y-axis: avg_throughput metric values [priced swaptions / second]*

6.2 Experimental campaign

We want to demonstrate tesiCris validity; for both synthetic application versions and *Swaption*, we have focused our attention on

models prediction goodness in various scenarios; we have studied execution times, especially for Design Space Exploration phases with one and more than one executing applications at the same time, highlighting benefits in sharing DSE; we have revealed applications behavior during execution on different cases; next paragraphs shows results, firstly for synthetic applications, finally for the real one.

6.2.1 Synthetic application

In this paragraph we show experimental results for both versions of synthetic application, divided as explained before.

Models prediction quality

Concerning model prediction goodness, both synthetic application versions have been executed several times with the introduction of all noise weights (*1%, 5%, 15%, 10, 25%* and *50%*), focusing on the prediction of throughput metric, that is the one affected by *noisePercentage* value; a Face Centered Central Composite Desing of Experiments with one Center Point has been used, firstly gathering 1 repetition, then 5 repetitions and finally 10 repetitions for each DoE configuration during Design Space Exploration phase; for each predicted Operating Point for each application setting, we have calculated a *deltaError* measure, that is how much the predicted metric value distances itself from the corresponding exact measure of the related reference model, in percentage; next figures summarize results:

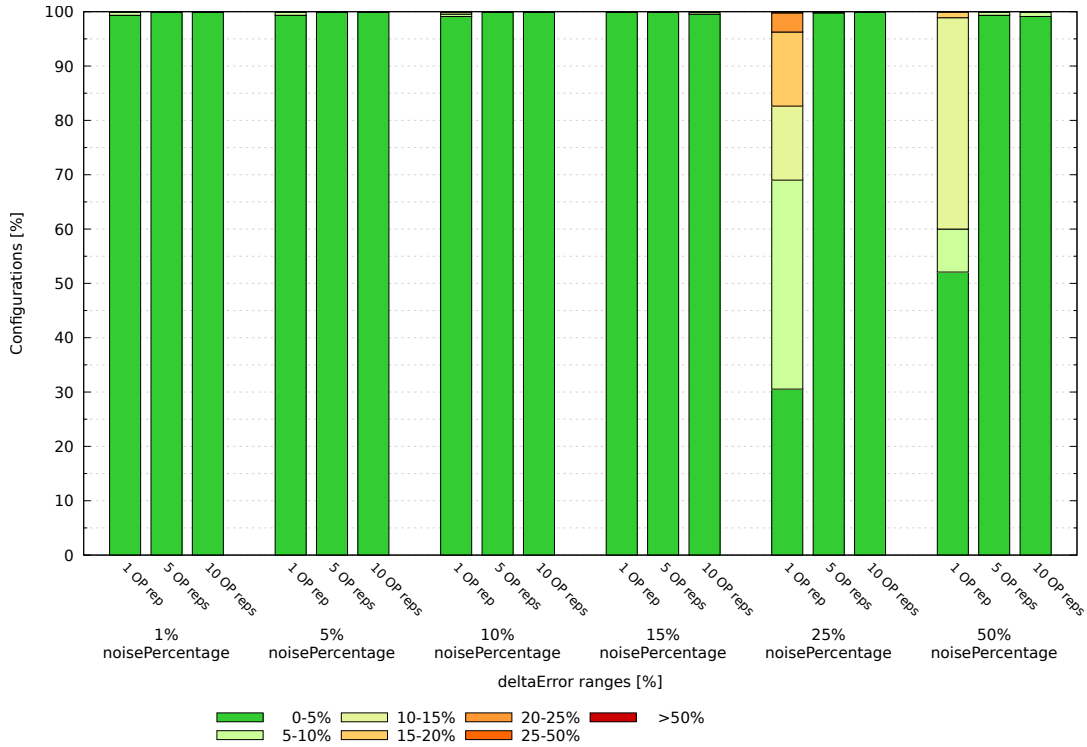


Figure 6.4: *deltaError results for synthetic application version 1; used RSM: 1st version of implemented GLR ("transformations by functions", see 2.6.2); each stacked bar shows deltaError results for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: number of configurations [%]*

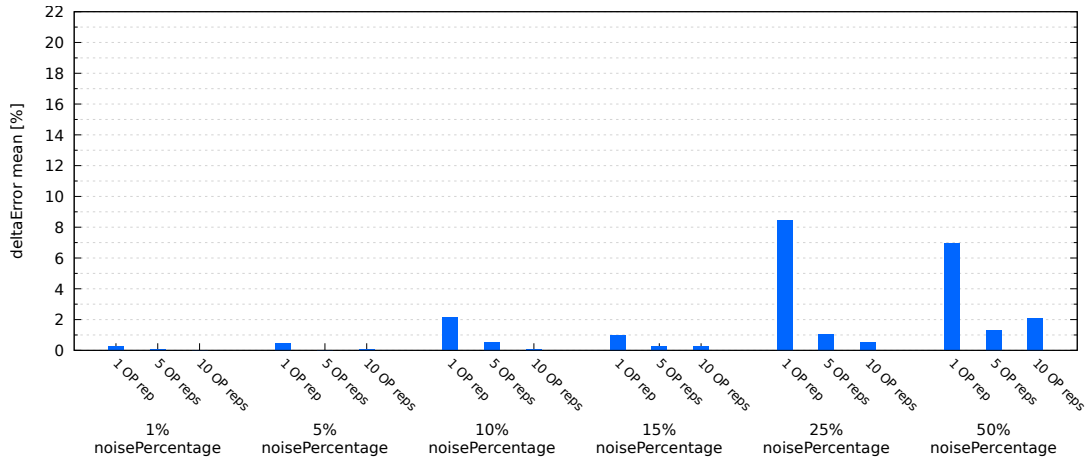


Figure 6.5: *deltaError mean values for synthetic application version 1; used RSM: 1st version of implemented GLR ("transformations by functions", see 2.6.2); each bar shows deltaError mean for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: deltaError mean [%]*

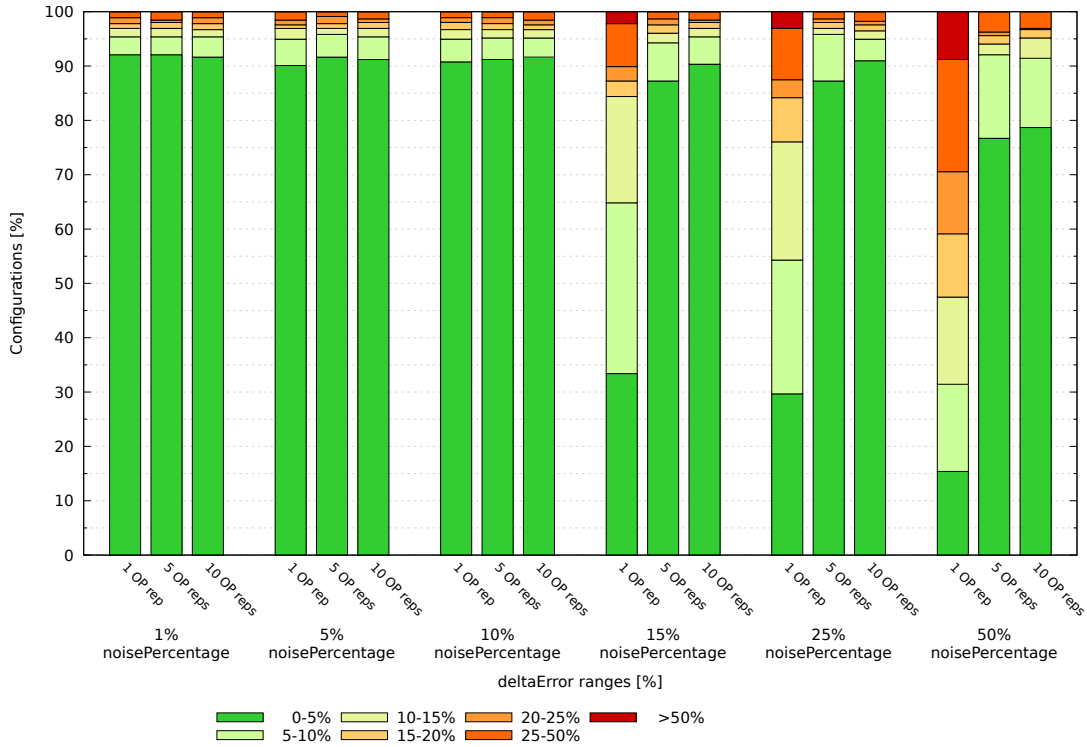


Figure 6.6: *deltaError results for synthetic application version 1; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each stacked bar shows deltaError results for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: number of configurations [%]*

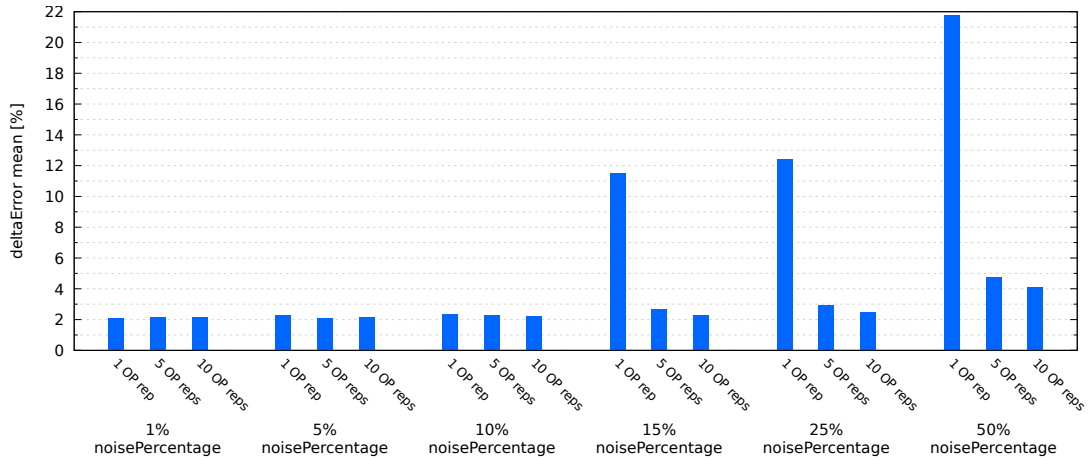


Figure 6.7: *deltaError mean values for synthetic application version 1; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each bar shows deltaError mean for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: deltaError mean [%]*

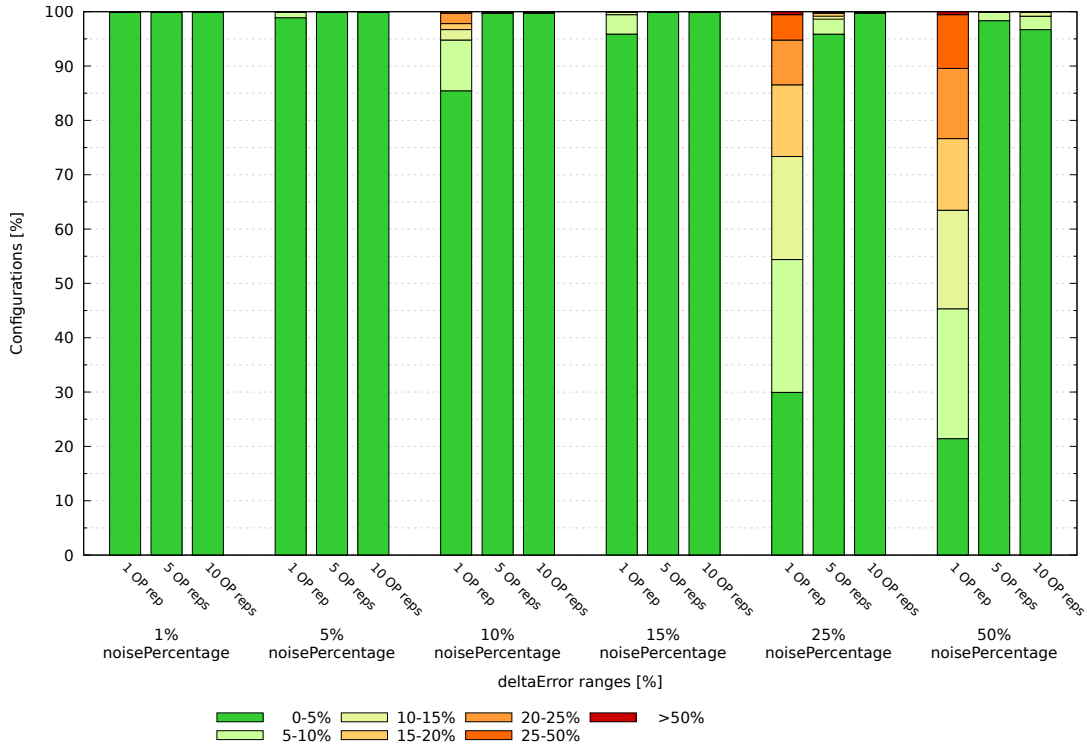


Figure 6.8: *deltaError results for synthetic application version 2; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each stacked bar shows deltaError results for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: number of configurations [%]*

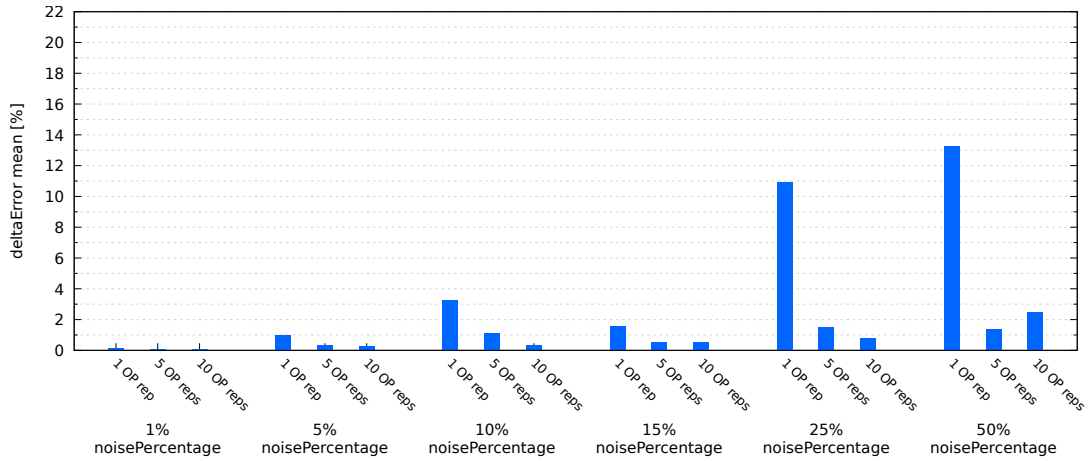


Figure 6.9: *deltaError mean values for synthetic application version 2; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each bar shows deltaError mean for a particular application configuration, with respect to noisePercentage value and number of collected OP repetitions during DSE phase; on y-axis: deltaError mean [%]*

In figures 6.4, 6.6 and 6.8, every stacked bar chart represent an application setting with respect to $noisePercentage$ value and the number of repetitions, for each DoE configuration, collected during DSE phase; on y-axis there are the number of configurations, in percentage, grouped with respect to measured $deltaError$; related figures 6.5, 6.7 and 6.9 show $deltaError$ mean for each setting.

From figure 6.4 we can see that, until $noisePercentage = 15\%$, model prediction is really precise even with only 1 repetition for each DoE configuration: almost the totality of configurations have a *deltaError* below 5%, the majority of which below 2%; with 5 repetitions and 10 repetitions, essentially all configurations have a *deltaError* below 1%. With the introduction of a strong noise weight, 25% and 50%, prediction get worse with 1 OP repetition, even if, respectively, around 70% and 60% of configurations remains with a *deltaError* below 10%; prediction gets back really accurate with 5 and 10 OP repetitions also with these high noise values.

Figure 6.6 shows that, for synthetic application 1, 2nd version of

the implemented GLR produces, in general, results less precise that the ones generated with the 1st GLR version: up to $noisePercentage = 10\%$, quality of predicted models is very satisfying, where more than 90% of configurations have a *deltaError* below 5%; from noise weight equal to 15%, 25%, 50% and with 1 OP repetition, this *deltaError* percentage decreases to 35%, 30% and 15% respectively and a visible number of configurations have a *deltaError* greater that 25%; for these settings, model prediction quality remarkably increases collecting more training data for each Design of Experiments configuration: with 5 and 10 OP repetitions we notice very few configurations with a *deltaError* greater that 10%, so prediction quality becomes very accurate even in these awful cases.

Concerning synthetic application version 2 with the 2nd version of implemented Generalized Linear Regression as RSM (figure 6.8), results are very similar to the ones related to synthetic application version 1 with 1st version of implemented Generalized Linear Regression (figure 6.4): quality of models prediction is generally very high, even with heavy noises but, in these cases, there is the need of more OP repetitions in order to mitigate this strong disturbance.

For synthetic application version 1, as it can be understood, 1st implemented version of GLR works slightly better than the 2nd one, but their prediction times are very different, as shown in table 6.1:

	1 OP rep	5 OP reps	10 OP reps
GLR 1st version	155.52 sec	155.7 sec	157.72 sec
GLR 2nd version	23.21 sec	23.64 sec	23.8 sec

Table 6.1: *Model predictions times for each implemented Generalized Linear Regression for each number of OP repetitions scenario*

2nd GLR version takes less than 24 seconds to predict complete model, while 1st GLR version approximately 156 seconds: the former, almost to the same level of quality, is more than 6 times faster than the latter. Moreover, we have noticed that the number of OP

repetitions affects very poorly model prediction time: from table 6.1 we can understand that, among 1, 5 and 10 OP repetitions for each DoE configuration case, times vary very few.

We have not disclosed models prediction quality for synthetic application version 2 with the 1st implemented version of GLR: transformations by functions are not able to capture functions of second order behavior, as variable *executionTime* is built in this case. On the contrary, polynomial expansion of second order has the capability to highly predict functions with, for instance, logarithmic or square root values, as demonstrated in the second models prediction analysis (6.6).

From last analyses, we can assert that Generalized Linear Regression that uses polynomial expansion of second order as parameters transformations is a Response Surface Methodology much more powerful than GLR that uses transformations by functions: the latter version can predict a restricted set of scenarios, while the former is able to include those cases and the wide variety of quadratic functions. tesiCris has been able to predict very well these metric behaviors and to properly manage strong noise disturbances, with the necessity to do, in some cases, a longer Design Space Exploration phase, collecting more OPs for each Design of Experiments configuration; lastly, figures 6.5, 6.7 and 6.9 recap all various scenarios, showing *deltaError* mean values that increase with high noises, going back still very low with 5 and 10 OP repetitions, even in the worst cases.

Execution times

Concerning execution times, we want to show the significant benefit of sharing Design Space Exploration phase among several nodes, running the same application: the 2nd version of synthetic application with $noisePercentage = 10\%$ has been executed, using Face Centered Central Composite Desing of Experiments with one Center Point

and collecting, for each configuration, 20 repetitions; for this application setting, total number of configurations to be explored was 300. We have accomplished this DSE phase with 1 up to 10 executing applications, collecting the overall executing time, represented by each bar chart in figure 6.10:

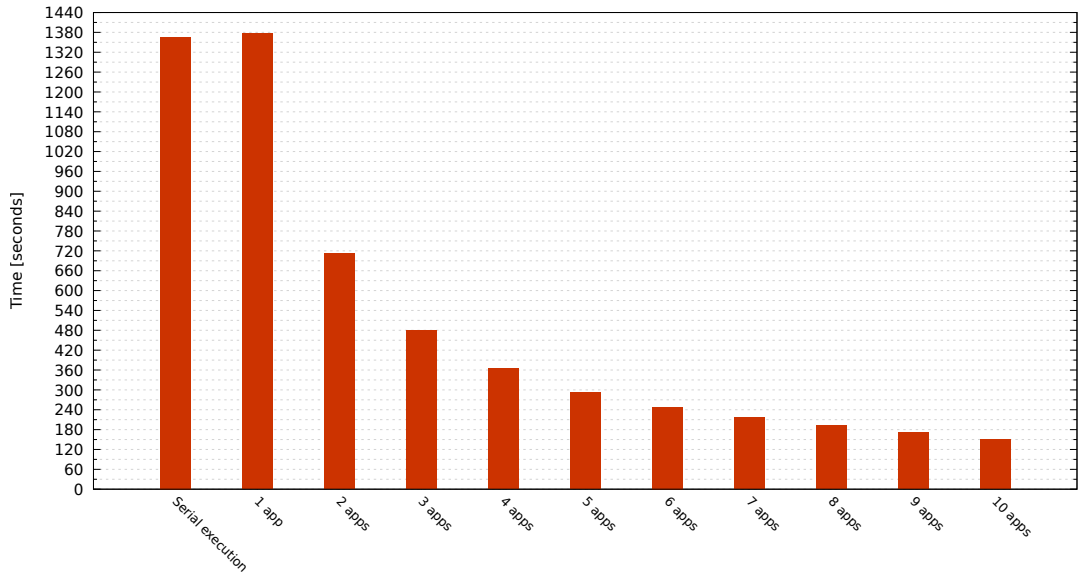


Figure 6.10: *Design Space Exploration times for several numbers of executing applications; first bar is related to DSE carried out exploring each configuration in a serial way, without Agorà; other bars are related to DSE phase done with implemented framework, through 1 executing application up to 10 ones; on y-axis: elapsed time [seconds]*

From figure 6.10 we can see that tesiCris took about 23 minutes to terminate Design Space Exploration with 1 executing application (second bar chart). The first bar chart shows the amount of time needed to execute application with all the 300 configurations in a serial way, without tesiCris; this execution lasted around 10 seconds less than the first analyzed scenario, due to the lack of tesiCris overheads; as we can understand, they are in any case very little: they add only approximately 0.73% of time with respect of the serial execution. From 2 executing applications on, DSE time strongly decreases: tesiCris took around 12 minutes to finish this phase with just 2 ex-

ecuting applications, 8 minutes with 3 ones, up to only 2 and a half minutes with 10 ones, more than 9 times lower than DSE phase with 1 application. Sharing Design Space Exploration phase among more than one executing application clearly speeds up overall time, since collection of training data duration, that is much longer than model prediction one (table 6.1), considerably goes down.

Now we want to show general tesiCris advantages, comparing a Full-Factorial, so exhaustive, application execution with the one supervised by this work; in figure 6.11, first bar chart shows overall time needed to compute Full-Factorial run for synthetic application version 2 with $noisePercentage = 15\%$, analyzing every possible configuration one time; next stacked bar charts show overall time, divided by DSE phase and model prediction, for the same application setting using tesiCris, collecting 1, 5 and 10 OP repetitions for each DoE configuration:

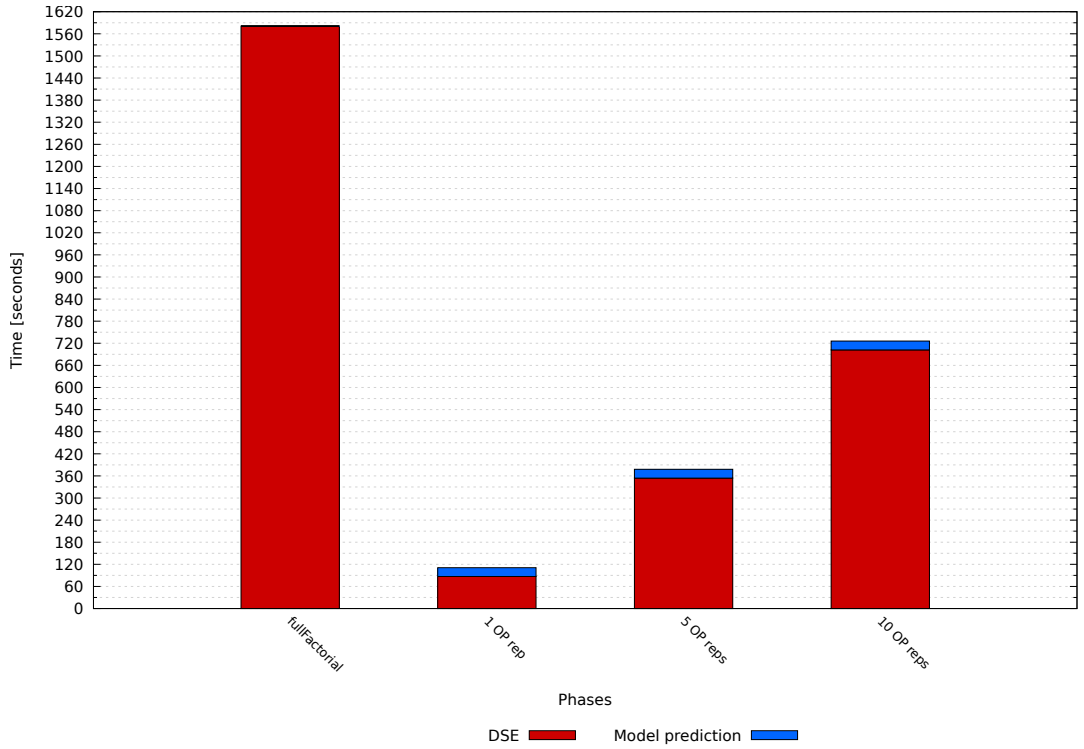


Figure 6.11: *Synthetic application version 2 (with $noisePercentage = 15\%$) execution times; first bar shows elapsed time for exhaustive execution; adjacent ones reveal overall time, until complete model has been predicted, using 1, 5 and 10 DoE configuration repetitions during DSE phase; on y-axis: elapsed time [seconds]*

Figure 6.11 shows that an exhaustive execution of analyzed application takes more than 26 minutes to finish; with the introduction of tesiCris, using Face Centered Central Composite Desing of Experiments with one Center Point and the 2nd version of implemented GLR as RSM, overall time becomes 12 minutes if 10 repetitions for each DoE configuration are collected, about 6 minutes with 5 repetitions and even around 2 minutes with 1 repetition; we specify that, if we wanted to explore every possible configuration in the exhaustive analysis as much as we have done for 5 and 10 OP repetitions of DoE configurations with tesiCris, of course overall time would be 260 minutes and 130 minutes respectively, increasing even more the gap between Full-Factorial execution and the ones with the supervise of this work. Finally, we also want to remind, from figures 6.8

and 6.9, that tesiCris, for this application setting, can predict a complete model with a very high quality even collecting just 1 repetition for each DoE configuration, so the final result is very similar to an exhaustive search. Therefore, this work can produce high benefits in the prediction of applications complete model, starting from a small subset of configurations.

Application behavior over time

Last figure wants to show application behavior during execution with the supervise of tesiCris and mARGOt autotuner; on x-axis there is time, while on y-axis there are metrics of interest values assumed by application during execution. We have run synthetic application version 2 with $noisePercentage = 15\%$, using Face Centered Central Composite Design of Experiments with one Center Point and the 2nd version of implemented GLR as RSM, collecting 5 repetitions for each Design of Experiments configuration during DSE phase; objective functions have been set to $Throughput > 3$ and $Error < 1$; figure 6.12 shows results:

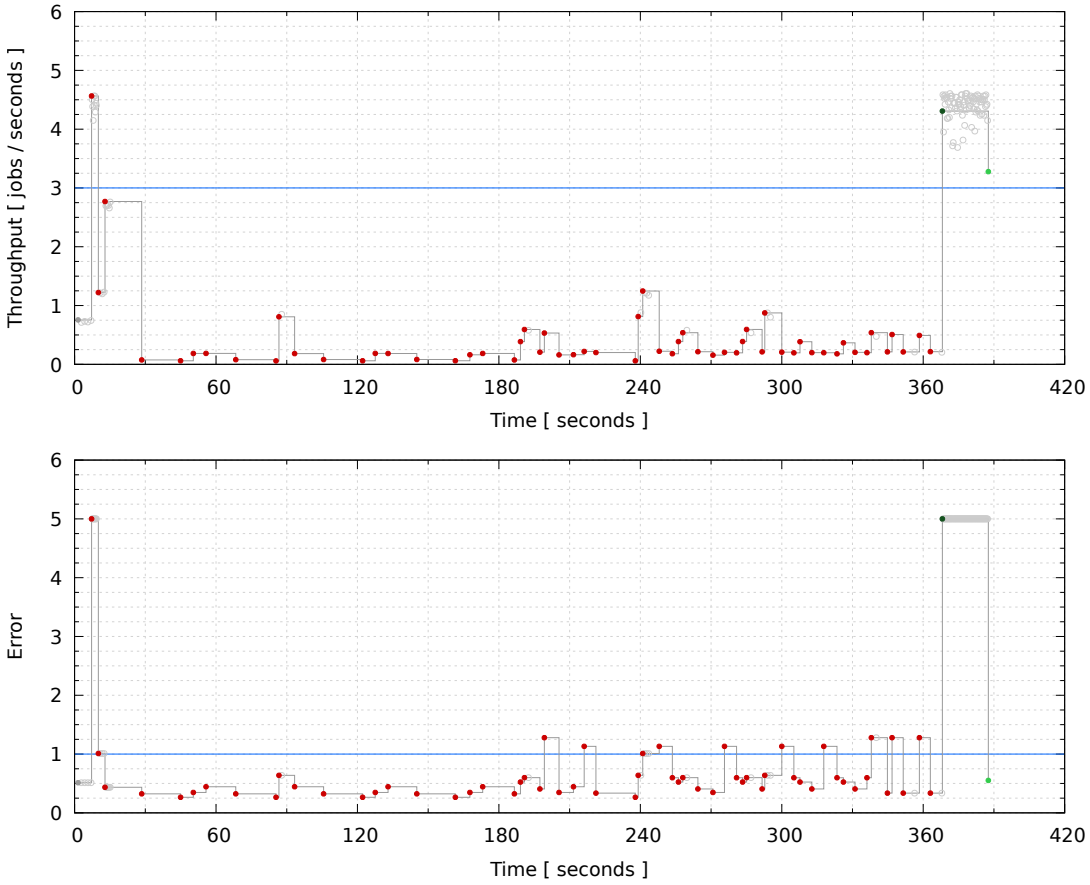


Figure 6.12: *Observed metric values during application execution; blue lines highlight boundaries of objective functions; on x-axis: time [seconds]; on y-axis: metric values*

Every couple of corresponding points in graphs of figure 6.12 represents a configuration with which the application has been executed, obtaining certain throughput and error metrics values; from a colored couple of points to the following one, the application has maintained parameters values corresponding to the former couple; grey empty couples of points indicate application execution with parameters values equal to the first previous colored couple. At the beginning, application runs with default configuration, corresponding to an error of around 0.5 and a throughput equal to about 0.75 (grey couple of points); tesiCris starts Design Space Exploration, sending DoE configurations: mARGOt, from time to time, sets application parameters with those values, so observed metrics of interest values

vary during all this phase (red couples of points); when DSE phase terminates, tesiCris sends a partial model (see 5.2.3) and it starts predicting the complete list of Operating Points: mARGOt chooses the best application configuration that fulfills objective functions, represented by dark green couple of points; throughput is around 4.25, so the first goal is respected, while error is about 5, so the second goal is not achieved; when application receives the complete model (around at 6 and a half minutes after starting time), it is set with another configuration (light green couple of points): obtained throughput is approximately 3.25 and error is about 0.5, so all objective functions are fulfilled; of course, if objective functions don't change, program keeps running with this Operating Point.

6.2.2 Swaptions application

We have tested tesiCris on a real scenario; we have used Face Centered Central Composite DoE with One Center Point as Design of Experiments and the 2nd version of implemented Generalized Linear Regression as Response Surface Methodology; due to application computing kernel (Monte Carlo simulations), we have decided to collect, for each DoE configuration, a meaningful number of repetitions, setting this value to 15.

Model prediction quality

Next figures shows results on predicted model using *deltaError* measure, as done for synthetic application analysis (see 6.2.1):

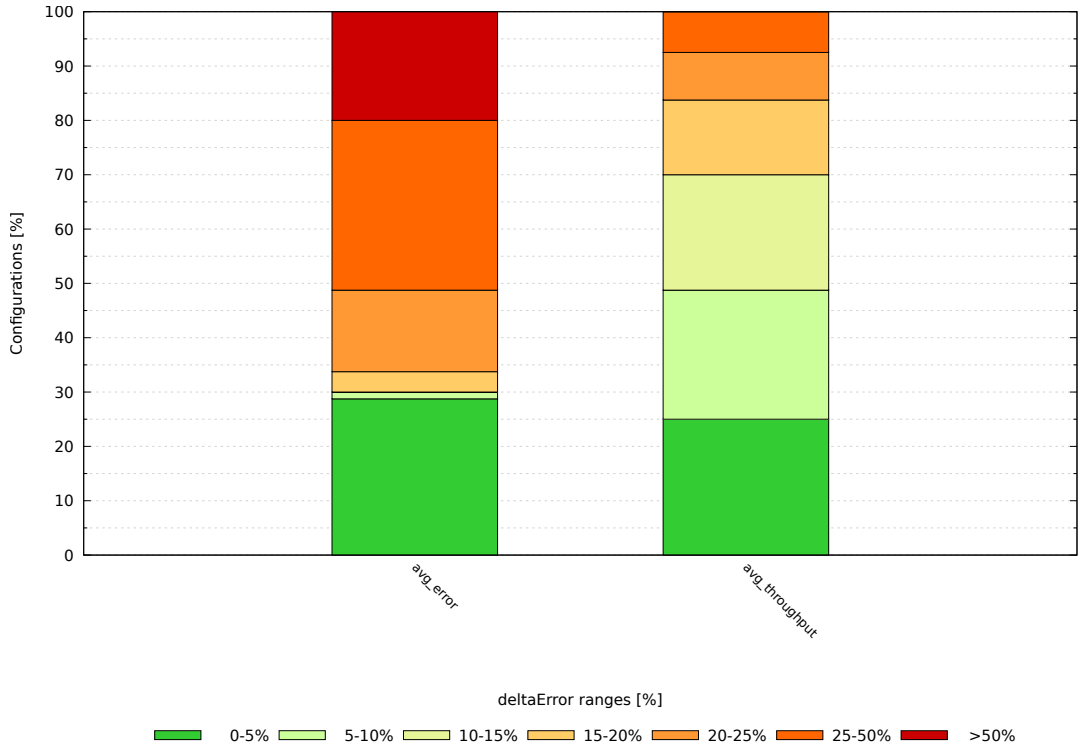


Figure 6.13: *deltaError results for both Swaptions metrics of interest; left stacked bar focuses on avg_error metric, right one for avg_throughput; on y-axis: number of configurations [%]*

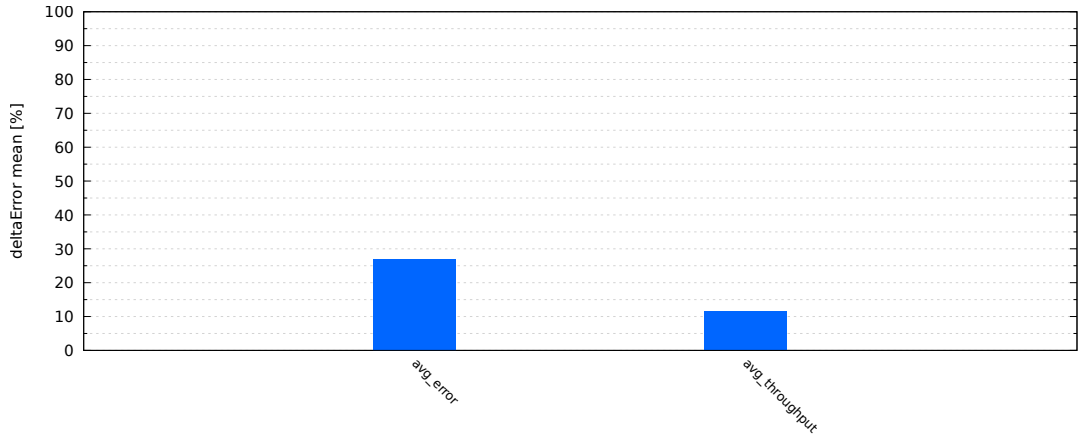


Figure 6.14: *deltaError mean values for both Swaptions metrics of interest; left bar shows deltaError mean for avg_error metric, right one for avg_throughput; on y-axis: deltaError mean [%]*

We have obtained an average *deltaError* of around 11% for *avg_throughput* and about 27% for *avg_error*, as it is shown in figure 6.14; from figure

6.13 we can see that 70% of *avg_throughput* predictions has a *deltaError* below 15%, while this percentage decreases to 30% for *avg_error* metric.

Execution times

Regarding execution times, we compare analyzed run with a Full-Factorial, so exhaustive, one in which we have collected, for each configuration, 15 repetitions for each Operating Points, as shown in figure 6.15:

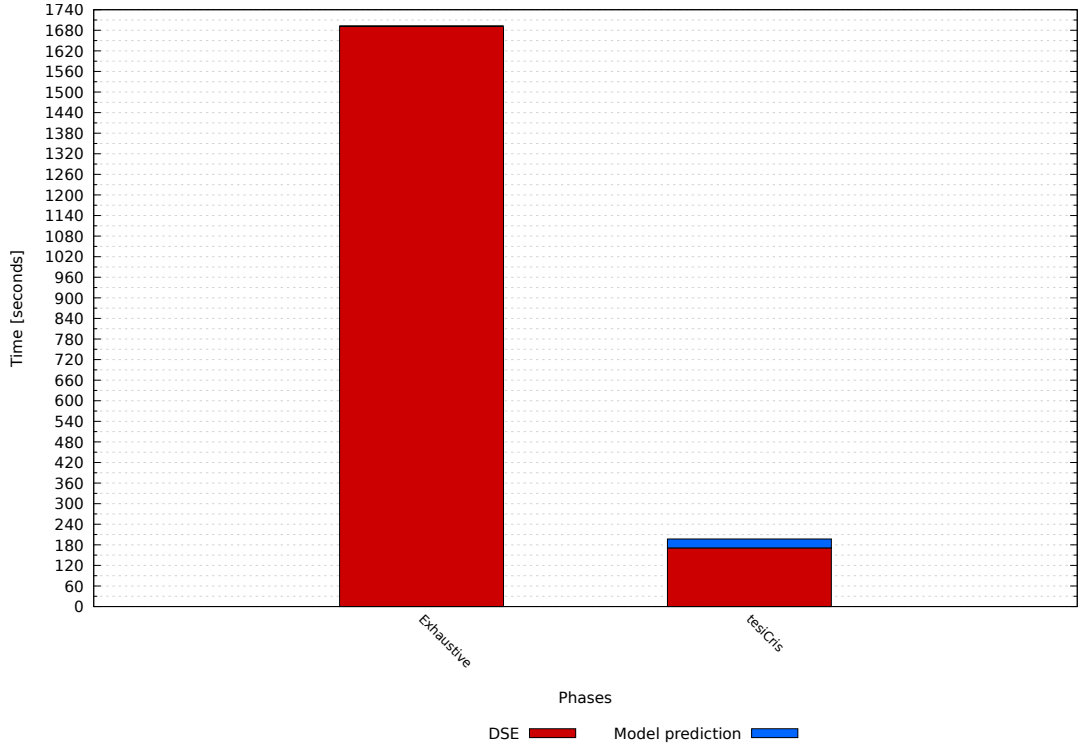


Figure 6.15: *Swaptions execution times; first bar shows elapsed time for exhaustive execution; adjacent one reveal overall time, until complete model has been predicted, with the supervise of Agorà; on y-axis: elapsed time [seconds]*

Exhaustive execution took around 28 minutes to finish, while the analyzed one less than 3 and a half minutes, so approximately 8 times less; of course, as explained before, we have not obtained precise predictions for each metrics value; nevertheless, *Swaptions* can be

executed with the assistance of tesiCris plus mARGOt autotuner and, finally, prearranged objective functions are achieved, also because mARGOt is able to collect feedback information that corrects application model.

Application behavior over time

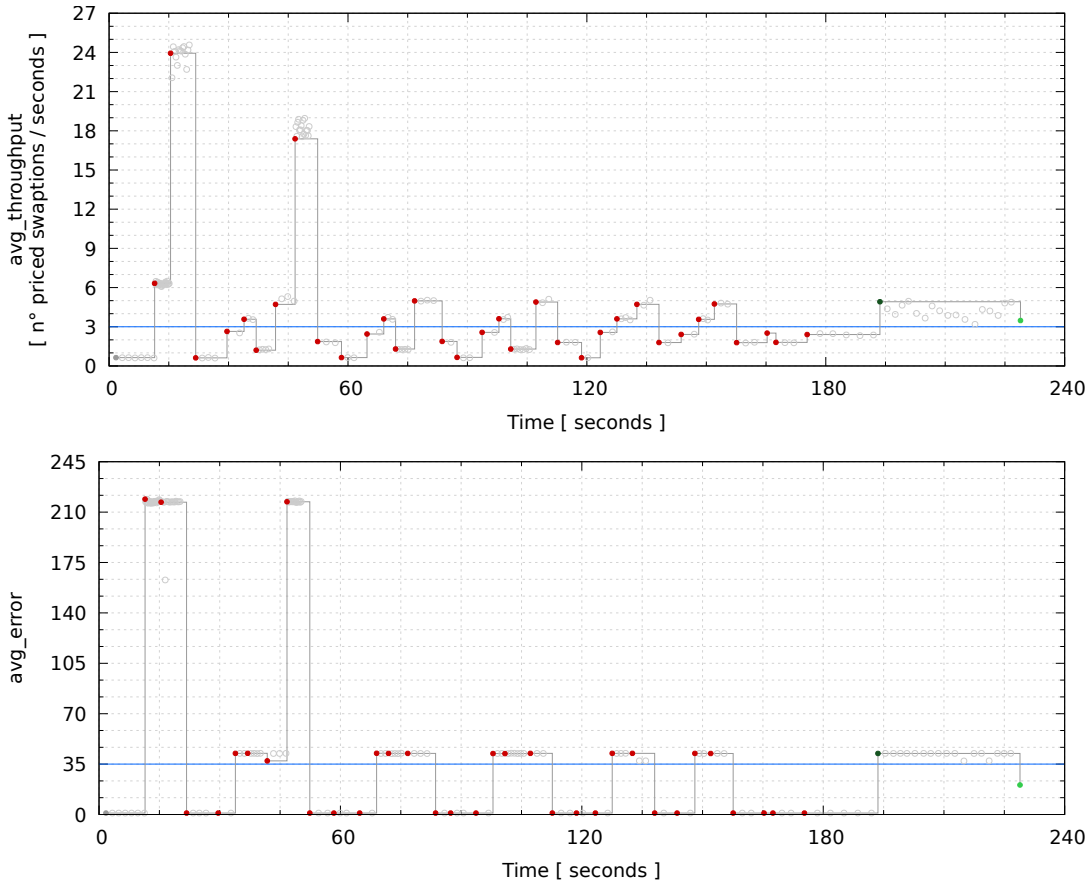


Figure 6.16: *Observed metric values during Swaptions execution; blue lines highlight boundaries of objective functions; on x-axis: time [seconds]; on y-axis: metric values*

Figure 6.16 shows application behavior during execution; we have set, as objective functions, $avg\_throughput > 3$ and $avg\_error < 35$; during Design Space Exploration phase, application is executed with Design of Experiments configurations in order to collect training data (red couples of points); around 195 seconds, DSE phase finishes and

application receives partial model (see 5.2.3); mARGOt sets up *Swaptions* with best possible parameters values with respect to goals and requirements (dark green couple of points): during model prediction phase, *avg_throughput* is about 5 (so, greater than the minimum required value 3) and *avg_error* approximately 42 (higher than 35, so the corresponding objective function is not followed); tesiCris sends complete model with predicted metrics values for each application configuration: mARGOt changes current Operating Point (light green couple of points), obtaining an $avg\_throughput > 3$ and an $avg\_error < 35$, so both objective functions are achieved.

CHAPTER 7

Conclusion and future works

Acknowledgements.

Bibliography

[1] America's data centers consuming and wasting growing amounts of energy.

[2] Hivemq - enterprise mqtt broker.

[3] Learning from data - machine learning course taught by caltech professor yaser abu-mostafa.

[4] Top500 green supercomputers.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[7] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–100. ACM, 2012.

[8] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[9] Andrew Banks and Rahul Gupta. Mqtt version 3.1.1. *OASIS standard*, 2014.

[10] Blaise Barney. Introduction to parallel computing. lawrence livermore national laboratory. *Available on https://computing.llnl.gov/ tutorials/parallel_comp*, 2012.

[11] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8. IEEE, 2012.

[12] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[14] Massimiliano Caramia and Paolo Dell´ Olmo. Multi-objective optimization. *Multi-objective Management in Freight Logistics: Increasing Capacity, Service Level and Safety with Optimization Algorithms*, pages 11–36, 2008.

[15] João MP Cardoso, Tiago Carvalho, José GF Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. Lara: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 179–190. ACM, 2012.

[16] João MP Cardoso, José GF Coutinho, Tiago Carvalho, Pedro C Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 2014.

[17] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[18] I-Hsin Chung, Jeffrey K Hollingsworth, et al. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004*

ACM/IEEE conference on Supercomputing, page 30. IEEE Computer Society, 2004.

[19] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.

[20] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31, 2006.

[21] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[22] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[23] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 173–180. IEEE, 2015.

[24] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, 1142(2011):1–12, 2011.

[25] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[26] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] André I Khuri and Siuli Mukhopadhyay. Response surface methodology. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(2):128–149, 2010.

[28] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[29] R Light. Mosquitto - an open source mqtt v3.1 and 3.1.1 broker. *http://mosquitto.org*, 2013.

[30] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, et al. Introducing the human brain project. *Procedia Computer Science*, 7:39–42, 2011.

[31] Sparsh Mittal. Power management techniques for data centers: A survey. *arXiv preprint arXiv:1404.6681*, 2014.

[32] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):62, 2016.

[33] Gordon E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[34] Mary Natrella. *e-handbook of statistical methods*, chapter 5.3, Choosing an experimental design. NIST/SEMATECH, 2013.

[35] N O'Leary. Paho - open source messaging for m2m. *Eclipse Paho's MQTT*, 2014.

[36] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, 2009.

[37] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *The International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[38] Christoph Schaefer, Victor Pankratius, and Walter Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. *Euro-Par 2009 Parallel Processing*, pages 9–20, 2009.

[39] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, et al. The antarex approach to autotuning and adaptivity for energy efficient hpc systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 288–293. ACM, 2016.

[40] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat

Avasare, Geert Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, pages 47–63. Springer, 2011.

[41] Apache Spark. Apache spark: Lightning-fast cluster computing, 2015.

[42] Balaji Subramaniam, Winston Saunders, Tom Scogland, and Wu-chun Feng. Trends in energy-efficient computing: A perspective from the green500. In *Green Computing Conference (IGCC), 2013 International*, pages 1–8. IEEE, 2013.

[43] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review*, 50(2):607–621, 2016.

[44] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[45] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[46] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[47] Lucy Zhang. Building facebook messenger, 2011.

[48] Marcela Zuluaga, Andreas Krause, and Markus Püschel. ϵ-pal: An active learning approach to the multi-objective optimization problem. *Journal of Machine Learning Research*, 17:1–32, 2016.