Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Doctoral Programme In Information Technology

Dynamic Application Autotuning for Self-Aware Approximate Computing

Doctoral Dissertation of:
Davide Gadioli

Supervisor:
Prof. Gianluca Palermo

Co-Advisor:
Prof. Cristina Silvano

Tutor:
Prof. Andrea Bonarini

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

Year 2019 – Cycle XXX

Abstract

IN the autonomic computing context, we perceive the system as an ensemble of autonomous elements capable of self-managing, where end-users define high-level goals and the system shall adapt to achieve the desired behaviour. This runtime adaptation creates several optimisation opportunities, especially if we consider approximate computing applications, where it is possible to trade off the result accuracy and the performance. Given the power consumption limit on modern systems, autonomic computing is an appealing approach to increase the computation efficiency.

I divided this PhD thesis into three main sections. The first section focuses on a dynamic autotuning framework, named *mARGOt*, which aims at enhancing the target application with an adaptation layer to provide self-optimisation capabilities at the production phase. In this context, the end-user might specify complex high-level requirements, and the proposed approach automatically tunes the application accordingly.

The second section evaluates the *mARGOt* framework, by leveraging its features in two different scenarios. On the one hand, we evaluated the orthogonality between resource managers and application autotuning. On the other hand, we proposed an approach to enhance the application with a kernel-level compiler autotuning and adaptation layer in a seamless way for application developers. The third section focuses on two application case studies, showing how it is possible to significantly improve computation efficiency, by applying approximate computing techniques and by using *mARGOt* to manage them.

Contents

CHAPTER 1

Introduction

With the end of Dennard scaling [1], power consumption limits the performance of modern systems. For this reason, there is a trend to shift the optimisation focus toward energy efficiency in a wide range of scenarios, not only related to embedded systems but also related to high-performance computing (HPC) [2].

Among all the possible directions that promise to improve the computation efficiency of a system, this thesis focuses on two approaches at the software layer. On the one hand, when application developers write the source code, the best practice is to expose implementation parameters that alter the extra-functional properties of the application, such as execution time or power consumption. We have algorithm-agnostic parameters, such as the number of software threads, the dimension of communication buffers or the tile size in a loop; but we also have algorithm-specific parameters that alter the procedure to obtain the result, such as the compression factor in an image converter application. In literature, these parameters are also named *software-knobs*, since a change on their value leads to a change in the extra-functional properties as well. If it is possible to change their configuration at runtime, they are named *dynamic knobs* [3].

On the other hand, several approaches aim at finding *good enough* results for the end-user, thus saving the unnecessary computation effort, improving efficiency. In literature, this approach is named *approximate computing*. A large class of applications implicitly expose software-knobs at algorithm-level to find accuracy-throughput tradeoffs. They might found especially in multimedia [4] and whenever it is possible to use approximation techniques such as loop perforation [5] or task skipping [6]. Since approximate computing can significantly increase the application throughput by decreasing the result accuracy [7], several works in literature investigate the possibility to use also approximate hardware accelerators [8,9].

Among the implications of this trend, the application requirements are increasing in complexity. Due to the tradeoffs created by using software-knobs and approximate computing, the end-user might have complex requirements which involve extra-functional properties (EFPs) in conflict with each other, such as power consumption, throughput, and accuracy. Moreover, these extra-functional properties might depend on the actual inputs of the application, on the available resources, and on the configurations of the underlying architecture (such as the core frequencies).

In this context, the autonomic computing field investigates how to enhance the target system with a set of *self-** properties [10], such as *self-healing*, *self-optimization* or *self-protection*. In this thesis, we focus on the *self-optimization* property, where the target system shall automatically identify and seize optimisation opportunities according to the system evolution.

1.1 Thesis Motivations

Given the vast difference in performance between software-knobs configurations, researchers have investigated several approaches for finding the ones that lead to optimal tradeoffs between EFPs of interest for end-user [11–13]. However, finding a one-fits-all software-knobs configuration is complex if we consider the system evolution. The application requirements may change according to external events. For example, end-user might have different requirements according to whether the target platform is relying on batteries or not. Moreover, there might be changes in the underlying architecture. For example, a power capper might lower the core frequencies due to thermal reasons, or the available resources might change due to workload fluctuations. Furthermore, the EFPs might have heavy input dependency. Therefore, a one-fits-all software-knobs configuration might lead to sub-optimal performance.

For these reasons, the *self-optimization* capability requires an adaptation layer that tunes the software-knobs configuration at runtime. However, how to provide to a target application the optimal software-knob configuration, according to end-user requirements and system evolution, is still an open question. This is a known problem investigated in the literature using different approaches. The work carried out in this thesis aims at advancing the state-of-the-art toward this direction.

1.2 Thesis Contributions

The main outcome of this thesis is a methodology to enhance a target application with an adaptation layer that exposes mechanisms to adapt reactively and proactively. Moreover, additional contributions of this thesis are the methodology evaluation in different contexts and the analysis carried out in real-world applications, to show how it is possible to improve computation efficiency. Furthermore, the proposed framework is a key component of the ANTAREX approach, developed in the context of the Horizon 2020 European Project "AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems". In particular, the thesis contributions are the followings:

1. The methodology implementation, named *mARGOt*, is a C++ library that is linked to the target application and works at the function level. *mARGOt* employs separation of concerns between functional and extra-functional requirement. End-user might define or change requirements at runtime, according to application phases. Moreover, by using feedback information from runtime monitors, it is possible to react to changes in the execution environment, providing to the application the most suitable software-knobs configuration. Furthermore, it leverages input features to identify and seize optimisation opportunities according to the current input. We publicly released the framework source code [14], along with user manuals, with the hope that application developers can use *mARGOt* for improving the computation efficiency.

2. A framework has been implemented for learning the application knowledge online, using a distributed approach. It uses models ensemble to increase the predictive capabilities of base models that perform out-of-sample predictions. Moreover, it uses an iterative procedure to obtain the application knowledge using as few samples as possible.

3. An experimental evaluation of *mARGOt* has been performed in a wide range of scenarios, from embedded to High-Performance Computing,

to assess the benefits of each feature of the proposed approach in real-world applications and case studies. In particular, we target a Stereo-matching application targeting an embedded platform, a Probabilistic Time-Dependent Routing application targeting an HPC node, and a Geometric Docking application targeting an HPC platform.

4. In the context of *resource consolidation*, we evaluated the orthogonality between application autotuning and resource management. In particular, we compared different policies for sharing computational resources between co-running applications. Moreover, we proposed a light-weight technique for run-time resource management based on platform sensing at the application level, leveraging *mARGOt*.

5. A framework has been proposed to automatically tune compilation flags and OpenMP parameters at the function level, in a seamless way for the application developer. Beside *mARGOt*, it leverages the compiler autotuning framework COBAYN [15] for extracting the most promising compilation fags, while it uses the LARA [16] aspect-oriented language to enhance the original source code automatically.

6. In the context of smart cities, this thesis focused on a Probabilistic Time-Dependent Routing application to show how it is possible to increase the computation efficiency, by using *mARGOt*. In particular, we analysed the effect of roads characteristic to the quality of the output, to identify and seize optimisation opportunities at runtime.

7. In the context of a drug discovery process, a geometrical docking application has been analysed for identifying software-knobs that expose accuracy-throughput tradeoffs. *mARGOt* leverage these tradeoffs, together with features of the actual input, to tune the application for respecting a requirement on given a time-to-solution for the HPC job.

In the remainder of this thesis, I will write using the first-person plural to acknowledge the support from advisor and colleagues. However, I take responsibility for all the decisions and choices described in this thesis, since I was the main investigator. The only exceptions are the works described in Chapter 5 and Chapter 7, that are a joint effort with other colleagues.

1.3 Thesis Outline

I divided this thesis into three main sections. The first section describes the proposed framework, and it represents the main outcome of the thesis since it has been continuously developed during the doctoral studies (Chapter 2-4). In particular, to introduce the reader to the field of autonomic computing, Chapter 2 describes the background and defines the main concepts used in this thesis. Moreover, it provides an overview of the state-of-the-art, highlighting the contribution of this thesis. Then, Chapter 3 describes in details the proposed approach, defining the methodology and the *mARGOt* implementation. To better clarify the required integration effort and to show the framework workflow, it also provides an example of how to leverage *mARGOt* features in a toy example. Chapter 4 evaluates the benefits and limitations of the proposed approach, by measuring the introduced overheads and by evaluating each feature exposed by the adaptation layer, highlighting the required changes in the application source code. This first section relates to contributions 1-3.

The second section (Chapters 5,6) describes the exploitation of the framework in two scenarios outside application autotuning. On the one hand, Chapter 5 evaluates the orthogonality between resource managers and application autotuners, and it shows the benefits and limitations of using *mARGOt* as a lightweight resource manager (contribution 4). On the other hand, Chapter 6 proposes a structured approach for the runtime selection of the most suitable application configuration concerning compiler flags and parallelism parameters of the OpenMP runtime, in a transparent way for application developers (contribution 5).

The third section (Chapters 7,8) shows how it is possible to use *mARGOt* for leveraging the tradeoff between the EFPs of interest in two interesting application case studies. Chapter 7 focuses on tuning a server-side car navigation system, in the context of smart cities (contribution 6). While Chapter 8 describes how we applied approximate computing techniques in a geometrical molecular docking application, in the context of a drug discovery process, for improving computation efficiency, managed by *mARGOt* (contribution 7).

Finally, Chapter 9 concludes the thesis, by summarising the findings and limitations of the proposed approach and by stating recommendations for future works.

Part I

The proposed framework

CHAPTER 2

Previous work

The main focus of this thesis is an approach to dynamically autotune an application, implemented as the *mARGOt* autotuning framework. This chapter provides at first an introduction to the related research field, and it defines a common terminology used to compare *mARGOt* with the state-of-the-art. Then, it describes in more details the most similar works, and it identifies the differences with *mARGOt*. The summary of this chapter highlights the contributions of this thesis.

2.1 Background and definitions

The research carried out in this thesis belongs to the autonomic computing field [10]. In this context, we perceive a computing system as an ensemble of autonomic elements that are able of self-management, without a human-in-the-loop. According to the proposed vision, an autonomic element must have the self-configuration, self-optimization, self-healing and self-protection abilities. Self-configuration is the ability to incorporate in the system new components whenever they become available, as in the Rainbow framework [17]. Self-healing is the ability to recover from a hardware or software failure, as proposed in [18]. Self-protection is the ability to defend itself against malicious attack or failures not corrected by any self-healing mechanism, as proposed in [19]. Eventually, self-optimization is the ability to identify and seize opportunities to improve the application performance or efficiency. How to provide any of the self-* abilities within a single framework and without losing in generality is still an open question. Since the goal of *mARGOt* is to enhance an existing application with an adaptation layer that provides the ability of self-optimization, we focus on works related to the latter property. Previous surveys [20, 21] provide a more detailed overview of the field, regarding the other self-* abilities.

The definition of a system in the context of autonomic computing involves both, hardware and software. Therefore, several works in literature aimed at optimising the system performance or efficiency. In this thesis context, we might divide them into two main categories: *resource managers* and *application autotuners*. Resource managers address system adaptability through resource management and allocation. For example in data centre context [22, 23], in the grid computing context [24], in multi/many core node contexts [25–27] or for embedded platforms [28, 29].

Application autotuners work at the software level, leveraging the assigned resources to reach end-user requirements. Therefore, they take orthogonal decisions. Before discussing the related work in literature, it is important to clearly define the key concepts behind the methodology presented in this thesis. With the term application, we may refer to any software that is possible to execute on the target architecture. However, we consider only applications that perform an elaboration and that do not require human interaction, such as a video encoder, a navigation system or scientific applications. Moreover, end-users or system administrators may have preferences or requirements on the application extra-functional properties (EFPs), such as execution time, energy consumption or quality of the results. For example, the user of a video encoder application would like to convert a video streaming with the highest quality, provided a throughput of at least $25fps$; or the administrators of a navigation system would like to minimize the energy consumption while respecting a Service Level Agreement on the response time and quality of the results. We refer to the set of EFPs relevant for the end-user or system administrator as *metrics*, defining the application performance as a vector of values.

A large class of applications expose tunable parameters that alter the application performance, named *software-knobs*. We may have application-specific software-knobs and application-agnostic software-knobs such as tile size or the number of Monte Carlo simulation. The main idea is that a change in the software-knobs configuration leads to a change in the application performance as well.

The main goal of an application autotuner is to automatically tune the software-knobs according to end-users or system administrator preferences or requirements. The main challenge is that the relation between a software-knobs configuration and application performance is unknown and usually depends also on the underlying architecture and on the current input. For this reason, it is possible to use the characteristics of the current input, such as its size or autocorrelation, to better describe the relationship with application performance. This set of values is named *input features*. The representation used by application autotuner to describe the relation between software-knobs configurations, input features and the application performance is named *application knowledge*.

2.2 Application autotuning

In synergy with resource managers, application autotuning frameworks aim at selecting the most suitable configuration of the software-knobs to leverage the assigned resources. Among these approaches, we have *static autotuners* which select the most suitable configuration before the production phase, and we have *dynamic autotuners* which select the most suitable configuration during the production phase. The following sections describe the most related work in literature.

2.2.1 Static autotuning frameworks

Static autotuners target software-knobs that tailor the application for the underlying architecture, such as tiling size, loop unrolling factor, compiler options and algorithm selection. This tailoring process implies that static autotuners have to consider a fair amount of knobs with a large, possibly unbounded, domain of possible values. The Design Space (DS) of an application grows exponentially, making a full-factorial Design Space Exploration (DSE) in this context unfeasible. Therefore, static autotuning frameworks are typically designed to find the configuration that maximises/minimise a utility function in a reasonable amount of time. Even if a fraction of static autotuners perform such exploration at runtime, once they settle with an optimal configuration they are not willing to change it anymore.

As examples of static autotuning frameworks, we may consider the following works. ATune-IL [30] focuses on pruning the Design Space before of the tuning step. On the main hand, it prunes the space by handling dependencies between software-knobs. On the other hand, it analyses the code structure to split the set of software-knobs into independent regions that might be tuned separately. This approach focuses on minimising the execution time. AutoTune [31] targets multi-node applications, and it leverages the Periscope framework [32] to measure the execution time. In their work, they tune pipeline parameters such as buffer sizes, OpenHMPP [33] related parameters and MPI related parameters. However, it is possible to expand the framework using plugins to consider an arbitrary class of software-knobs. QuickStep [34] and Paraprox [35] targets parallel regions of an application and they perform code transformation without preserving the code semantics. The rationale behind this choice is to expose and leverage the accuracy-throughput trade-off automatically. In particular, they minimise the execution time given a threshold on the minimum accuracy. OpenTuner [36] explicitly addresses the problem of exponential growth in the complexity for exploring the DS, by using an ensemble of search algorithms. Since each algorithm shines for a particular class of applications, OpenTuner uses a multi-armed bandit solver to find and exploit the best search algorithm for the given application. Moreover, it is possible to define the extra-functional requirements as a constrained multi-objective optimisation problem. PowerGAUGE [37] manipulates the assembly code of an application, using a genetic algorithm, to expose and optimise the accuracy/performance trade-off. ATF framework [38] is a language agnostic autotuning framework that enables a user to tune the application according to a constrained multi-objective optimisation problem. Moreover, it enables the user to specify complex dependencies between software knobs values. Recent work [39] investigates the effect of tuning independent regions of code which share common software-knobs, e.g. the number of threads. In particular, they show how a global tuning can further optimise the application with respect to a local tuning of independent regions of code.

Moreover, in the context of High-Performance Computing, there are several autotuning frameworks targeted at specific tasks. ATLAS [40] for matrix multiplication routine, FTTW [41] for FFTs operations, OSKI [42] for sparse matrix kernels, SPIRAL [43] for digital signal processing, CLTune [44] for OpenCL applications, Patus [45] and Sepya [46] for stencil computations, are some examples in this area.

These works are typically employed in a predictable execution environment, and they usually target a different class of software-knobs with respect to dynamic autotuners. Indeed, by choosing a configuration at design time, it is not possible to react to changes in either the application requirements or of the observed performance. Moreover, the decision algorithm is not able to leverage input features.

2.2.2 Dynamic autotuning frameworks

The defining characteristic of dynamic autotuning frameworks is that they can continuously tune the software-knobs configuration at runtime. The main idea is to leverage information about the actual execution context, rather than the average behaviour when they decide which is the most suitable software-knobs configuration to apply. Usually, they rely on application knowledge to predict the behaviour of a configuration and to drive the decision process. In this section, we describe the most relevant work for the methodology proposed in this thesis.

Configuring an application at runtime has been an appealing idea investigated in literature for a long time. For example, the ADAPT framework [47] aims at decoupling run-time code generation from the selection of the best variant. It monitors execution time for evaluating variants and for identifying hot-spots in the code. It uses a remote optimiser to generate versions of a variant, applying different optimisation techniques. Locally it uses rules to flag specific transformations as stale and therefore avoids their usage. For example, if the number of available cores is not greater than one, parallelisation is not used.

Moreover, the work that proposes the ABLE framework [48] shows how it is possible to derive an autotuner. This example targets the Lotus Notes servers, and it tunes two software-knobs to maintain the CPU and memory utilisation below the desired level. At first, it generates a synthetic workload to build application knowledge and then it leverages control theory [49] to drive the selection of the configuration. Although the example framework adapts the application at run-time, it focuses more on providing self-protection abilities rather than self-optimization. Since the employed autotuner is tailored for a specific application, later work [50] formalise a blueprint for a generic auto-tuner based on control theory, highlighting limitations and challenges.

Control theory is not the only scheme for adaptation investigated in the literature. Indeed, a previous work [51] proposes a more proactive approach based on machine learning. It leverages features of the actual input to select the most suitable algorithm version. At first, it uses domain knowledge to identify the features of an input which are related to the application execution time. Then, starting from a real-world problem, it proposes to generate synthetic inputs to train and validate a Bayesian network to select the most promising version. On the production phase, the approach leverage application knowledge to adapt at run-time according to the actual input. Since the focus of this pioneering work was to learn the effects of actual inputs

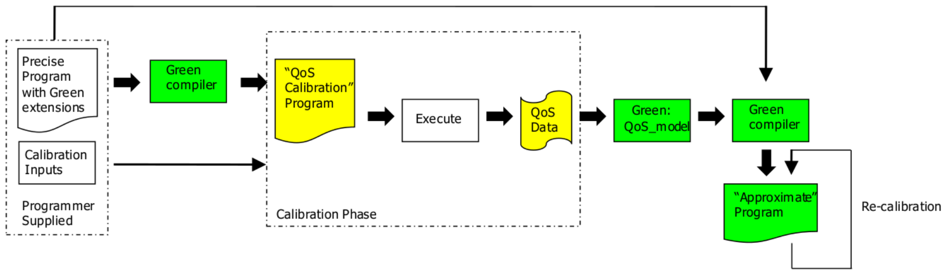


Figure 2.1: *The GREEN framework overview. Image from [55].*

at design time, it addresses only one metric, and it targets a predictable execution environment.

More recent works evaluate the possibility of relaxing the constraint on functional correctness to improve efficiency. The rationale is that we may tolerate a lower accuracy of the results as long as the output of the computation is useful for the end-user. A large class of applications implicitly define application-specific software-knobs that relate with the output quality [52], for example in the context of multimedia. It might be a complex task to identify such software-knobs, therefore works in literature describes techniques to expose such tradeoffs by failing task on purpose [6] or by skipping iterations of a loop [5]. A later work [7] investigates the effect of loop perforation using a large set of applications from the PARSEC benchmark [53], showing how a small loss in accuracy may lead to a significant increment in performance.

Following this trend, the Sage framework [54] investigates three source transformations that expose accuracy-throughput tradeoffs, targeting CUDA kernels. It takes as input the original CUDA kernel and a metric that represents elaboration quality. In the first step, Sage analyses the kernel code and find opportunities to apply the proposed transformations, generating different tunable versions of the code. In a second step, it uses a greedy approach to select the kernel version and to tune its parameter to minimise the execution time given a lower bound on the quality. At runtime, it periodically monitors the execution time and quality. If it detects a violation of the target output quality, then it selects a more accurate configuration. Since it focuses on a specific class of software-knobs, the integration effort is negligible.

One of the pioneering framework designed to harness the throughput-accuracy tradeoff for a generic application is the Green framework [55]. Figure 2.1 provides an overview of the approach. After an integration step, the Green compiler performs at first a DSE to generate QoS Data, and an external program in MATLAB performs curve fitting and interpolation. In the second step, the Green compiler generates the adaptive executable taking into account the desired QoS requirements. At run-time it periodically measures QoS, triggering a re-calibration if the observed value differs with respect to the expected one. The Green default re-calibration increases or decreases the QoS requirements, however, the user may define its re-calibration function.

Another interesting example of a framework that manages the accuracy-throughput trade-off is PowerDial [3]. It takes as input the source code of the application, the command lines options, a representative input set, and an output abstraction to measure the accuracy. In the first phase, it leverages llvm to identify, from the command lines options, the actual variables in the source code that alters the extra-functional properties of the application. In the second phase, it performs a DSE to sort the software-knobs configuration according to a speed-up with respect to the baseline throughput, i.e. the default configuration. In the third phase, it generates a binary with a manager based on control theory, that selects the speedup required to reach the target throughput. It uses the Heartbeats framework [56] to measure the actual throughput, and it uses the application knowledge to convert the control signal to a software-knobs configuration. PowerDial defines the throughput goal at compile time, and it targets application with homogeneous inputs or with few abrupt changes. Moreover, the designed controller manages a tradeoff between the two metrics. To overcome this limitation, later work [57] investigate an approach to extend the controller to handle a trade-off between several metrics, by introducing limitations and assumption on the software-knobs.

SmartConf [58] uses a similar adaptation scheme. However, it focuses on Java server application, such as Cassandra or Hadoop, which expose a large number of command line options that affect the system performance. Indeed, a wrong configuration of those command line options may lead to poor performance or crash due to memory usage. In a first phase, they use an experimental campaign to model the relationship between a command line option and the related metric. SmartConf leverages control theory to stabilise the metrics to a target value, with some assumptions regarding their interaction with the command line options. Although SmartConf adapts the application at run-time, it focuses more on providing self-protection abilities rather than self-optimization.

Among previous work that manages the throughput-accuracy trade-off, the IRA framework [59] proposes an interesting approach to adapt an ap-

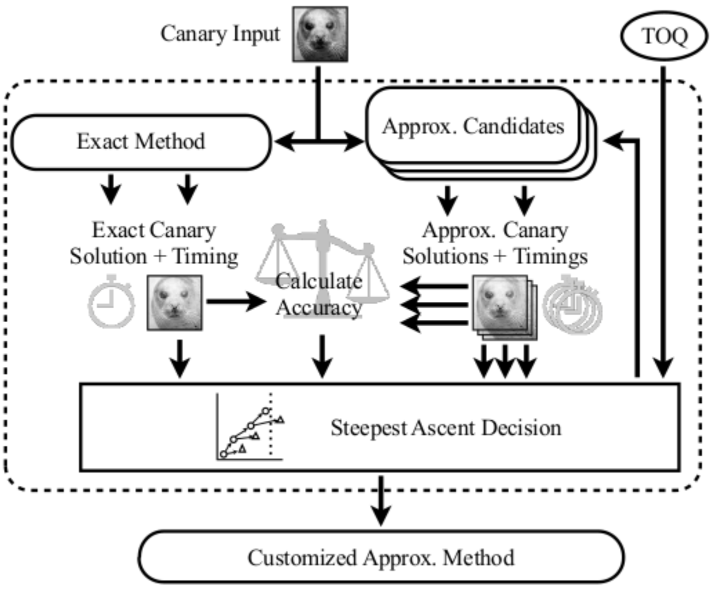

Canary Input
TOQ
Exact Method
Approx. Candidates
Exact Canary Solution + Timing
Calculate Accuracy
Approx. Canary Solutions + Timings
Steepest Ascent Decision
Customized Approx. Method

Figure 2.2: *The IRA framework overview. Image from [59].*

plication at run-time. Figure 2.2 provides an overview of the framework. It investigates several features of the input, such as the mean value or its autocorrelation, to generate a canary input. The latter is the smallest sub-sampling of the actual input which has the same property as the original input. It uses a statistical hypothesis test to perform such an evaluation. However, the related paper investigates techniques for sub-sampling only images or matrix-like inputs. IRA uses the canary input to perform a DSE for each input, selecting as the most suitable software-knobs configuration the fastest one within a given bound on the minimum accuracy. Then it uses the best configuration with the actual input to produce the desired output.

A fascinating work that leverage input features is Capri [60], which inspired us in the *mARGOt* development. At design time it uses a set of representative inputs to model a cost metric (e.g. execution time or energy) and an error metric as a function of software-knobs configuration and input features. The controller that selects the most suitable configuration is based in Valiant's probably approximately correct (PAC) theory [61]. In particular, it aims at finding at runtime, the software-knobs configuration that minimises the cost function given an error bound and a probability that the bound is satisfied according to the representative inputs. Since Capri does not address stream applications, the work is not investigating any reaction mechanism to adapt the application knowledge according to system evolution. Due to the chosen formulation of the problem, the feasible region given by the error function does not depend on the actual input. This assumption might miss optimisation opportunities when input features are related to the error, for example in Monte Carlo algorithms.

A rather different approach with respect to the previous ones is Anytime Automaton [62]. It suggests source code transformations to re-write the application using a pipeline design pattern. The idea is that the longer the given input executes in the pipeline, the more accurate the output becomes. The work targets hard constraint on the execution time, interrupting the algorithm when it depletes the time budget. In this way, it is possible to have guarantees on the feasible maximum accuracy.

Beside frameworks that provide an adaptation layer to the target application, Petabricks [63] is a language to expose algorithmic choices. The Petabricks framework (compiler and autotuner) analyses the code and generates a configuration file that selects the fastest algorithm and software-knob configuration according to the input size. The Petabricks run-time can dynamically manage the application parallelism, taking into account the input size. Since Petabricks is a language, the strategy to select the algorithm version and software-knobs are hard-coded in the generated executable. In later works, the framework has been enhanced to leverage the accuracy-throughput trade-offs at the tuning phase and to check the quality level at run-time [64]. In particular, the Petabricks compiler emits code to check the accuracy of the output and if it is below the threshold, it will re-execute the algorithm with next higher level of accuracy or execute user code. In a more recent work [65], Petabrick has been further enhanced by taking into consideration also input features, besides its size, in the tuning process at design time. At runtime, Petabricks classify an input based on known clusters features, and it selects the most suitable algorithm and configuration accordingly. The proposed framework is interesting indeed, however, it generates the adaption strategy at design-time without preserving the application knowledge. Thus, it is not flexible to changes on requirements, and they assume a predictable execution environment.

On the opposite side, Siblingrivarly [66] uses the Petabricks framework, but it targets a very unpredictable execution environment. In particular, it partitions the available cores in two identical groups. The first group experiments new algorithms and configurations, using Petabricks and a genetic algorithm for exploring the Design Space. The second group always choose the safest configuration that minimises the execution time given a bound on the minimum accuracy. In this way, it is possible to react to changes in the execution environment.

Table 2.1: *Classification of related work according to the considered metrics of interest*

Category	*Previous Work*
One metric	Voss [47], Guo [51], Ansel [63]
Two metrics	Samadi [54], Baek [55], Hoffmann [3], Laurenzano [59], Miguel [62], Ansel [64], Ding [65], Ansel [66], Sui [60]
Arbitrary metrics	Filieri [57]

2.3 Comparison with the state-of-the-art

The previous section provided a review of the literature related to autonomic computing, focusing on application autotuners. In our opinion, all the work that we described earlier are indeed interesting, and they have provided contributions to the field, due to their unique point of view on how to provide the self-optimization ability to a target application. This section aims at highlighting the contribution of this thesis, by comparing the more related approaches known in the literature, according to research questions that define our point of view and drove the *mARGOt* development.

2.3.1 What are the metrics of interest?

The main goal of this thesis is to enhance an application with an adaptation layer that provides self-optimization capabilities. Therefore, the first question aims at classifying related work in literature according to the metrics involved in the optimisation process. Usually, the term performance is synonymous with throughput or execution time since these are typically the most critical metrics for end-users. However, if we would like to leverage the benefits of approximate computing and given that the used or dissipated power limits the performance of a system [2], the throughput is seldom enough for describing the application performance. Indeed, by addressing additional metrics (such as result accuracy, energy consumption and resource usage), we can define several trade-offs that might be of interest for end-user, considering the recent shift toward efficiency in a wide range of contexts, not only related to embedded platforms, but also on High-Performance Computing.

Table 2.1 shows the literature classification according to three main categories. The first category represents works that aim at decreasing the execution time, or in general at minimising/maximising a single metric. Typically, in this category belong pioneering work that assesses the benefits of adapting at runtime. The second category represents works that consider a tradeoff between two metrics, typically a cost and an error metric. Usually, these works define the application requirements as a minimisation (maximisation) problem of one metric, given a constraint on the other metric. In particular, Sage [54], Green [55], IRA [59], Petabricks [64, 65], Siblingrivarly [66] and Capri [60] maximise the throughput given a lower bound on accuracy. On the contrary, PowerDial [3] and Anytime Automaton [62] maximise the accuracy given a lower bound on the throughput. The third category represents works that consider several tradeoffs between an arbitrary number of metrics. The only work that considers more than two metrics is the blueprint framework analysed by Filieri et al. [57]. However, it introduces limitations on the number of metrics according to the number of software-knobs. Moreover, it relies on assumptions about the relations between software-knobs and metrics.

Relation with the proposed methodology

The autotuning framework proposed in this thesis belongs to the third category. Indeed, one of the key design goals of *mARGOt* is flexibility, enabling end-user to define application requirements as a constrained multi-objective optimisation problem, with an arbitrary number of metrics of interest. Moreover, we provide the possibility to consider software-knobs in the objective function and in the constraints definition. For example, to limit the number of software threads according to the assigned resources. Using its flexibility, we applied *mARGOt* in a wider range of scenarios, and it makes room for a broader range of adaptation requirements. For example, by taking into account accuracy, execution time, resource usage and energy consumption. On the other hand, defining the application requirements as a predefined optimisation problem limits the applicability of the approach. For example, the applicability of a significant fraction of the related works that belong to the second category depends on whether the target application has a constraint on the throughput or on the accuracy.

2.3.2 How does it react to changes during the application evolution?

One of the main benefits of delaying the choice of most suitable software-knobs configuration at the production phase is that it provides the opportunity of reacting to changes in the application knowledge or requirements. Given that the application performance typically depends on the underlying architecture configuration, such as the frequency of the cores, it is possible that during the production phase that configuration changes. For example, if a power capper throttles the core frequency due to thermal reasons.

Table 2.2: *Classification of related work according to adaptive reaction scheme*

Category	*Previous Work*
None	Guo [51], Ansel [63], Sui [60]
Accuracy	Samadi [54], Baek [55], Ansel [64], Ding [65]
Knowledge	Voss [47], Hoffmann [3], Filieri [57], Laurenzano [59]
Knowledge and Requirements	Miguel [62], Ansel [66]

In this case, the application knowledge is no more accurate, and it might lead the autotuner to select a software-knob configuration that is no more able to deliver the requested performance. Moreover, given that application performance might be input-dependent, each abrupt changes in the input might lead to a violation of the application requirements. For example, if we consider a video streaming application, the quality metric is typically related to the video evolution. If we can monitor the error metric, we might react to abrupt changes in the video, instead of relying on a conservative sub-optimal configuration.

Furthermore, application requirements may change according to phases of the application. For example, suppose that we are considering a video surveillance application deployed either on a drone or a battery powered surveillance system. In this context, the end-user would like to execute the application with low power requirements if nothing is interesting in the scene, while switch to a more accuracy oriented requirements otherwise.

Table 2.2 shows the literature classification according to four main categories. Even if some approaches could in principle react to changes in the application requirements or knowledge, our classification considers only the reaction mechanisms explicitly addressed or investigated in the related article. The first category represents works that rely on a predictable execution environment, and therefore they do not provide any adaptive reaction scheme. In this category falls pioneering works or frameworks that do not focus on streaming application. The second category represents works that check at runtime if the accuracy differs from the expected value. In this case, the adaptation policy is to select a more accurate configuration or to run a user-defined code, relying on a trial and error approach or offloading the task to application developers. In the third category we have works that provide mechanisms for reacting to changes in the expected behaviour using a more structured approach: ADAPT [47] uses a rule-based system to flag configurations as “stale” and therefore not eligible; PowerDial [3] and its enhancement [57] uses control theory to adapt; while the IRA framework [59] performs a DSE using a “smaller” input. The last category rep-

Table 2.3: *Classification of related work according to proactive adaptation scheme*

Category	*Previous Work*
Proactive	Guo [51], Laurenzano [59], Ansel [63], Ansel [64], Sui [60], Ding [65]
Non-proactive	Voss [47], Samadi [54], Baek [55], Hoffmann [3], Filieri [57], Ansel [66], Miguel [62]

resents works that also reacts to changes in the application knowledge.

Relation with the proposed methodology

Due to the application knowledge representation, *mARGOt* provides a reaction mechanism to adapt according to changes in application knowledge or requirements. Both of them might be defined or changed at runtime according to application phases. Moreover, *mARGOt* uses telemetry information from monitors, to adjust application knowledge if the observed metrics value differs from the expected ones. Therefore, the framework presented in this thesis belongs to the fourth category.

2.3.3 Is it able to leverage input features?

Given our definition of application, its performance usually depends on features of the input, such as its size. Unless the autotuner provides a mechanism to adapt proactively, it must select a configuration considering the average behaviour or selecting a more conservative one. This approach may lead to sub-optimal behaviours. If we focus on streaming applications with few abrupt changes in the input features, such as a multimedia application, adapting using a reaction scheme may suffice. However, if we consider applications that elaborate a set of input without any clear relation between them, such as a High-Performance Computing application, we need to take proactive decisions.

Table 2.3 shows the literature classification according to two main categories: whether the proposed approach provides a mechanism to adapt proactively, leveraging input features, or not. From this classification, we may notice how autotuning frameworks that leverage proactive adaptation have limited reaction mechanisms and vice-versa, due to their different approach for providing self-optimization capabilities. The only exception is the IRA framework [59] since it performs a DSE using the canary input as a proxy for the actual input. However, it is not trivial to build canary inputs for heterogeneous data structures that are not matrix-like, limiting its applicability. For example, in the case study of a molecular docking application

Table 2.4: *Classification of related work according to the integration effort*

Category	*Previous Work*
Low	Voss [47], Samadi [54], Hoffmann [3]
Mild	Guo [51], Laurenzano [59], Sui [60], Baek [55], Filieri [57]
High	Ansel [63], Ansel [64], Ansel [66], Miguel [62], Ding [65]

considered in Chapter 8, it is complicated to define a sub-sampled input due to relations between the input data.

Relation with the proposed methodology

The framework proposed in this thesis can leverage input features to adapt proactively. Therefore *mARGOt* belongs to the first category.

2.3.4 What is the integration effort?

From the application developer point of view, the effort required to integrate a dynamic autotuning framework in the target application matters. Even if application developers are the ones who are in charge of writing the source code, this activity is often performed in cooperation with domain experts or end-users, especially in the High-Performance Computing context. Given that the main goal of the approach proposed in this thesis is to enhance an existing application, the integration effort was a crucial point during the *mARGOt* development. Table 2.4 shows the literature classification according to three different categories. On the one hand, the first category represents all the works that provide mechanisms to automatically apply the approach, requiring a minimal integration effort from the application developers. On the other hand, the third category represents all the approaches that require a massive refactor of the source code or a porting of the application in a new language. Due to the diversity of the approaches and due to the fact the integration effort is often deemed as an implementation detail and thus omitted from the related paper, it is complicated to define objective criteria for a more fine-grained classification of the remaining category. Therefore, we consider the other frameworks in an intermediate category, requiring a mild integration effort.

Relation with the proposed methodology

From the methodology point of view, to minimise the intrusiveness of the proposed approach, we designed *mARGOt* as a wrapper for the managed regions of code. Moreover, to enforce the separation of concerns between functional and extra-functional requirements we provide a tool that automatically generates the required glue code, starting from an XML configuration file of extra-functional concerns. In particular, the glue code defines a high-level interface composed of few functions that hide as much as possible implementation details of the autotuning framework. Although we minimised the effort required to use *mARGOt*, the proposed framework belongs to the second category.

In Chapter 6 we present a possible workaround to eliminate the *mARGOt* integration effort from the application developer. However, it targets a particular set of software-knobs, and an extension to a generic class of software-knobs is not possible, due to design choices.

2.4 Summary

Given the limitations emerged in the literature analysis done in this chapter, the approach proposed in this thesis tries to overcome them by introducing the following contributions:

- Flexibility to express application requirements has been one of the critical points on the methodology. In *mARGOt* application requirements are expressed as a constrained multi-objective optimisation problem, with an arbitrary number of constraints, and it might address an arbitrary number of EFPs as well.

- The main benefits of dynamic autotuners are due to the possibility to leverage the actual information rather than relying on the expected average case. For this reason, *mARGOt* provides mechanisms to react to changes in the application performance and requirements. Moreover, it also provides a mechanism to adapt proactively according to input features.

- From the implementation point of view, the effort to integrate *mARGOt* in the target application is a key factor for the application developers. For this reason, we tried to minimise as much as possible the number of lines to change, and we designed the interface as a *wrapper* around the managed region of code; therefore limiting the intrusiveness.

CHAPTER 3

Dynamic Autotuning Framework

This chapter describes the methodology proposed in this thesis and its implementation. At first, we provide an overview of the framework, and we define the optimisation problem that *mARGOt* aims to solve. Then we describe in details the framework components, highlighting design choices. The integration workflow is then discussed along with a summary of the framework main features.

3.1 Framework overview and problem definition

Figure 3.1 shows an overview of *mARGOt* and how it interacts with an application. To simplify the description of the autotuning methodology, we consider an application that is composed of a single phase. However, *mARGOt* is designed to manage different phases, or *blocks of code*, independently. Each phase is composed of a single kernel g that elaborates an input i to generate the desired output o. Moreover, we assume that the kernel algorithm exposes software-knobs that alter its EFPs, such as the number of Monte Carlo simulations or the parallelism level. Let $\overline{x} = [x_1, \ldots, x_n]$ the vector of software-knobs, then we might define a kernel as $o = g(\overline{x}, i)$. In this chapter, we assume for simplicity that the application is composed

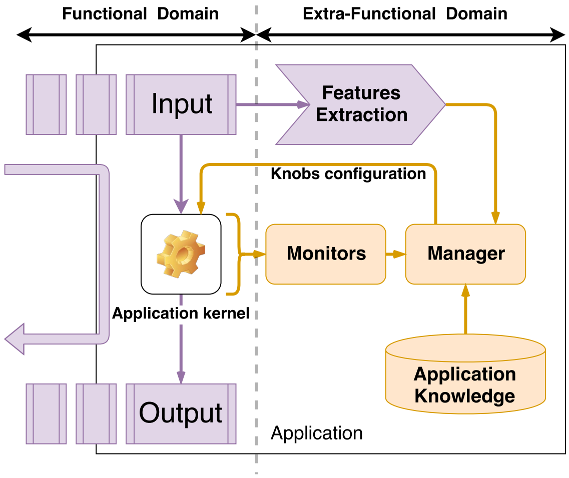


Figure 3.1: *Global architecture of the proposed framework. Purple elements represent application code, while orange elements represent mARGOt high-level components. The black box represents the executable boundary.*

of only one kernel. However, we might extend the latter definition to the whole application, as a composition of several independent phases.

Within this abstraction, we define the end-user requirements as follows. We denote the metrics of interest (i.e. EFPs) as the vector $\overline{m} = [m_1, \ldots, m_n]$. Suppose that the application developers can extract features of the current inputs, for example, the ones analysed in IRA [59]. We denote such properties as the vector $\overline{f} = [f_1, \ldots, f_n]$. The end-user can define the application requirements as in Equation 3.1:

$$\begin{aligned} \max(\min) \quad & r(\overline{x}; \overline{m} \,|\, \overline{f}) \\ \text{s.t.} \quad & C_1 : \omega_1(\overline{x}; \overline{m} \,|\, \overline{f}) \;\propto\; k_1 \quad with\ \alpha_1\ confidence \\ & C_2 : \omega_2(\overline{x}; \overline{m} \,|\, \overline{f}) \;\propto\; k_2 \\ & \ldots \\ & C_n : \omega_n(\overline{x}; \overline{m} \,|\, \overline{f}) \;\propto\; k_n \end{aligned} \tag{3.1}$$

where r denotes the objective function (named *rank* in *mARGOt* context), defined as a composition of any of the variables defined in $\overline{x}$ or $\overline{m}$, using their mean values. Let C be the set of constraints, where each C_i is a constraint expressed as the function ω_i, defined over the software-knobs or the EFPs, that must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$ with a threshold value k_i and with a confidence α_i (if ω_i targets a statistical variable). Since we are agnostic about the distribution of the target parameter, the

```xml
<?xml version="1.0" encoding="UTF-8"?>
<points version="1.3" block="example">
  <point>
    <parameters>
      <parameter name="knob1" value="3.4"/>
      <parameter name="knob2" value="100"/>
    </parameters>
    <system_metrics>
      <system_metric name="metric1" value="212.862" standard_dev="6.49"
          />
      <system_metric name="metric2" value="27.6" standard_dev="0.9"/>
    </system_metrics>
    <features>
      <feature name="feature1" value="100"/>
      <feature name="feature2" value="10" />
    </features>
  </point>
</points>
```

Figure 3.2: *Example of an XML configuration file which defines the application knowledge as an Operating Point list. This example shows a list with a single Operating Point.*

confidence is expressed as the number of times to consider its standard deviation. If the application is input-dependent, the value of the rank function r and the constraint functions ω_i also depend on the features of the input $\overline{f}$.

In this formulation, the main goal of *mARGOt* is to solve the optimization problem: finding the configuration $\hat{\overline{x}}$ that satisfies all the constraints C and maximizes (minimizes) the objective function r, given the current input i. The application must have a configuration to use even if it is not feasible to satisfy all the constraints. For this reason, *mARGOt* might relax constraints until a feasible solution is found, starting by relaxing the lowest priority constraint. Therefore, the end-user must sort the set of constraints by their priority. As shown in Figure 3.1, the *mARGOt* framework is composed of the application manager, the monitors' module, and the application knowledge. The following sections explain in details each component.

3.2 Application knowledge

For a generic application, the relation between software-knobs, EFPs of interest and input features is complex and unknown a priori. Therefore, we need a model of the application extra-functional behavior to solve the optimization problem stated in Eq. 3.1. *mARGOt* uses a list of *Operating Points* (OPs) as application knowledge, where each Operating Point θ states the target software-knob configuration and the achieved EFPs with the given input features; i.e. $\theta = \{x_1, \dots, x_n, f_1, \dots, f_n, m_1, \dots, m_n\}$. We choose this solution mainly for three reasons: (i) we are able to solve the optimisation problem by inspection efficiently, (ii) it guarantees that *mARGOt* will not choose an illegal configuration for the application, and (iii) it provides great management flexibility.

Figure 3.2 shows an example of application knowledge configuration file in XML, with a single Operating Point (lines 3-16). Let us suppose that the target application exposes two software-knobs (*knob1* and *knob2*), it is interested on two metrics (*metric1* and *metric2*) and it is able to extract two features from the current input (*feature1* and *feature2*). In this example, three sections compose the OP: the target software-knobs configuration (lines 4-7), the reached performance distribution (lines 8-11) and the related feature cluster (lines 12-15).

The OPs list is considered a required input. Therefore, *mARGOt* is agnostic on the methodology used to obtain the application knowledge. Even if the latter is considered an input, it is of paramount importance to *mARGOt* for solving the optimisation problem. Moreover, since the Design Space grows exponentially with the number of software-knobs, how to find set of software-knobs configurations that are Pareto-optimal, is a well-known problem in the literature, where several approaches are investigated [11–13]. In particular, the XML configuration file that describes the OPs list is compatible with the output generated by the Multicube Explorer [67]. Usually, this is a design time task since it requires the evaluation of several configurations, before obtaining the model. As alternative options, we provide to the application developer the possibility of learning the application knowledge at runtime, using a distributed approach. The latter will be described in details in Section 3.5.

3.3 Monitors module

This module provides to *mARGOt* the ability to observe the actual behaviour of either the application or the execution environment. This feature is critical for an autonomic manager because it provides feedback information, enabling the self-awareness ability [68]. The application knowledge defines the expected behaviour of the application. However, it might change according to the evolution of the system. For example, a power capper might reduce the frequency of the processor due to thermal reasons. In this case, we would expect that the application notices a degradation in its performance and it reacts, by using a different configuration to compensate. This adaptation is possible only if we have feedback information.

From the implementation point of view, *mARGOt* provides a suite of predefined monitors with broad applicability both at high- and low-level. Some examples of monitors implemented in *mARGOt* are:

Time Monitor. This monitor reads the time elapsed between a start point and a stopping point. It uses the std::chrono interface and might be configured to use different time units (e.g. *nsec*, *usec*, *msec*, *sec*).

Throughput Monitor. This monitor computes the throughput as the amount of elaborated data over the observed time interval. The metric is $data/second$. The time interval is measured as a difference between a start point and a stopping point as the time monitor while also reporting the throughput.

Memory Monitor. This monitor observes the resident set size of the virtual memory that the process is using. To gather the data, it parses the "/proc/self/statm" metafile. The unit of measure is the kilobytes, thus the monitor stores integer values.

System CPU Usage Monitor. This monitor computes the average utilisation of the processors at the system-level. The unit of measure is a percentage, and it is computed as the system busy time (both on the user and system level), over the considered time interval. To collect these data, the monitor parses the "/proc/stat" metafile. The OS updates the metafile values with a granularity of $msec$, but to get a significant measure, the interval of time should be greater than $50msec$.

Process CPU Usage Monitor. This monitor is similar to the System CPU Usage Monitor, but it computes the average utilisation of the processor by the application, defined as the time the application spent executing on the processors over the elapsed time. The std::chrono interface is used to compute the latter, while the *getrusage* function at OS level is used for the former. Even in this case, to get a significant measure the interval of time should be greater than $50msec$.

PAPI Monitor. It is used to observe low-level metrics by wrapping the widely adopted PAPI [69] framework. It enables an application to observe platform-related metrics, such as cache misses or instruction per cycles, transparently. The maximum number and type of observed metrics depend on the platform.

As stated in Chapter 2, approximate computing is a promising path to further improve computation efficiency, as shown in several works of literature. However, this approach requires to observe a metric related to the output quality, which typically is application-specific. For this reason, we implemented the monitors using a modular approach. In this way, application developers might implement a custom monitor for observing an application-specific metric easily. Since measuring quality metrics might

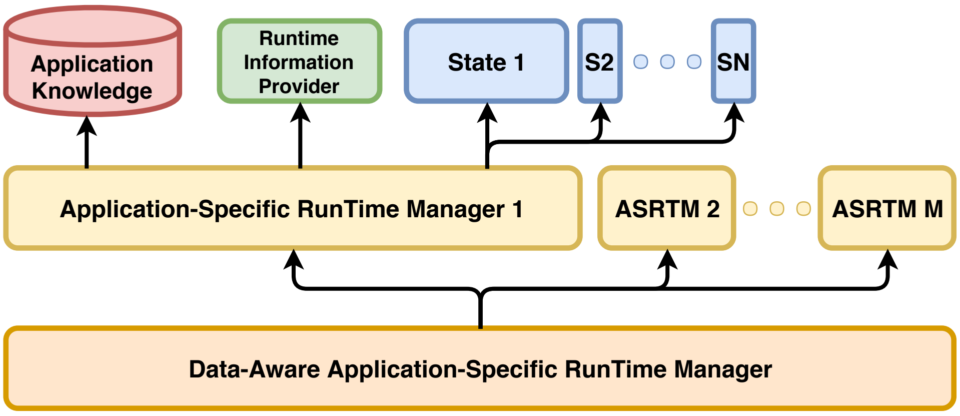

Application Knowledge
Runtime Information Provider
State 1
S2
SN
Application-Specific RunTime Manager 1
ASRTM 2
ASRTM M
Data-Aware Application-Specific RunTime Manager

Figure 3.3: *Overview of the Application Manager implemented in mARGOt, based on a hierarchical approach.*

be expensive, *mARGOt* does not require a continuous observation of a metric. The application developers choose if monitoring an EFP on each iteration, periodically or sporadically. Obviously, by decreasing the observations frequency, it delays the reactions of *mARGOt*. If it is not possible to monitor an EFP at runtime, *mARGOt* relies only on the expected behaviour, operating in an open-loop.

3.4 Application Manager

This component is the core of the *mARGOt* dynamic autotuner, which provides the self-optimization capability using a lightweight framework. From the methodology point of view, this component is in charge of solving the optimisation problem stated in Eq. 3.1: to find the software-knobs configuration $\hat{\bar{x}}$, while reacting to changes in the execution environment and adapting proactively according to input features.

From the implementation point of view, the application manager has a hierarchical structure, as shown in Figure 3.3, where each sub-component solves a specific problem. The *Data-Aware Application-Specific Run-Time Manager* (DA AS-RTM) provides a unified interface to application developers to set or change the application requirements, to set or change application knowledge and to retrieve the most suitable configuration $\hat{\bar{x}}$. Internally, the DA AS_RTM clusters the application knowledge according to input features $\bar{f}$, creating an *Application-Specific Run-Time Manager* (AS-RTM) for each cluster of Operating Points with the same input features. Therefore, the application knowledge implicitly defines the clusters

Algorithm 1: How the State component builds the internal representation of the optimization problem.

```
Data: Application knowledge $OP_{list}$, optimization function $r$, list of constraints $C$
Result: list of valid OPs $L\_valid$, lists of invalid OPs $L_{c_i}$
$L_{valid} = OP_{list}$ ;
for $c_i \in C$ (ascending priority order) do
    $L_{c_i} = \emptyset$;
    for $OP_j \in L_{valid}$ do
        if $OP_j$ does not satisfy $c_i$ then
            $L_{c_i} = Lc_i \cup OP_j$;
        end
    end
    $L_{valid} = L_{valid} \setminus L_{c_i}$;
    $L_{c_i} = sort(L_{c_i}, dist(OP_j, c_i))$;
end
$L_{valid} = sort(L_{valid}, r)$;
```

Algorithm 2: How the State element solves the optimization problem.

```
Data: list of valid OPs $L\_valid$, list of invalid OPs $L_{c_i}$, list of constraints $C$
Result: most suitable Operating Point $\overline{OP}$
if $L_{valid}! = \emptyset$ then
    return $L_{valid}[0]$;
else
    for $c_i \in C$ (descending priority order) do
        if $L_{c_i}! = \emptyset$ then
            return $L_{c_i}[0]$;
        end
    end
end
```

of Operating Points. Given the input features of the current input, the DA AS-RTM selects the cluster with features closer to the ones of the current input. It is possible to use a Euclidean distance between the two vectors, or a normalised one in case an element of the vector $\overline{f}$ is numerically different with respect to the others. Moreover, it is possible to express constraints on the selection of the cluster. For example, it is possible to enforce that the feature $f_i^{cluster}$ of the selected cluster must be lower (greater) or equal than the feature f_i^{inpt} of current input, i.e. $f_i^{cluster} \propto f_i^{inpt}$. Once the cluster for the current input is selected, the corresponding *Application-Specific Run-Time Manager* (AS-RTM) solves the optimisation problem relying on the following components.

The *State* element is in charge of solving the optimisation problem by using a differential approach. The initial optimisation problem does not have any constraints (i.e. $C = \emptyset$), and the objective function minimises the value of the first software-knob. From this initial state, the application might dynamically add constraints, define a different objective function or change the application knowledge. The solver can find the new optimal configuration efficiently, evaluating only the involved ones, by building an internal representation of the optimisation problem. Algorithm 1 shows the pseudo code for its initialisation. At first, it assumes that the application knowledge satisfies all the constraints. Therefore L_{valid} contains all the OPs. Then, for each constraint c_i, *mARGOt* iterates over the set of OPs in L_{valid} and it performs three operations. (1) It creates the list L_{c_i} which contains all the Operating Points invalidated by the constraint c_i. (2) Then it removes the OPs contained in the set L_{c_i} from the set L_{valid}, i.e. it removes from the set of valid OPs the ones that do not satisfy c_i. (3) Eventually, it sorts all the OPs in L_{c_i} according to their distance from satisfying the constraint c_i. After iterating over the constraints, *mARGOt* sort the list of valid OPs L_{valid} according to the objective function r. Using this representation, each time that *mARGOt* is invoked to solve the optimisation problem, it updates the internal structure and then it follows Algorithm 2. In particular, if the list L_{valid} is not empty, *mARGOt* returns the one that maximizes the rank function, i.e. $L_{valid}[0]$. Otherwise, *mARGOt* iterates over the constraints according to their priority, in reverse order, until it finds a constraint c_i with a non-empty L_{c_i}. Then the best OP is the closest to satisfy the constraint c_i, i.e. $L_{c_i}[0]$. If there is more than one OP at the same distance from c_i, *mARGOt* will narrow this set of the possible solutions using the constraints at the lower priority than c_i and the objective function r.

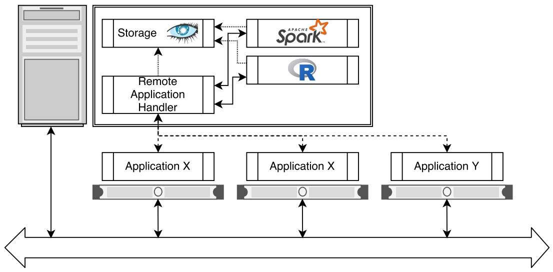
Given that the end-user might have different requirements according to different phases of the application, it is possible to define different states and switch among them at runtime. For example, in a video surveillance application, the end-user would like to perform a more accurate computation or a more energy-efficient one, according to the presence of an interesting scenario to analyse.

The *Runtime Information Provider* correlates an EFP of the application knowledge with an application monitor. In particular, it compares the observed behaviour with the expected one, and it computes a coefficient error defined as $e_{m_i} = \frac{expected_i}{observed_i}$, where e_{m_i} is the error coefficient for the i-th EFP. To avoid the zero trap, we add 1 to the numerator and denominator when $observed_i$ is equal to zero. Since it is impossible to observe the error coefficient also for other configurations (the application uses only one configuration each time), we assume that their error coefficients are equal to the observed one. This assumption implies that if we observe a performance degradation of 10% for the current configuration, we assume that also the other configurations will have a performance degradation of 10%. Therefore we scale the constraint value accordingly to react. For example, suppose that the end-user would like a throughput of at least $25fps$ and that we are using a configuration that has an expected throughput of $30fps$, but we observe a throughput of $15fps$. Then, the *Runtime Information Provider* will double the constraint value to compensate. The linear error propagation assumption might hold in several cases, providing a reaction mechanism in a seamless way for the developer. However, it does not apply to all cases. Typically, this happens when co-running applications share computational units. In this case, it is required to employ a more complex reaction mechanism, as described in details in Chapter 5.

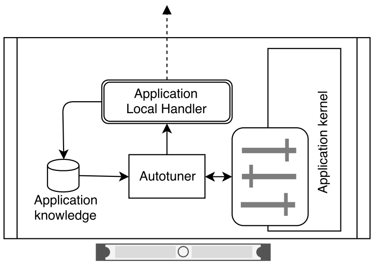
3.5 On-line Design Space Exploration

The *mARGOt* implementation let application developers define the application knowledge at runtime, enabling the possibility to learn it online, during the production phase. To achieve this goal, we propose an additional component that distributes the Design Space Exploration (DSE) among all the instances of an unknown application, integrated with *mARGOt*, at runtime. The benefits of this approach are the following: 1) it is possible to leverage all the available nodes to reduce the time-to-knowledge; 2) the application knowledge is tailored for the current input, and 3) we measure the EFPs with the production environment. From the methodology point of view, we employ two strategies to minimise the time required to obtain the application knowledge. On the one hand, we use design of experiment techniques (DoE) [70] to efficiently sample the design space and state-of-the-art modelling techniques to perform out-of-sample predictions. On the other hand, we employ an iterative exploration strategy to reduce as much as possible the required number of samples. In particular, the framework starts to explore a fraction of the design space and a learning plugin tries to obtain the application knowledge. If the derived EFPs models are not able to reach a target quality in the validation phase, the framework will resume the Design Space Exploration (DSE).

From the implementation point of view, Figure 3.4a shows the overall picture of the component, highlighting the two main actors: the *Remote Application Handler* and the running application instances. Each instance of the application has an *Application Local Handler* (client), as shown in Figure 3.4b, which interacts with the *Remote Application Handler* (server)



(a) *Global structure of the distributed DSE framework*



(b) *Structure of an application instance*

Figure 3.4: *The proposed approach to perform a distributed on-line Design Space Exploration, using a dedicated server outside of the computation node. We use MQTT protocol to perform extra-node communication.*

through MQTT or MQTTs protocols. The *Application Local Handler* is an asynchronous utility thread, that sends to the *Remote Application Handler* telemetry information and it manipulates the client application knowledge. In particular, during the learning phase, it will force the autotuner to select the software-knobs configuration to evaluate, while it sets the application knowledge once available. The *Remote Application Handler* is a worker thread-pool that interacts with clients to obtain the application knowledge, and it runs in a dedicated out-of-band node. The server stores information in a Cassandra database or CSV files, according to the execution context scale. Moreover, it uses a plugin system to model and to interpolate the relations between the EFPs, the software-knob configurations and the input features clusters, including also a wrapper interface for R and Spark.

Although the implementation of a plugin to derive a metric is straightforward, in the current implementation *mARGOt* provides three default plugins. The first one is rather simple, and it computes the mean value and standard deviation for each observed software-knob configuration. It can be used for a full-factorial Design Space Exploration, observing the whole Design Space, including the possible input features. The second plugin leverages a well-known approach [71] to interpolate application performance, implemented by the state-of-the-art R package [72]. The third plugin is a more complex *learning module* and it leverages model ensembles to boost the predictive capabilities of several base models. Section 3.5.2 provides more details of the plugin, while Section 3.5.3 describes the selection and validation algorithms.

The typical workflow of the framework when it interacts with an unknown application is as follows:

1. The clients notify themselves to the server.
2. The server asks one client information about the application, such as the number of software-knobs and their domain, the DoE technique or the number of observation for each software-knobs configuration.
3. Once the server has collected the information, it will call a model plugin to generate a set of configurations to explore.
4. The server dispatches to the available clients the configurations to evaluate in a round robin fashion.
5. Once the clients have explored all the configurations, the model plugin generates the application knowledge.
6. If the quality of the derived model is above the acceptance criteria, the server broadcasts the model to clients. Otherwise, it restarts from step 3, appending the new observations to the previous ones.

The framework implementation is resilient to crash of the server and the clients. Moreover, whenever a new client becomes available, it can join the design space exploration or receive the model directly. If application developers use a CASSANDRA database as back-end storage, it is also possible to use standard tools to visualise the extra-functional values (e.g. the execution traces of all the application instances running on the platform), or for query the application knowledge.

3.5.1 Design of Experiment

The approach proposed in this section aims at obtaining the application knowledge at the production phase. Therefore we want to reduce the design space exploration as much as possible. To reach this goal is essential to sample the design space to maximise the retrieved information. This problem is well-known in literature, where the different design of experiments (DoE) techniques are investigated [70], such as latin hypercube sampling or full-factorial. On top of them, the application developer might choose to leverage the Dmax algorithm [73] which maximises the determinant of the correlation ρ_{ij} defined as in Eq. 3.2,

$$\rho_{ij} = \begin{matrix} 1-\gamma & \text{if } h_{ij} \leq \varepsilon, \\ 0 & \text{if } h_{ij} > \varepsilon, \end{matrix} \tag{3.2}$$

where h is the distance between points x^i and x^j, ε is the threshold distance of the correlation between two points, and γ is a variogram.

This DoE technique exposes two free parameters: the total number of points to explore n and the threshold distance ε. We set n as $d{\cdot}m$, where d is the number of dimensions of the design space (i.e. the number of software-knobs) and m is the number of points to explore for each dimension. We provide to end-user the possibility to change the parameters m and ε from their default values (10 and 0.2 respectively). Moreover, the end-user might specify how many times explore each point in the DoE.

3.5.2 The learning module

This section describes in more details the modelling techniques used to learn the relation between EFPs and software-knobs by the *learning module*. In *mARGOt* context, the learning plugins model each EFP independently. Therefore, in our notation $\hat{y}$ represents the expected value of the target EFP, while x represents the vector of software-knobs and input features.

Linear models

The linear regression with n dependent variables and p explanatory variables is defined in Eq. 3.3,

$$\hat{y} = \alpha X \beta + \varepsilon, \tag{3.3}$$

where α is a constant, β is a vector of n parameters, X is a $n \times p$ matrix of explanatory terms, and ε is the vector of residuals or errors.

We use two flavours of linear models: in one case we consider only the model with a constant and the explanatory variables; in the second case we also consider two-way interactions of explanatory variables. The latter is created as the multiplication of pairs of explanatory variables.

MARS models

The second family of models used in the learning module is multivariate adaptive regression splines (MARS) [74]. This model iteratively adds basis functions to create the best possible representation of the variables interactions (nonparametric model). The MARS representation is defined in Eq. 3.4,

$$\hat{y} = c + \sum_{i=1}^{k} w_i B_i(x), \tag{3.4}$$

where c is a constant, k is number of basis functions, w_i is the constant coefficient of the basis function i, $B_i(x)$ is the basis function i. The basis function is of the form $\max(0, d_i - x)$, $\max(0, x - d_i)$ or the multiplication of multiple basis functions. The parameter d_i is a constant estimated by the model. In the *learning module* we also use a variation of this model, named POLYMARS, which enables a maximum of two-way interactions in the model [75].

Kriging model

The *learning module* uses an extension of the original Kriging model, named Universal Kriging (UK) [76], which assumes that observed values y comes from a deterministic process Y given by Eq. 3.5,

$$Y(x) = \mu(x) + Z(x) \tag{3.5}$$

where μ is trend defined by the number of basis functions, and Z is a known covariance kernel.

In the context of *mARGOt*, the generating process is seldom deterministic; therefore we need to relax this assumption. In particular, we forced the determinism by averaging the observed values for each observed software-knobs configuration.

Ensemble models

Model ensembling is a well-known approach to increase the predictive capabilities of base models by combining them, using different techniques. The *learning module* leverages two techniques based on cross-validation models: bagging [77] and stacking [78].

The bagging approach aims at decreasing the variance of the prediction. It focuses on a single base modelling technique, and it combines instances of the model trained with different data sub-samples. To perform prediction, it uses the mean of the predictions generated by the model instances. Given that we use average values for training the kriging model, the *learning module* is not allowed to leverage bagging ensembles, since the generated model will lead to an extremely biased validation.

The stacking approach aims at increasing the robustness of the prediction by combining base models. A stacked model should be able to decrease the weaknesses of the individual models and leverage their strengths. The *learning module* uses a weighted mean to combine the base models, finding the weights that lead the stacked model to best fit the observations vector. Moreover, the weights must be positive and sum up to one. We use the R package [79] to solve this problem of quadratic optimisation. As stated in previous work [78], this definition of the stacking significantly reduces the exploration space and makes the weights estimation robust.

3.5.3 Model validation and selection

This section describes how the *learning module*, described in Section 3.5.2, validates the available models and how it selects the best one. The typical approach for testing how the models fare in the prediction is to divide input data in a training and in a validation set. Given that we aim at reducing the number of observations to explore, we are not willing to holdout samples for the validation. Therefore, the *learning module* uses a k-fold validation scheme: the whole observations are divided into k-parts of equal size. We always use one part as a holdout set, and we use the rest of the observations to train the model. This data partitioning scheme implies that the *learning module* trains k models and each of them will have out-of-sample predictions on a different part of data. We will call these models *cross-validation models*.

To quantify the prediction quality of a model, we consider two metrics. A variant of the coefficient of determination (R^2) [80], and the mean absolute error, normalised by the observed values range (MAE_adj). In our case R^2 is computed as the square of the correlation between observed and predicted data. We choose this variant [80] because it can be used on the cross-validation and out of sample predictions to compare the results consistently. For evaluating these metrics for each base model, we consider the median of R^2 and MAE_adj across the *cross-validation models*. For model ensembles, we compute them considering the whole set of observations.

Once we evaluate all the models, we deem as eligible the ones that have R^2 higher than ϵ_r and MAE_adj less than ϵ_m, to enforce a minimum quality. Among the eligible models, we select the one that minimises the MAE_adj. If no model is eligible, the proposed approach will restart the design space exploration, up to a maximum number of iterations. When the *learning module* reaches the maximum number of iterations ($maxIt$), it concludes the exploration phase and does not perform out-of-sample predictions. The parameters ϵ_r, ϵ_m and $maxIt$ are exposed to end-user and by default, they are set to 0.5, 0.1 and -1 respectively.

3.6 Integration in the target application

In this section, we describe the effort required from end-users and application developers to integrate *mARGOt* in their application. In this context, end-users are the final users of the application, and therefore they are in charge of defining the application requirements and identifying input features (if any). Application developers are the ones that write the application source code; therefore they are in charge of identifying software-knobs and extracting features from the input (if any). From the implementation point of view, we designed the framework: (i) to apply the separation of concern approach between functional and extra-functional properties; (ii) to limit the code intrusiveness in terms of the number of lines of code to be changed and (iii) to propose an easy-to-use instrumentation of the code. Indeed, to ease the integration process in the target application, *mARGOt* provides a utility tool that starting from an XML description of the extra-functional concerns, it generates a high-level-interface tailored for the target application. The main configuration file describes the adaptation layer by stating:

1. the monitors of interest for the application;
2. the geometry of the problem, i.e. the EFPs of interest, the application software-knobs, and data features of the input;
3. the application requirements, i.e. the optimisation problem stated in Eq. 3.1. Optionally, the online DSE information.

If the application developers derive the application knowledge at design-time, the second configuration file states the list of Operating Points as shown in Figure 3.2.

Starting from this high-level description of the layer, the utility tool generates a library with the required glue code that aims at hiding, as much as possible, the *mARGOt* implementation details. In particular, the high-level interface exposes five functions to the developers:

- **init**. The global function that initializes the data structures.
- **update**. The block-level function that updates the application software-knobs with the most suitable configuration found.
- **start_monitor**. The block-level function that starts all the monitors of interest.
- **stop_monitor** The block-level function that stops all the monitors of interest.
- **log** The block-level function that logs the application behavior.

These functions hide the initialisation of the framework and its basic usage. For example, the update function takes as output parameters the software-knobs of the application and as input parameters the features of the current input. It uses the features to select the most suitable cluster, and then it sets software-knobs parameters according to the most suitable configuration found by *mARGOt*. However, if application developers need a more advanced adaptation strategy, for example, to change the application requirement at runtime, they need to use the *mARGOt* interface on top of the high-level one.

To show the integration effort, in the following example we focus on a toy application with two software-knobs (*knob1* and *knob2*) and two input features (*feature1* and *feature2*). The application algorithm is rather simple: it is composed of a loop that continuously elaborates new inputs. In this example, we suppose that the end-user is concerned about execution time and the computation error. In particular, he/she would like to minimise the computation error, provided an upper bound on the execution time.

In the context of this example, Figure 3.5 shows the main XML configuration file that states the extra-functional concerns. This file is composed of three sections: the monitor section (lines $4 - 21$), the application geometry section (lines $23 - 31$) and the adaptation section (lines $33 - 41$).

The monitor section lists all the monitors of interest for the user. In this example, we have an execution time monitor (lines $5 - 7$) and a custom monitor for observing the error (lines $8 - 21$). All the monitors might expose to application developers a statistical property over the observations, such as the average value in this example (line 6 and 20). If the end-user is not interested in observing the behaviour of the application, he/she might omit this section.

The application geometry section lists the application software-knobs (lines $24, 25$), the metrics of interest (lines $26, 27$) and the features of the input (lines $28 - 31$). In particular, it is possible to specify how to compute the distance between feature vectors (line 28) and to specify constraints on their selection, as described in Section 3.4. For example, if we consider *feature2* (line 30), we state that a cluster is eligible to be selected only if its *feature2* value is lower or equal than the *feature2* value of the current input. If we consider *feature1* (line 29) instead, we state that we do not impose any requirement on a cluster to be eligible. This mechanism provides to *mARGOt* a way to adapt proactively by sizing optimisation opportunities according to the actual input.

While the application geometry describes the boundaries of the problem, the adaptation section states the application requirements of the end-user. In particular, it states the application goals (line 34), the feedback information from the monitor (line 35) and the constrained multi-optimization problem (lines $36 - 41$). In the definition of a constraint (line 40), it is possible to specify a confidence interval and a priority. The confidence

```xml
<margot application="toy_app" version="v1">
  <block name="foo">

    <!-- MONITOR SECTION -->
    <monitor name="exec_time_monitor" type="Time">
      <expose var_name="avg_exec_time" what="average"/>
    </monitor>
    <monitor name="error_monitor" type="Custom">
      <spec>
        <header reference="margot/monitor.hpp"/>
        <class name="margot::Monitor&lt;float&gt;"/>
        <type name="float"/>
        <stop_method name="push"/>
      </spec>
      <stop>
        <param>
          <local_var name="error" type="float"/>
        </param>
      </stop>
      <expose var_name="avg_error" what="average"/>
    </monitor>

    <!-- APPLICATION GEOMETRY -->
    <knob name="k1" var_name="knob1" var_type="int"/>
    <knob name="k1" var_name="knob1" var_type="int"/>
    <metric name="exec_time" type="float" distribution="yes"/>
    <metric name="error" type="float" distribution="yes"/>
    <features distance="euclidean">
      <feature name="feature1" type="double" comparison="-"/>
      <feature name="feature2" type="double" comparison="LE"/>
    </features>

    <!-- ADAPTATION SECTION -->
    <goal name="exec_time_goal" metric_name="exec_time" cFun="LE"
        value="2"/>
    <adapt metric_name="exec_time" using="exec_time_monitor" inertia="
        3"/>
    <state name="normal" starting="yes">
      <minimize combination="simple">
        <metric name="error" coef="1.0"/>
      </minimize>
      <subject to="exec_time_goal" confidence="1" priority="10"/>
    </state>

  </block>
</margot>
```

Figure 3.5: *The main XML configuration file for the toy application, stating extra-functional concerns. In this example, we highlighted each section of the file.*

specifies how many times *mARGOt* shall take into account the standard deviation, to improve the resilience against noise with respect to the average behaviour. The priority is used to sort the constraints by their importance for the end-user. Application goals and feedback information provide to

```
 1  #include <margot.hpp>
 2
 3  int main()
 4  {
 5    margot::init();
 6
 7    int knob1 = 4;
 8    int knob2 = 2;
 9    float error = 0.0f;
10
11    while (work_to_do())
12    {
13      new_input = get_input();
14      const double feature1 = extract_feature1(new_input);
15      const double feature2 = extract_feature2(new_input);
16
17      MARGOT_MANAGED_BLOCK_FOO
18      {
19        do_job(new_input, knob1, knob2);
20        error = compute_error(new_input);
21      }
22    }
23  }
```

Figure 3.6: *Stripped C++ code of the target toy application, after the mARGOt integration. The omitted code is application logic and required include files.*

mARGOt the possibility to adapt reactively. Indeed, a violation of a goal in the optimisation problem or a discrepancy between the observed and expected behaviour of the application, triggers an adaptation form *mARGOt*, reacting to the event.

Starting from this configuration file, *mARGOt* automatically generates the glue code accordingly, exposing to application developers a high-level interface tailored for the specific problem. For a complete description of the XML syntax and semantics, please refer to the user manual in the *mARGOt* repository [14].

Figure 3.6 shows the source code of the application after the integration with *mARGOt*. To highlight the required effort, we hide the application algorithm in three functions: *work_to_do* (line 11) tests whether input data are available, *get_input* (line 13) retrieves the last input to elaborate and *do_job* (line 19) performs the elaboration. The integration effort requires application developers to include the *mARGOt* header (line 1), to initialize the framework (line 5) and to wrap the block of code managed by *mARGOt* (lines 17, 18, 21). Due to the structure of the code, it is possible to use a pre-processor macro to hide the five functions described earlier.

Even if we minimised the framework integration effort, we still need application developers to identify and to write the code that extracts meaningful features from an input (lines 14, 15); and a function that computes the elaboration error (line 20). Although these metrics are heavily application-dependent, a large percentage of works in literature analyse generic error metrics [3], and generic input features [59]. Application developers might consider these works as starting points to identify more customised metrics for their application. Moreover, the next chapters of this thesis evaluate the benefits of using the proposed framework in a wide range of scenarios and presenting examples of error metrics and input features as well.

3.7 Summary

The *mARGOt* framework provides a runtime self-optimization layer for adapting applications reactively and proactively. Differently, from static autotuner frameworks, *mARGOt* focuses on application-specific software knobs, whose optimal value depends on the system workload, on changes in the application requirements or on features of the actual input. In particular, *mARGOt* may change the software-knobs configuration if: 1) the application requirements changes, 2) the application knowledge changes, 3) the expected performance differs from the observed one, and 4) according to features of the current input. *mARGOt* has been designed to be lightweight and flexible to enable its deployment in a wide range of scenarios.

A key feature of *mARGOt* is how to derive the application knowledge. We offer to application developers two possibilities. First, they may leverage well-known techniques to run a DSE at design time. Second, we provide a software architecture to run the DSE directly at runtime, leveraging the *mARGOt* ability to change application knowledge.

From the implementation point of view, *mARGOt* minimises the integration effort by generating the required glue code automatically, starting from an XML description of extra-functional concerns, that hides implementation details. Moreover, the generated code exposes an easy-to-use interface for wrapping the managed region of code. Advanced adaptation rule might be expressed on top of this interface.

Furthermore, the *mARGOt* source code has been publicly released [14], along with user guides on the framework itself and on the tool that generates the high-level interface. Moreover, it is possible to derive a Doxygen documentation of the internal *mARGOt* implementation details to offer the possibility to customise or extend the framework.

CHAPTER 4

Experimental evaluation

This chapter aims at providing a brief experimental evaluation of the proposed framework, highlighting the benefits of each feature exposed by the adaptation layer. At first, we evaluate the overheads introduced by *mARGOt*. Then, we show the benefits of the reaction mechanisms and how considering input features might lead to identify and seize optimisation opportunities. Moreover, we show how learning at runtime the relation between software-knobs, extra-functional properties of interest and input features might be beneficial for the target application. Furthermore, for each feature we show a snippet of the target application source code, to better identify the required integration effort for the evaluated feature. Later chapters of this thesis provide a more detailed description of the framework exploitation in different scenarios and application case studies.

Given the flexibility of *mARGOt*, we deployed it on different platforms ranging from embedded to HPC. As a representative embedded platform, we used a Raspberry Pi (R) 3 model B. The board has a quad-core ARMv7 (R) (@ 1.2 Ghz) CPU with 1 GB of memory. To represent a typical HPC node, we used a platform composed of two Intel(R) Xeon(R) CPU E5-2630 v3 (@ 2.40GHz) with 128 GB of memory with dual channel configuration (@1866 MHz). All the experiments described in this chapter make use of the Intel platform, except the ones that evaluate the reaction mechanisms (Section 4.2) which uses the ARM platform. Moreover, for the experiments that aim at evaluating the online learning component, we use a platform with eight CPUs Intel(R) Xeon(R) X5482 @3.20GHz with 8 GB of memory.

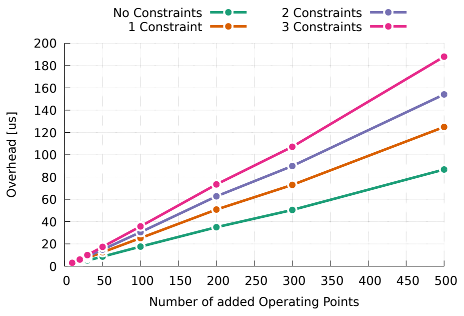
4.1 Evaluating the framework overheads

The proposed framework enables to instrument the code to introduce the adaptation layer as a standard C++ library that executes synchronously with the application. Therefore, we should consider the time spent by the *mARGOt* library to select a new configuration, to change the knowledge base, or to update the internal structures that represent application requirements as an overhead introduced to the target application.

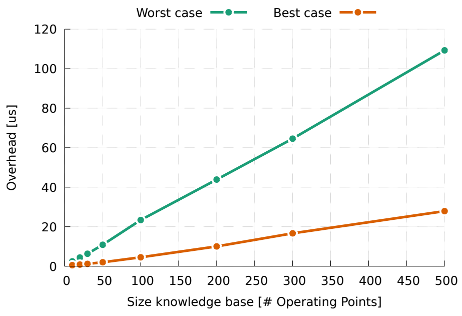
This experiment aims at evaluating the overheads introduced by *mARGOt* in the most significant operations exposed to application developers. Instead of providing a single value, in this experiment, we increase the problem complexity to show the trend of the overheads. Before discussing the results, it is important to remember that the *mARGOt* implementation follows a differential approach to solve the optimisation problem efficiently. It starts with a default base optimisation problem, where the application knowledge is empty, there are no constraints, and the optimisation function maximises the value of the first software-knob. From this initial state, every change issued from the application, such as adding Operating Points or defining a new objective function, affects only the Operating Points involved in the change. Even if the worst-case complexity of the algorithm is the same, it reduces the complexity of the average- and best-case scenarios.

To measure the overheads of the framework, we rely on a benchmark application that stresses the most demanding operations. Given that the execution time depends on the underlying architecture, this utility is included in the *mARGOt* repository. Therefore it is possible to measure the overheads on the target platform.

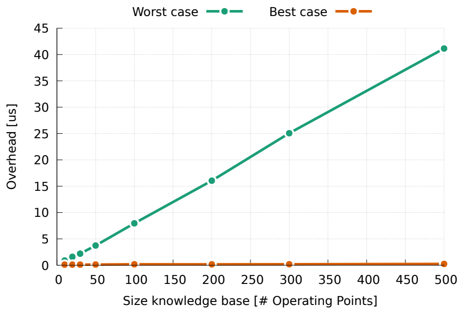
Figure 4.1 shows the introduced overheads by varying the size of the application knowledge or input feature clusters across the evaluated operations. In particular, Figure 4.1a shows the overhead for introducing Operating Points in the application knowledge by varying their number. Given that each constraint uses a dedicated “view” over the OPs, the introduced overhead also depends on their number. Figure 4.1b shows the overhead for introducing a new constraint in the optimisation problem. The overhead of this operation depends on how many OPs are admissible for the new



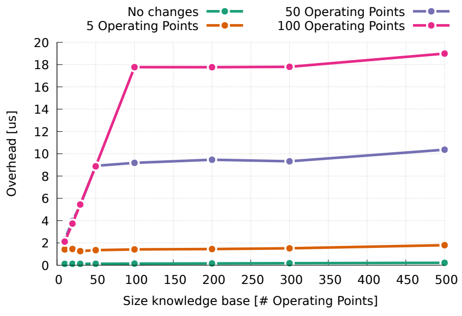
(a) *Add Operating Points*



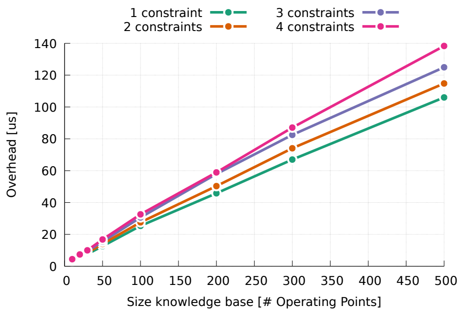
(b) *Add a constraint*



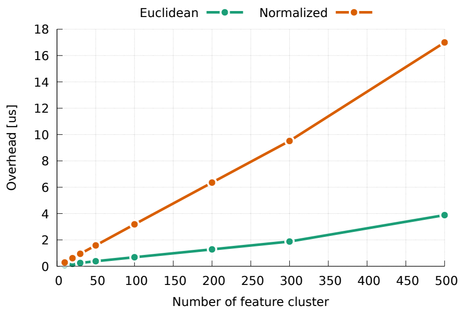
(c) *Define objective function*



(d) *Find best configuration (flat)*



(e) *Find best configuration (scaling)*



(f) *Select features cluster*

Figure 4.1: *Evaluation of the overheads introduced by mARGOt at runtime.*

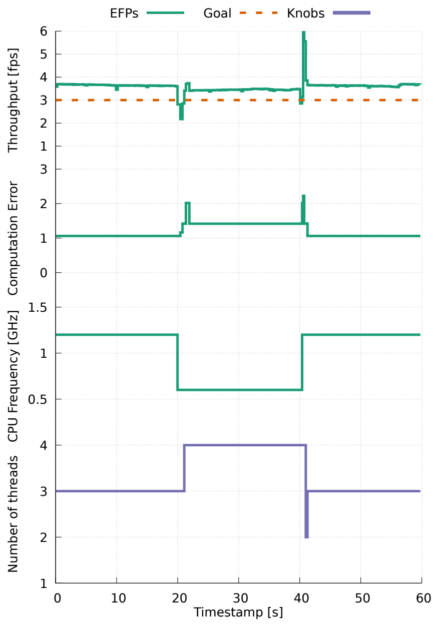
constraint. Even when no OPs are admissible, the introduced overhead is due to the building of a dedicated “view”, which involves all the OPs in the knowledge base. Figure 4.1c shows the overhead of defining a new objective function for the problem. In this case, the overhead depends on the number of OPs that satisfy all the constraints of the optimisation problem. Figure 4.1d and 4.1e show the overhead of solving the optimisation problem by inspection. While we might consider the previous operations as an initialization cost, we pay this overhead each time the application enters in the managed region of code. As shown in Figure 4.1d, the introduced overheads depend only on the number of OPs involved in the change with respect to the previous time that the optimisation problem was solved. Figure 4.1e shows the introduced overhead in the worst case scenario, which is not only because all the Operating Points are involved in the change, but it takes into consideration also the solver algorithm, by using a knowledge base to stress the implementation. Where all the OPs have the same value for the metrics related to the constraints and with the objective function. Figure 4.1f shows the overhead of selecting the closest feature cluster of the current input, where the feature vector is composed of three values. We pay this overhead each time the application enters in the managed region of code, and we shall add it to the overhead of solving the optimisation problem.

4.2 Evaluating the reaction mechanisms

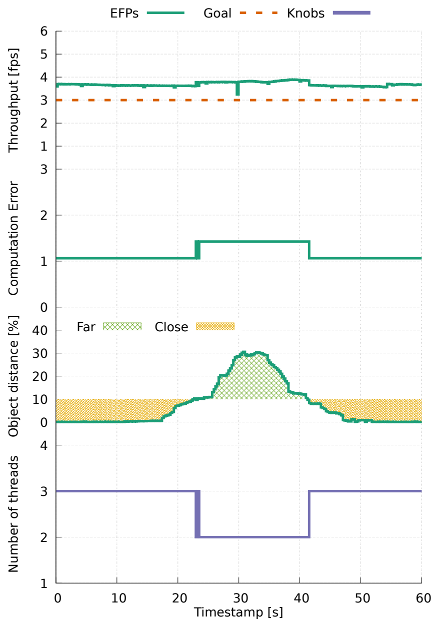
To demonstrate the benefits of reacting to changes in the application requirements or on the execution context, we target the *Stereomatching* application. It takes as input a pair of images from a stereo camera, and it computes a disparity map of the captured scene. The output of this application is required for estimating the depth of the objects in the scene. In this section, we consider a scenario where a smart camera is deployed either on a drone or a battery powered surveillance system.

The algorithm derived by [81] builds adaptive-shape support regions for each pixel of an image, based on colour similarity, and then it tries to match them on the other image, computing its disparity value. The algorithm implementation [4] exposes five application-specific knobs to modify the effort spent on building the support regions and on matching them in the second image to trade off the accuracy of the disparity image (the *Stereomatching* output) and the execution time (and thus the reachable application throughput). The accuracy metric is the *disparity error*, defined as the average intensity difference per pixel, in percentage, between the computed output and the reference output. The application has been parallelised by using OpenMP, making available as sixth software-knob the number of threads used for the computation.

The end-user does not require the application to sustain the throughput of the input video stream, but he requires that the application must reach a minimum throughput for detecting the position and depth of the objects in the scene. In this scenario, we set this high priority constraint to $3fps$. On top of this constraint, we envisioned two different application requirements



(a) *Reaction to a throughput degradation*



(b) *Reaction to a change in the requirements*

Figure 4.2: *Execution trace of Stereomatching running in an embedded platform. The x-axis shows the timestamp of the experiment, while the y-axes show extra-functional properties of the system or the value of target software-knob.*

according to the scene observed from the stereo camera. First, if in the previous scene there is no object close to the camera, the objective function minimises the disparity error with an additional low-priority constraint for executing the application by using a single software thread. Second, if there are objects close to the camera, the objective function minimises the geometric mean between the disparity error and the number of software threads, without any other constraint except the one on the throughput. The philosophy behind these two *states* is that in the first one we try to execute in a "low-power" mode because nothing is interesting in the scene, while in the second *state* we focus on the output quality without forgetting that the smart camera is placed on a battery-powered device.

To demonstrate the adaptivity added to the *Stereomatching* application, we focused on two different use cases as shown in Figure 4.2a and 4.2b. The first use case (Figure 4.2a) shows how the feedback information from the monitors trigger the adaptation, reacting to a change in the application performance. The second use case (Figure 4.2b) shows the benefits of reacting to changes in the application requirements (such as switching from

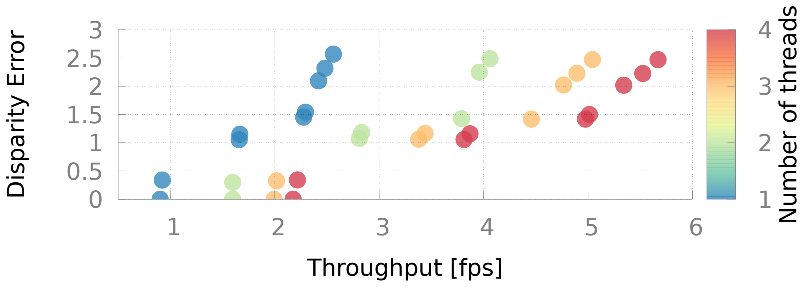


Figure 4.3: *Application knowledge of the Stereomatching application. Each circle represents an Operating Point. The x-axis represents the expected average throughput, the y-axis the expected average error, and the color range the parallelism level.*

one *state* to the other) according to the system evolution. Figure 4.2 shows the results of these experiments, while Figure 4.3 reports the application knowledge (i.e. the Pareto-optimal Operating Points). For clarity reasons, in Figure 4.2 we omitted the software-knobs that are not relevant to the experiment.

In the first use case (Figure 4.2a), we execute *Stereomatching* for $60s$. After $20s$, we reduce the frequencies of the platform cores by using the CPUfreq framework, for example, to simulate the effect of a power capping due to thermal reasons, and then we restore the original frequency of the cores after $20s$. We executed the entire experiment under the assumption that there is an object close to the camera. Figure 4.2a shows the execution trace of this experiment in terms of CPU frequency, number of threads, computation error and throughput.

At the beginning of the experiment, *mARGOt* selects among the configurations that satisfy the constraint on the throughput, the one that minimises the error and resource usage. When we reduce the frequency of the cores, the throughput monitor observes a degradation on the performance with respect to the expected one, triggering the adaptation. In particular, *mARGOt* chooses among the valid configurations, the one that minimises the objective function, while providing the requested throughput adjusted by the measured degradation. When we restore the original frequency, the throughput monitor observes a performance improvement and triggers the second adaptation. Given that we restored the original condition, the selected configuration is the same as the initial one.

In the second use case (Figure 4.2b), we processed a video stream captured from the stereo camera while it slowly moves from one close object (from 0s to around 20s) to another one (around 40s to 60s). During the transition between the two objects, there is a period (around 20s to 40s) where there is no object close to the camera. Figure 4.2b shows the execution trace of this experiment in terms of measured object distance, number of threads, computation error and throughput.

At the beginning and at the end of the experiment, when there is an object close to the camera, the configuration selected by *mARGOt* is the same used to start the previous experiment (the conditions are the same). However, when at time $22s$ there are no more objects close to the camera, *mARGOt* switches to a more power safe state, which introduces the constraint on a single thread execution. From the knowledge base (see Figure 4.3), we notice that on this platform there is no configuration reaching a throughput of $3fps$ by using a single thread. For this reason, *mARGOt* automatically relaxes the lower priority constraint, selecting the configuration which is closest to satisfy it, i.e. using two threads. Among the software-knob configurations that use two threads, *mARGOt* selects the one that minimises the objective function.

4.2.1 Integration effort

While the previous section demonstrated the benefits of the reaction mechanisms provided by the adaptation layer, this section aims at showing the integration effort in the target application to achieve the evaluated behaviour. Figure 4.4 reports the source code of the *Stereomatching* application. To highlight the integration effort, we omitted application-specific code, but we kept source code structure. The application is composed by the main loop that continuously elaborates incoming images, for the requested stream duration (lines 19-51). The *mARGOt* integration is similar to the one described in Chapter 3.6: we include the required header (line 2), we initialize the framework (line 14), and we wrap the managed region of code (lines 21-30 and lines 34-43). For this application, the error measurement involves a computation of the output image using a reference configuration (lines 40-42), besides the actual computation (line 32). Therefore, observing this metric at the production phase will kill the benefits of the accuracy-throughput tradeoffs, because for evaluating the error it requires the computation of the output using the reference software-knobs configuration. For this reason, the *start_monitor* and *stop_monitor* functions do not handle the error monitor, but we manage it manually. The idea is to measure the error metric only to obtain the application knowledge at design time; i.e. only when the application knowledge is not empty (line 38). The reaction to changes in the execution environment, as in the first use case,

```cpp
// omitted application includes
#include <margot.hpp>

// default software-knobs configuration
int max_hypo_value = 100;
int hypo_step = 1;
int max_arm_length = 18;
int color_threshold = 26;
int matchcost_limit = 60;
int num_threads = 4;

int main()
{
  margot::init();

  // omitted initialization code

  // main loop of the application
  while(std::chrono::steady_clock::now() < stop_time)
  {
    // update the parameters
    if (margot::disparity::update( max_hypo_value, hypo_step,
                                   max_arm_length, color_threshold,
                                   matchcost_limit, num_threads ))
    {
      margot::disparity::manager.configuration_applied();
    }

    // start the monitors
    margot::disparity::start_monitor();

    // omitted application code

    // stop the monitors
    margot::disparity::stop_monitor();

    // stop the monitors (cpu and throughput) and compute error
    if (margot::disparity::manager.is_application_knowledge_empty())
    {
      cv::Mat ref_img = do_job(left_image, right_image, ref_conf);
      const auto error = compute_error(output_img, ref_img);
      margot::disparity::monitor::error_monitor.push(error);
    }

    // rule to change the state
    const float proximity = compute_closeness(output_img);
    if ( proximity <= 10.0f + epsilon )
      margot::disparity::manager.change_active_state("close");
    else
      margot::disparity::manager.change_active_state("far");
  }
}
```

Figure 4.4: *Stripped C++ code of the Stereomatching application, highlighting the integration effort required to achieve the desired behavior.*

does not require additional integration effort. However, we need to add an "if-then-else" rule (lines 47-50) to change the *mARGOt state* according to the observed scene from the camera (line 46).

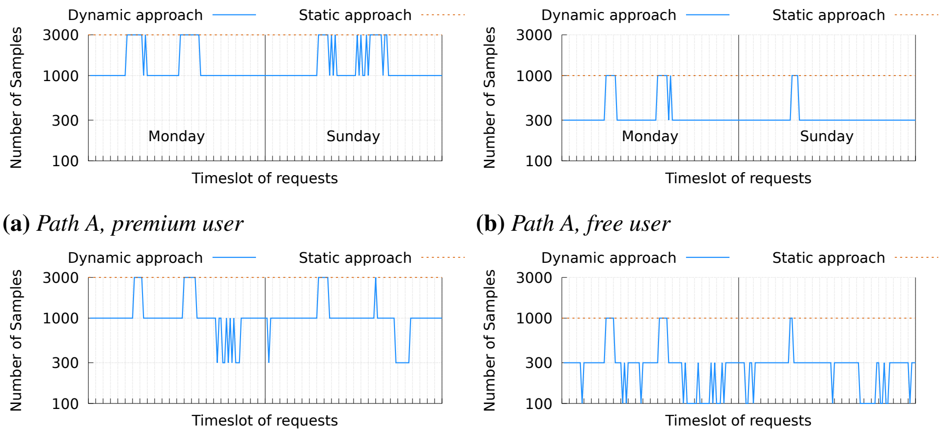
4.3 Evaluating the proactive adaptation

To demonstrate how the adaptation layer provided by *mARGOt* can identify and seize optimisation opportunities at production phase by using features of the current input, we target an application that uses Monte Carlo simulations to estimate the travel time distribution in a processing pipeline for a car navigation system. In particular, the *Probabilistic Time-Dependent Routing* (*PTDR*) algorithm [82] is a crucial component of a cooperative routing task computing the estimated travel time distribution. Then, later stages of the navigation system leverage this information to select the best solution among different routes.

To generate this output, *PTDR* must first estimate the travel time distribution and then extract statistical properties to be forwarded to the later stages of the navigation system. Each trial of the Monte Carlo simulates an independent route traversal over an annotated graph in terms of speed profiles. Given a sufficient number of trials, the sampled distribution of travel times will asymptotically converge towards the real distribution. The application derives the statistical property of interest using this distribution (such as the average or the 3rd quantile), which represents the actual output of the application.

The application is designed and already optimised to leverage the resources of the target HPC platform [83] and exposes as software-knob, the number of Monte Carlo samples to be used to compute the output. We defined the error metric as the difference between the value extracted with a limited number of samples and the one extracted with a very large (theoretically infinite) number of samples (we used 1M samples). Moreover, as defined in [83], we can differentiate among paths with a broad or narrow distribution of speed profiles, resulting respectively less or more predictable regarding travel time estimation. We call this feature, that we can extract easily before running the *PTDR*, *unpredictability*. We provide this data-feature to *mARGOt* for selecting the most suitable software-knob configuration for each simulation. Chapter 7 describes in more details the application and the effects of the *unpredictability* on the results quality.

Concerning application requirements, the end-user would like to minimise the number of samples used to compute the output, with a limit on the error upper bound. In this use case, we want to demonstrate how it



(a) *Path A, premium user*

(b) *Path A, free user*

(c) *Path B, premium user*

(d) *Path B, free user*

Figure 4.5: *Number of samples used by the adaptive PTDR application by changing the starting time of the request.*

is possible to use *mARGOt* to further increase the application efficiency, by adapting the number of trials proactively according to the road and in particular to its unpredictability. Without dynamic adaptation, the end-user should find the minimum number of samples that leads to a satisfying computation error for the worst case scenario. Moreover, the end-user would like to differentiate the threshold on the computation error constraint, according to whether a *premium user* (error $< 3\%$) or a *free user* (error $< 6\%$) generates the request.

Before running the application, we performed an experimental campaign by using random requests from 300 paths in the Czech Republic [82], in different moments of the week, to build the application knowledge. Moreover, we limited the software-knob values to $[100, 300, 1000, 3000]$ according to the previous analysis of the application [82]. Furthermore, to increase the robustness of the approach, we consider three times the standard deviation of a software-knob configuration for the constraint on the computation error.

Figure 4.5 shows the selected number of samples in an experiment that generates four types of requests every $15min$ on two days of the week, Monday and Sunday. In particular, for each type of user, we consider two different paths. Figures 4.5c and 4.5a shows the results for the premium user, while Figures 4.5d and 4.5b shows the results for the free user. On the one hand, this experiment shows how changing the application requirements (premium and free users) decreases the number of samples used to

```cpp
float ptdr(const Routing::MCSimulation& route)
{
  // default software-knobs value
  int num_samples = 10000;
  float unpredictability = 0;

  // declare the minimum amount of samples
  int min_samples = 100;

  // run with lowest number of samples
  std::vector<float> travel_times = route.Simulate(min_samples);

  // extract input feature
  const float f = extract_feature(travel_times);

  // update the application knobs, if neeeded
  if (margot::travel::update(num_samples, unpredictability))
  {
    margot::travel::manager.configuration_applied();
  }

  // check if we need to perform
  // additional simulations
  if (num_samples > min_samples)
  {
    route.Simulate(travel_times, num_samples - min_samples);
  }

  // return the results
  return compute_output(travel_times);
}
```

Figure 4.6: *Stripped C++ code of the PTDR application, highlighting the integration effort required to adapt in a proactive fashion.*

satisfy the request, considering both static and dynamic approaches. On the other hand, this experiment shows how input features (dynamic approach) decreases the number of samples with respect to using a single conservative configuration (static approach). This approach is feasible since different paths have different characteristics, defined by their *unpredictability*. For example, countryside requests are more predictable than those coming from an urban area. In this experiment, the proposed approach easily implemented by using *mARGOt* uses approximately the 30% of the number of samples with respect to a static approach, with an overhead comparable to compute 2 samples only. As previously mentioned, Chapter 7 describes in more details the relationship between the input features and the quality of the results. In particular, it will demonstrate through an extensive experimental campaign the validity of the approach.

4.3.1 Integration effort

From the integration point of view, leveraging input features to adapt proactively implies providing additional parameters to the generated *update* function. Figure 4.6 shows the computation kernel of the *PTDR* application, used at production phase. In particular, it takes as an input parameter the Monte Carlo simulator for the given route (line 1), and it provides as output the desired statistical property (line 30). In the first phase of the execution, it performs the route traversal simulations with the lowest number of samples (line 11). With this initial population, the application computes the input feature (line 14). Then *mARGOt* selects the most suitable number of samples according to the input feature (lines 17-20). Given that in this application we tuned a single software-knob and given the high-level policies from end-user, we are not interested in observing the actual behaviour of the application. If the selected number of samples is greater than the minimum number, *mARGOt* simulates further route traversals to satisfy the constraint on the accuracy (lines 24-27).

4.4 Evaluating the online learning module

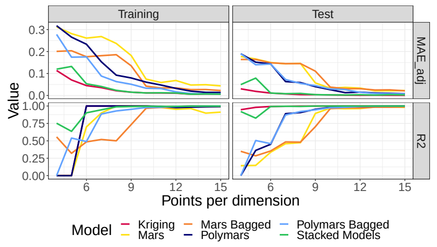
This section aims at experimentally assessing the ability of *mARGOt* to learn the application knowledge at runtime. In particular, Section 4.4.1 evaluates the *learning module*, using synthetic applications with a known relation between EFPs and software-knobs. Section 4.4.2 focuses on a geometrical docking application to evaluate the benefits provided by the proposed framework for the end-user in a real-world case study.

4.4.1 Model validation

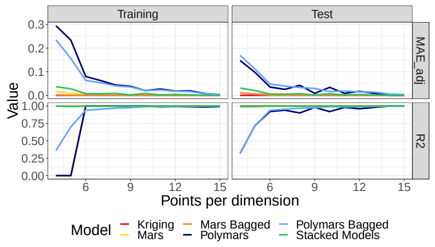
This experiment aims at evaluating the ability of the *learning module* to perform out-of-sample predictions, while we trained it with a fraction of the design space. In particular, we created two synthetic applications, with a known relation between the software-knobs and the EFPs.

The first application is based on a test function derived from the work of Binh [84], and the problem is defined in Eq. 4.1:

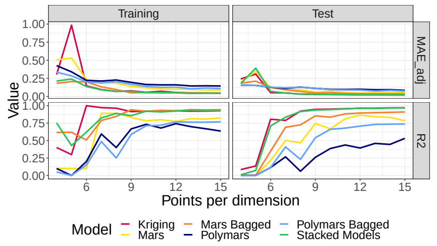
$$\begin{aligned} f_1(x,y) &= x^2 - y \\ f_2(x,y) &= -0.5x - y - 1 \\ \text{where} & \quad -7 \leq x, y \leq 4. \end{aligned} \tag{4.1}$$



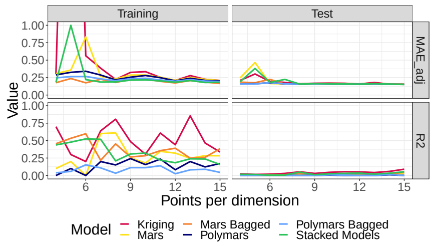
(a) *Binh* f_1



(b) *Binh* f_2



(c) *Kursawe* f_1



(d) *Kursawe* f_2

Figure 4.7: *R^2 and MAE_adj for the Binh and Kursawe synthetic applications, using different models, by changing the number of points per dimension.*

The second application is based on the test function introduced in Kursawe [85], and it is defined in Eq. 4.2:

$$\begin{aligned} f_1(x) &= \sum_{i=1}^{2} \left[-10 \exp\left(-0.2\sqrt(x_i^2 + x_{i+1}^2)\right)\right] \\ f_2(x) &= \sum_{i=1}^{3} \left[|x_i|^{0.8} + 5\sin(x_i^3)\right] \\ \text{where} & \quad -5 \le x_i \le 5. \end{aligned} \tag{4.2}$$

These functions were chosen to test the whole range of different function types such as linear, nonlinear and exponential. The Binh functions have two different variables and therefore two software-knobs, while the Kursawe functions have three different variables.

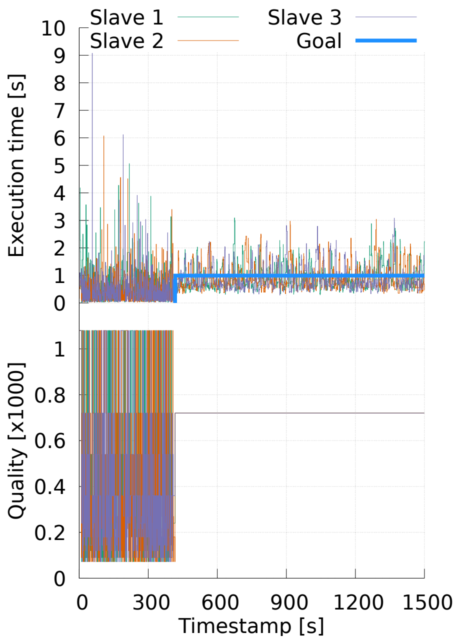
Figure 4.7 shows the results of this experiment. In particular, Figure 4.7a and 4.7b refer to the Binh application, while Figure 4.7c and 4.7d refer to the Kursawe application. For each function, we show the reached R^2 and MAE_adj for nonlinear models, for training and test, by varying the number of explored points per dimension. While they usually are included in the model selection phase, we excluded the linear models from the plot for graphical reasons. Indeed, 3 out of 4 studied functions are nonlinear, and linear models had poor results in comparison to other models. On the other hand, for the Bihn f_2 linear models had a perfect fit for all the tested points per dimension configurations.

From experimental results, we might notice a trend for the number of points per dimension. In cases of a low number of explored points per dimension, the R^2 and MAE_adj values of the models are spread over a considerable interval, where the *learning module* deems as best different model families. On the opposite, while increasing the number of points per dimension, it is possible to notice that the *learning module* converges to accurate models when considering the Binh functions (f_1 and f_2) and the Kursawe f_1. After 10 points per dimension, it is possible to notice how the R^2 and MAE_adj stabilises and how the training values are comparable to the test values. If we focus on the f_2 of the Kursawe application, we might notice how the *learning module* is not able to generate a good model to be used in prediction. This result is due to the complexity of the relation between f_2 and software-knobs in the synthetic application. If we consider the model selection, there are three models which are dominant: Kriging, MARS bagged and stacked models. We can see this trend in both synthetic applications. The actual model selected strongly depends on the predicted function and on the values of the model selection parameters (i.e. ϵ_r and ϵ_m).

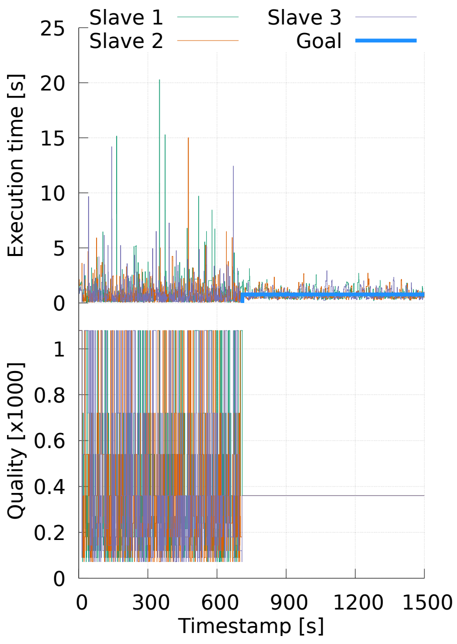
4.4.2 Molecular docking case study

Among the tasks that are involved in a drug discovery process, molecular docking is one of the earliest, and it is performed *in silico*. Usually, it is used to virtual screen a huge library of molecules, named *ligands*, to find the ones with the strongest interaction with the binding site of a second molecule, named *pocket*, to forward to later stages of the drug discovery process [86]. The complexity of this task is not only due to the considerable number of ligands to evaluate but also to the number of degree of freedom involved in the evaluation of a ligand-pocket interaction. In particular, it is possible to alter the shape of the molecule, without altering its chemical properties, by rotating a subset of bonds between the atoms of a ligand, named *rotamers*.

In this experiment, we focus on a geometric docking kernel, part of the LiGen Dock application [87], named *GeoDock*. Due to the complexity of evaluating the chemical interaction of a pocket-ligand pair, this kernel considers only geometrical information, and it is used to filter out the ligands that are unable to fit in the target pocket. The application exposes two software-knobs that generate quality-throughput tradeoffs by reducing



(a) *Pocket 1b9v*



(b) *Pocket 1cvu*

Figure 4.8: *Execution trace of the docking application learning phase, using three slaves. For each pocket, we show the execution time to compute the current ligand and the reached quality. We omitted the master trace because it does not perform any computation.*

the number of alternative poses evaluated for each rotamer of the ligand. The end-users are typically pharmaceutical companies that rent resources of an HPC platform, to evaluate a chemical library in a typical batch job. Therefore, they are interested in time-to-solution and on the quality of the elaboration, defined as the number of evaluated poses. Chapter 8 describes in details the application domain, stating the identified software-knobs and providing more insight into the application behaviour.

Given that the later stages of the drug discovery process include a monetary effort to perform tests *in-vitro* and *in-vivo*, the reproducibility of the experiment is a domain requirement. Therefore, once we have obtained the application knowledge, we restart to evaluate the chemical library with the configuration that maximises the quality, given a constraint on the time-to-solution that includes the time spent on the learning phase. In the following experiments, we used a library of $113k$ ligands, where each ligand has a number of atoms between 28 and 153 and a number of rotamers between 2 and 53. Moreover, we use six pockets (*1b9v*, *1c1b*, *1cvu*, *1cx2*, *1dh3* and

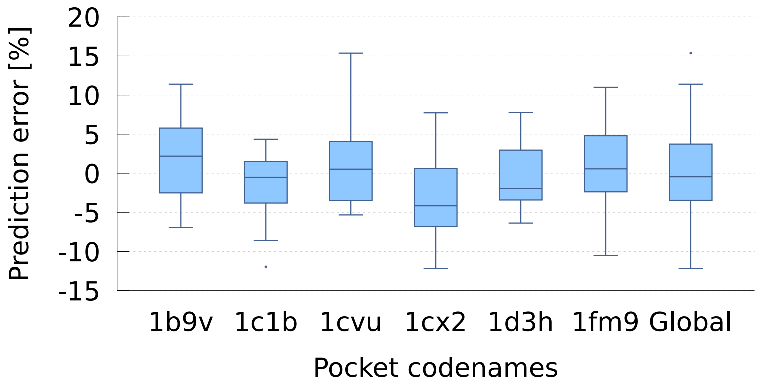


Figure 4.9: *Distribution of the prediction error in percentage, grouped by different target pockets. Negative values indicate an underestimation of time-to-solution.*

1fm9) from the RCSB Protein Databank (PDB).

Figure 4.8 shows an execution trace of the docking application using a "small" (Figure 4.8b) and a "big" pocket (Figure 4.8a), in terms of execution time for evaluating the pocket-ligand pair and quality of the results. The length of the learning phases in the two cases depends on the different model convergence time and on the input characteristic. After the learning phase, the application set the goal value as the average time required to elaborate a ligand multiplied by the number of ligands to elaborate. It is possible to notice how after the initial exploration of the design space, the application settles with a software-knobs configuration that leads to similar execution time, but with different quality according to the current input.

To validate the approach, we performed an experimental campaign using a library of $4k$ ligands, randomly sampled from the $113k$ ligands, targeting six different pockets. In particular, for each pocket we repeated the experiment ten times, reporting the normalised difference between the expected time-to-solution and the observed one (see Figure 4.9). For the learning phase, we observed each software-knobs configuration in the DoE with 200 different ligands. Figure 4.9 shows that a large fraction of the time-to-solution errors are within 5%. In this case, the proposed approach can accurately estimate the time-to-solution for the current inputs, maximising the quality of the results given the time budget. In this experiment, we found that Kriging, MARS bagged, and stacked models are the top three models according to MAE_adj, the actual selected model varied across the experiments.

4.4.3 Integration effort

```cpp
void geodock_kernel( /* omitted parameters */ )
{
  // parse the received ligand and build data structures

  // set the goal as received from master
  margot::docking::goal::exec_time_limit.set(target_exec_time);

  // update step for retrieving the new configuration
  if (margot::docking::update(high_precision_step,
      number_of_repetitions))
  {
    margot::docking::manager.configuration_applied();
  }

  // start the monitors
  margot::docking::start_monitor(  );

  // omitted kernel code

  // stop the monitors
  margot::docking::stop_monitor( quality );
}

void master_task( /* omitted parameters */ )
{
  bool i_was_in_dse = true;
  float min_exec_time = -1.0f;

  while( /* there are ligands in db*/ )
  {
    // check if we need to restart the elaboration
    if ((!margot::docking::manager.in_design_space_exploration()) &&
        (i_was_in_dse))
    {
      // compute execution time goal value

      // reset the database

      i_was_in_dse = false;
    }

    // send ligand and goal value
  }
}
```

Figure 4.10: *Stripped C++ code of the GeoDock application, highlighting the integration effort required to learn at runtime the application knowledge.*

From the integration point of view, the generated high-level interface handle the online Design Space Exploration automatically. Given the embarrassingly parallel nature of the *GeoDock* application, the implementation follows a master/slave pattern using the MPI framework, where a master task dispatch work to any available slave. Figure 4.10 shows the source code of the *GeoDock* application kernel (lines 1-21) and of the master task (lines 23-42). The adaptation layer is defined inside the kernel using the high-level interface: the update step retrieves the software-knobs configuration (lines 9-12) and the start/stop methods instrument the execution (lines 15 and 20). Due to the determinism requirement, the observed values are not used to react to changes in the execution environment, but to evaluate the software-knobs configurations for learning the application knowledge. Moreover, before the update step, the application kernel set the value on the execution time goal (line 6) according to a value sent by the master task.

The master task purpose is to dispatch the next ligand in the chemical library to the first available slave. Due to the determinism requirement, once we have collected the application knowledge and computed the goal value (lines 31-38), the master task must restart to evaluate the database. Moreover, to have a unique constraint value for all the slaves without introducing a global synchronisation point, each time the master sends the next ligand to be evaluated, it shall also send the goal value.

4.5 Summary

This chapter assessed the proposed dynamic autotuning framework by evaluating the benefits of the features exposed by the adaptation layer and by evaluating the introduced overheads. From the experiment results, it is possible to notice how the enhanced application can leverage the tradeoffs between the extra-functional properties of interest to reach high-level goals. Moreover, the enhanced application might be able to identify and seize optimisation opportunities at runtime by using input features or by learning the application knowledge at runtime for the actual input.

For each feature of the adaptation layer, this chapter described the integration effort by focusing on the related snipped of source code and highlighting the introduced changes. In the considered application, we may notice how the intrusiveness of the approach is limited. Although the high-level interface hides the *mARGOt* internal implementation details, application developers are still in charge of computing the most suitable error metric and of extracting the input features (if available).

Part II

Framework exploitation

CHAPTER 5

Evaluating Orthogonality between Application Autotuning and Resource Management

This chapter describes a scenario where we use the *mARGOt* autotuning framework in the context of co-running applications that share computation resources. Using a video processing application, we compare different techniques of *Run-Time Resource Management* (RTRM) to evaluate how much the interaction between RTRM and application autotuning can become synergistic yet orthogonal. Moreover, we propose a light-weight RTRM technique, based on *mARGOt* and its ability to sense the execution environment.

5.1 Introduction

A new trend in programming data-parallel computationally intensive applications is using OpenCL not only for programming heterogeneous platforms (by exploiting GPU or FPGA accelerators) but also for homogeneous platforms. OpenCL follows an "offload" programming model, where an accelerator is accessed via the host system and is programmed as a co-processor, to speed up the execution of computationally intensive kernels. OpenCL API [88] is designed to make efficient use of the massive computational parallelism provided by modern accelerators. On the contrary, there is not yet support for efficient deployment of multiple OpenCL applications on the same platform. However, the increasing number of processing units integrated on the same chip delivers computational capabilities that can exceed the performance requirements of a single application. Thus, server consolidation is a common approach to reduce the number of machines (and therefore the power consumption) needed to provide some services. In this area, runtime adaptability represents a key technique for computing systems to adjust their behaviour with respect to operating environments, usage contexts, resource availability and even to faults, thus enabling close-to-optimal operation in the face of changing conditions.

In this chapter, we address the problem of resource sharing in server consolidation for adaptive and computationally intensive OpenCL applications. We characterise our target application scenario by unpredictable, variable workloads, where applications have to serve concurrent requests and provide a best-effort service to the users. In this context, we are interested in evaluating the combination of *i)* application autotuning and *ii)* Run-Time Resource Management (RTRM) techniques to improve resource sharing among computationally intensive workloads. The lack of autotuning and runtime adaptation capabilities at the application-level leads to sub-optimal power/performance trade-offs at the system level given by the underutilization of system resources. On one side, the autotuning mechanism allows to trade off performance and Quality-of-Result metrics directly acting on application-level knobs; on the other side, runtime system management needs to optimise the computing capabilities concerning dynamic variations of the environmental conditions, computational demands, and resource availability.

In this context, *mARGOt* tunes the application behaviour according to available resources and given end-user requirements. Among the application knobs, we have included a parameter which controls, on a multi-core platform, the CPU quota used by the application. To this end, we exploit the *device fission* OpenCL API [88] to deploy OpenCL kernels on selected processing units of a multi-core CPU. The drawback of this API is that OpenCL dynamic compilation is required each time we reconfigure the computing device. Thus, our analysis also takes into account the benefits of asynchronous compilation to reduce the reconfiguration overhead. Additionally, this chapter introduces an innovative light-weight technique – called *resource-aware Application-Specific Run-Time Manager* (AS-RTM)

– where *mARGOt* is enabled to take autonomous decisions on resource utilization. The *resource-aware AS-RTM* considers the information of system workload gathered by platform sensing for taking reconfiguration decisions while minimising the impact on other applications that share the same resources. Therefore, the main difference from previous approaches (e.g. "invade and retreat" [89]) is that applications act as autonomous agents, without coordination among them. On the one hand, this solution has the advantage of being non-intrusive from a design point of view, since it does not require a communication infrastructure; on the other hand, it does not provide any guarantee of *fairness* nor *optimality* in resource allocation. To achieve system-level objectives such as *fairness*, we include in the experimental setup a configuration (called *Adaptive*-RTRM) which exploits a two-level run-time management: resource allocation delegated to a centralized resource manager and application-specific parameters controlled by *mARGOt*. To summarise, the main contributions of this chapter can be summarised as follows:

- The problem of resource allocation for computationally intensive OpenCL applications on multi-core OpenCL platforms has been analysed.
- Different solutions for run-time management based on *mARGOt* has been evaluated, to exploit the orthogonality between application-specific knobs and resource allocation.
- A light-weight technique for run-time resource management based on platform sensing at the application level has been proposed.

5.2 Background

In [90], the OpenCL standard is extended to support computation offloading in the automotive industry, by exploiting IP-based in-car networks. However, computation offloading introduces the problem of resource sharing in server consolidation [91]; thus it requires Run-Time Management (RTM) techniques.

The adaptive control technique proposed in [92], called *on-line architecture tailoring*, is based on control theory and provides for continuous self-adaptation of the application. However, this technique is demonstrated for a single application while we target more complex workloads that would require continuous adaptation at the system level.

A distributed RTM approach for homogeneous many-core systems based on game theory is presented in [89]. While distributed approaches suffer from communication overhead and convergence time, centralised solutions [93] require heuristics for optimal resource allocation within a short decision time. In [94], the authors combine design-time and run-time techniques in order to train a global resource manager. A step forward made on top of the previous approach has been done in [95] with a runtime management framework, called ARTE, supported by DSE. Even in this case, the run-time manager is a single one (system-wide) and, at the application level, it provides only the possibility to change the parallelization of the application.

5.3 Methodology

The basic idea of the methodology proposed in this chapter consists of exploiting the orthogonality between application auto-tuning and runtime resource management for compute-intensive OpenCL applications. To this purpose, we have envisioned a twofold approach. On one side, *mARGOt* uses software-knobs at the application level to trade off the performance with Quality-of-Result metrics. On the other side, system resources are partitioned and assigned to the running applications by the resource management layer.

In a plain OpenCL application, the platform resources are managed by the OpenCL runtime at application-level, so an application is enabled to use all devices available on an OpenCL platform and, by default, the entire quota of each device. On a multi-core CPU, for example, the OpenCL runtime binds each application to all the compute units by default and relies on the OS scheduler to assign CPU time to all applications (seen as different processes by the scheduler). The drawback of this approach is that application performance is not predictable, as the number of deployed applications (processes) changes over time. Moreover, it is not possible to control the amount of resource quota for each application: while the OS scheduler is fair in allocating user time to processes, this is unfair for the application, because applications might have different resource requirements, constraints or priority.

The methodology proposed in this chapter aims at extending the domain of application knobs, to manage resource-related parameters without the need of interacting with other actors. In this way, it is possible to avoid any synchronization/communication delay and increase the portability of the application to a new platform, without any additional effort, but generating the application knowledge specific for that platform.

In this chapter, we rely on the ability of *mARGOt* to define application requirements on any metric of interest or software-knob to act as a resource management layer. In this case study, we obtain at design time the performance metrics in the application knowledge by profiling the application in isolation on the target platform. Thus, for computationally intensive applications, the performance metrics – such as throughput – are likely to benefit from increased computational parallelism, thus higher resource usage. On the contrary, our target scenario consists of multiple applications, with different performance and resource requirements, deployed on the same platform.

It is possible to treat any resource-related parameter (such as the computational parallelism) as a generic application-specific parameter. However, plain management of such parameters could lead to system configurations where the total amount of computational parallelism required by the running applications exceeds the system resources. In turns, this would result in a degradation of application performance since the OS scheduler limits the process CPU usage.

To overcome this problem, we propose a resource-aware adaptive layer, which takes into account the CPU usage (as we target multi-core CPU platforms), for self-limiting the application parallelism (e.g. the number of working threads). According to the decision policy stated in Chapter 3, we add a constraint on the process CPU usage, on top of the application-specific ones. We initialise the value of this constraint to the maximum system CPU quota (Γ). At runtime, by monitoring the system CPU usage (γ) and the process CPU usage ($\pi_{measured}$), the application updates this constraint value according to the system evolution. In particular, *mARGOt* selects the most suitable software-knobs configurations among the ones where their expected CPU usage ($\pi_{expected}$) satisfies the following constraint:

$$\pi_{expected} \leq \Gamma - \gamma + \pi_{measured} \tag{5.1}$$

If only one application is running, γ and $\pi_{measured}$ are equal; thus the application is allowed to use the entire CPU resource. Otherwise, if the platform is congested, Γ and γ have the same value, which forces *mARGOt* to select the most suitable configuration among the ones whose expected CPU usage fits the quota assigned by the OS scheduler.

In the experimental results, our solution is compared to a centralised approach, where resource allocation is delegated and coordinated by a system-wide runtime resource manager (SW-RTRM), while *mARGOt* takes local decisions only on application-specific parameters. We will show that it is possible to reach the same average performance predictability with our framework, at the expenses of no guarantee on *fairness* nor *optimal* resource allocation.

5.4 Experimental Setup

We consider a case study based on the Stereo-Matching application [96], implemented with OpenCL APIs [88] and designed to export a set of parameters which impact on both application-specific and platform metrics. Stereo-Matching belongs to a class of applications that exposes throughput-accuracy tradeoffs using application-specific software-knobs [4].

5.4.1 Definition of metrics

The Stereo-Matching application has two metrics of interest, namely the *frame-rate* (measured as [frames/s]) and the *disparity error*, which represents a measure of the average error associated to the application result (the pixel disparity [96]). However, in our tests, we consider only normalised metrics that abstract from the specific application, defined as follows.

Normalized Actual Penalty (NAP)

This metric measures the degree of user satisfaction, with respect to a frame-rate goal set at the application start. The frame-rate goal is a soft real-time constraint, which should be met independently from the machine workload and resource availability.

$$NAP = \frac{GOAL_{measured} - GOAL_{demanded}}{GOAL_{measured} + GOAL_{demanded}} \tag{5.2}$$

Normalized Error

This is a measure of the output quality normalised on the range of valid values so that $ERR = 1$ when the application runs with the configuration that provides the lowest – but still acceptable, according to design requirements – output quality; while $ERR = 0$ when the quality is highest. It was obtained for Stereo-Matching from the *disparity error* (DErr) as follows:

$$ERR = \frac{DErr_{OP} - DErr_{MIN}}{DErr_{MAX} - DErr_{MIN}} \tag{5.3}$$

Difference w.r.t. to off-line profiling

Another metric of interest is the *deviation* (DEV) of the metrics (e.g. cycle period) observed at run-time with respect to the expected values, i.e. the ones profiled at design-time.

$$DEV = \left| \frac{Tcycle_{measured}}{Tcycle_{expected}} - 1 \right| \tag{5.4}$$

Since our tests consider dynamic scenarios, for the NAP and ERR metrics we compute a synthetic value that takes into account the temporal dimension:

$$NAP_{AVG} = \frac{\int NAP(t)\, dt}{\Delta t}\ ,\ ERR_{AVG} = \frac{\int ERR(t)\, dt}{\Delta t} \tag{5.5}$$

5.4.2 Definition of dynamic workload

A dynamic workload, in this chapter, consists of a set of applications with different schedules (start time), amount of data to process (number of frames in Stereo-Matching) and performance requirements (frame-rate). This use-case aims at mimicking the workload expected in server consolidation, which offloads computationally intensive OpenCL applications [90]. Although we use only one type of application (Stereo-Matching), we mimic a dynamic workload by exposing the following parameters:

- *Start delay*: each application instance is started upon user request; thus we use different start times.
- *Amount of input data*: each Stereo-Matching instance is required to process a different number of frames.
- *Frame-rate goal*: soft real-time constraint to guarantee a certain response time, as demanded by the user.

The above parameters are randomly chosen for each Stereo-Matching run, within a range of values shown in Table 5.1.

5.4.3 Platform description

We ran our experiments on two multi-core platforms:

1) AMD platform: NUMA machine with four nodes, each a Quad-Core AMD Opteron Processor 8378 at 2.4 GHz, with 8 GB of RAM per node,

Table 5.1: *Range of values for the random parameters of dynamic workload tests.*

Parameter	AMD	Intel
Number of frames	10-840	
Frame-rate goal [frames/s]	1-7	
Start delay [s]	0-90	
Num. instances	1-6	1-4

running a Linux distribution based on kernel 3.9. OpenCL 1.2 run-time provided by AMD OpenCL SDK v2.8.1.

2) Intel platform: Workstation with Intel Xeon Quad-Core CPU E5-1607 at 3.0 GHz and 8 GB RAM, running a Linux distribution based on kernel 3.5. OpenCL 1.2 run-time provided by Intel OpenCL SDK 2013.

We use the device fission API to partition a multi-core CPU device into sub-devices. The API defines several partition schemes. We use a partitioning by count [88] to create one sub-device of a specific size, in this way controlling the CPU resource usage. However, the OpenCL program is bound to a context, so every time a new sub-device is selected it is necessary to create a new context and rebuild the OpenCL program. The OpenCL program build introduces overhead at run-time, which might limit the benefits of application auto-tuning if the reconfiguration rate is high. To reduce this reconfiguration overhead, we used asynchronous dynamic compilation, a feature of the `clProgramBuild` OpenCL API [88] supported by some OpenCL run-time implementations (e.g. Intel). By passing to `clProgramBuild` a function pointer to a notification routine, the application can continue running in the previous configuration. When the OpenCL runtime finishes building the program, it will call the related routine to notify the application. Our measurements with Intel OpenCL SDK show that synchronous dynamic build of the Stereo-Matching kernels takes $624ms$ on average, while the reconfiguration overhead of asynchronous build is only $58ms$ (10x less).

5.4.4 Run-Time Management description

We consider five Run-Time Management configurations in the experimental campaign:

Plain-Linux

Baseline implementation without SW-RTRM and *mARGOt*. In this configuration, we deploy each application instance as a plain OpenCL application; thus it is bound by default to all processing elements available on the CPU. On the one hand, since there is no SW-RTRM, this configuration relies on the OS to schedule tasks from different applications. On the other hand, the application runs a fixed configuration, with 50% QoS.

Plain-RTRM

For this test we use an open source resource manager, the BarbequeRTRM [93], to allocate resources to the running applications. In BarbequeRTRM, we define application requirements by a set of Application Working Modes (AWMs), identified at design time, each one corresponding to a given amount of required resources. However, if the resource requirement gets higher at run-time, an application can also request to the manager a higher AWM, through a specialised API. Any change in the application resource requirements or in the system resource availability generates an event that triggers a system reconfiguration. The SW-RTRM has complete knowledge of the system state, including dynamic resource requirements of individual applications, which ensures optimal resource allocation w.r.t. system-level objectives – such as fairness, execution priority, reconfiguration overhead and congestion. Even in this case *mARGOt* is not used; thus the application runs with QoS fixed to 50%.

AS-Linux

No SW-RTRM but *mARGOt* is enabled; therefore the adaptation layer can leverage the trade-off between performance and QoS. Although *mARGOt* can control the computational parallelism through a software-knob, the effective resource usage depends on the allocation of CPU user time by the OS scheduler.

RA-AS-Linux

This configuration implements the approach presented in this chapter. Differently from the previous configuration, here the computational parallelism is used orthogonally with respect to the application-specific *knobs*. It implements the technique presented in Section 5.3, based on monitoring of the system CPU usage for *smart* adaptation of the resource requirement.

AS-RTRM

Two-level run-time management, which uses both the centralised resource manager and *mARGOt*. On the one hand, this configuration delegates resource allocation to the BarbequeRTRM, which enforces a fair allocation of platform resources among the running applications. On the other hand, *mARGOt* controls at application-level the trade-off between performance and accuracy metrics, by tuning the parameters orthogonal w.r.t. resource-related parameters.

5.5 Experimental Results

The experiments described in this section have been carried out to assess the benefits of the proposed methodology. In Section 5.5.1, a single stereo-matching application, with some constraints on resource usage, has been used to assess the capability of *mARGOt* to exploit the available trade-offs between performance metrics. In Section 5.5.3, we evaluate the orthogonality between the decision space of *mARGOt*, analysed before, and different RTRM techniques. We consider first an approach that manages resource utilisation as an application parameter in a flat configuration (*AS-Linux*); then, we present the two-level approach based on a centralised resource manager (*AS-RTRM*); finally, we present the proposed technique for efficient resource sharing based on platform sensing (*Resource-Aware AS-Linux*). We conclude this section with a campaign of experiments (Section 5.5.2) with random workloads to compare the different techniques analysed individually in the previous experiments.

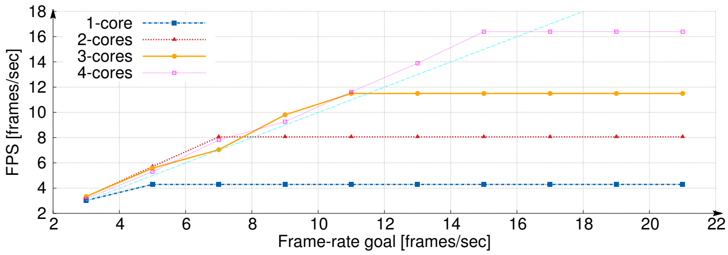
5.5.1 Application Auto-Tuning Results

This experiment aims at assessing the benefits of application adaptivity. It consists of a single Stereo-Matching application deployed on the Intel platform, with 200 frames to process. We repeated the test for each possible number of cores (4 in total on the Intel platform), with the frame-rate goal incremented at each run from 3 to 21 frames/s.

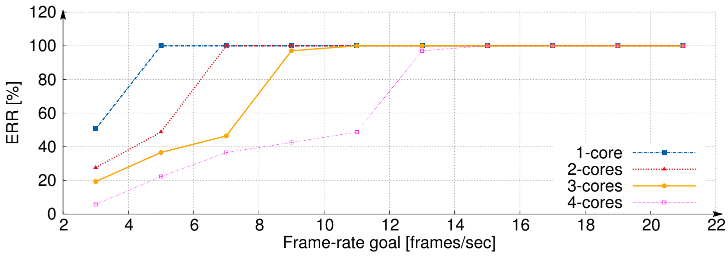
The results are shown in the three plots of Figure 5.1, where the x-axis is the goal value and y-axis represents, in order, the average measured frame-rate (5.1a), the average normalized error (5.1b), and the average NAP (5.1c). With the highest resource availability (4-cores) *mARGOt* can provide 3 frames/s without quality loss (ERR$\simeq$0%). On the contrary, configurations with lower resource availability, show a quality loss which ranges from 20% to 50%, depending on the number of cores. This means that there is a range of goal values, different for each amount of available resources, where *mARGOt* can trade off performance and computation error to meet the goal.

Figure 5.1 shows a similar behaviour but with different thresholds for the maximum frame-rate that they can reach: the test with 1-core provides up to 4.3 frames/s, with 2-cores up to 8.1 frames/s, with 3-cores up to 11.5 frames/s and with 4-cores up to 16.4 frames/s. After these frame-rate thresholds, *mARGOt* is unable to find any suitable software-knobs configuration that satisfies the constraint; thus the NAP value starts growing.

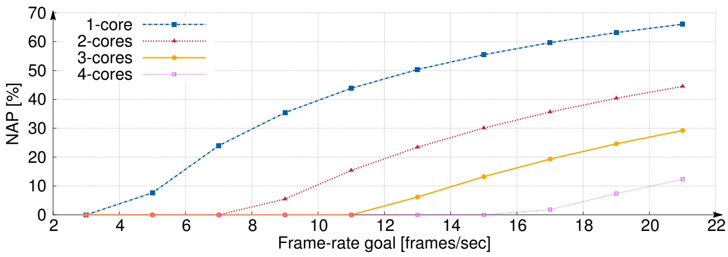
In conclusion, this test demonstrates that for the selected case study,



(a) *Average measured frame-rate vs. frame-rate goal*



(b) *Average normalized disparity error vs. frame-rate goal*



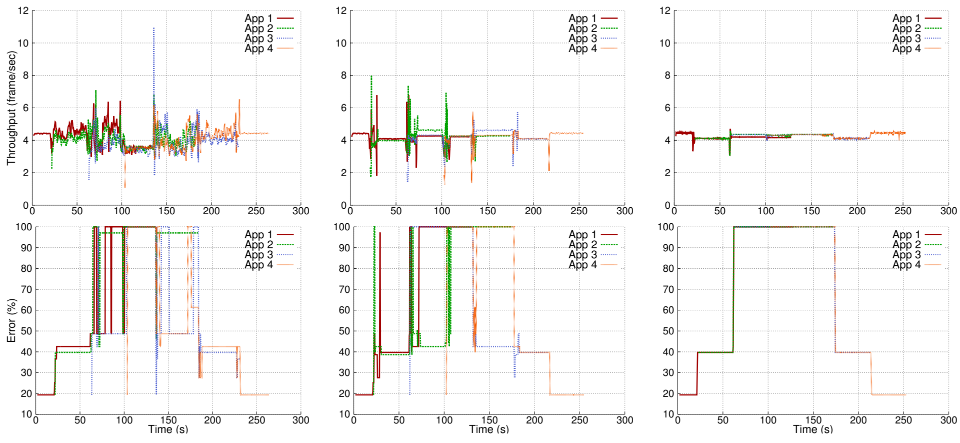
(c) *Average Normalized Actual Penalty vs. frame-rate goal*

Figure 5.1: *Observed frame-rate, normalized error and NAP by varying the frame-rate goal and the number of cores.*

mARGOt can satisfy higher throughput demands by exploiting the possible trade-offs in the objective space in terms of performance versus computation error. The dynamic workloads presented in the next section will benefit from this feature, since in a multi-application deployment configuration, each instance cannot use the full platform, but is constrained to a subset of resources.

5.5.2 Dynamic Workload Results

In this section, we compare in terms of predictability and fairness, the three RTM strategies that use *mARGOt*. The experiment analyses application adaptivity in a sequential scenario. In such a scenario, we executed four



(a) *AS-Linux* **(b)** *RA-AS-Linux* **(c)** *AS-RTRM*

Figure 5.2: *Behavior of the Run-Time Management strategies, in terms of throughput and normalized error.*

Stereo-Matching instances on the Intel platform, with a fixed delay between the start times. The number of frames to be processed by each instance has been chosen to let all the applications run together for approximately $30s$; then they complete their execution at different times. All instances have the same throughput goal (4 frames/s) and *mARGOt* minimises the disparity error. We can logically partition the experiment in two phases. In the first phase new applications are launched, so we can observe how already running applications react when the new applications steal resources. The second phase begins when the oldest instance has completed its execution. In this phase, one by one, all applications leave the execution context, so it is possible to see how the remaining instances exploit the resources that are released. Figure 5.2 represents the three evaluated RTM strategies: AS-Linux (5.2a), the proposed Resource-Aware AS-Linux (5.2b) and AS-RTRM (5.2c). For each configuration, the plots show the expected throughput and disparity error profiled at run-time, in a time window of 300 sec.

AS-Linux

When only one application is running, the throughput is stable, and the disparity error is constant. As soon as the second application is started ($t = 20s$), the throughput of both instances starts oscillating, but the error remains constant. The reason for this is that *mARGOt* does not change the OP (the throughput is above the goal) but, since the total amount of resources demanded doubles the number of cores, the throughput is strongly related to the scheduler policies. The third application executes after $60s$, demanding even more resources, which strengthens the relationship between the OS scheduling and the throughput oscillation. In this case, the measured throughput can go below the goal value, forcing *mARGOt* to select a faster OP, which in turns boosts the oscillation. In conclusion, this configuration is not fair neither predictable.

Proposed Resource-Aware AS-Linux

Here the behaviour is quite different: after an initial transitory period, the constraint on the CPU utilisation forces *mARGOt* to use only software-knobs configurations that match the available resources, preventing the throughput oscillations. Whenever a new application starts or ends, *mARGOt* waits until the CPU usage, of both the system and the application, becomes stable before updating the CPU usage constraint. During these periods, the number of threads might be higher than the number of cores for short periods (e.g. $t = 20s$, $t = 70s$), creating oscillations in the application performance. When the applications partition the available resources among them, the undesired oscillations end. The CPU monitor allows to gain predictability, however – as the disparity error plot shows – this strategy is not fair because there is no coordination in the resource allocation.

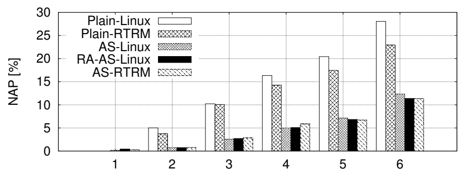
AS-RTRM

Thanks to the centralised coordination, the transitory periods – whenever an application starts or ends – are drastically reduced. This configuration provides the best performance predictability and allocation fairness. However, except for the transitory periods, the performance achieved by the proposed *Resource-Aware AS-Linux* and the *AS-RTRM* are similar (see the throughput plots).

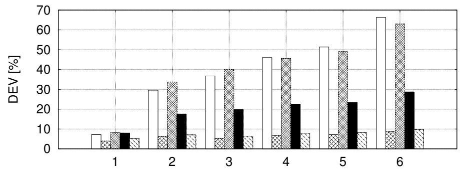
5.5.3 Evaluating RTM Strategies

This section describes the results obtained by deploying a multi-application configuration on both reference platforms. Table 5.1 shows the maximum number of instances and the maximum frame-rate goal for this experiment.

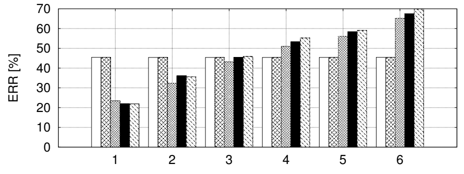
Figure 5.3 reports the results for the test on the AMD platform, while Figure 5.4 for the Intel one. As shown in Figure 5.3a and Figure 5.4a, *Plain-Linux* has the worst NAP metric: although the single application can reach all throughput demands, concurrent execution of applications with different resource demands introduces high penalties on the performance metrics. In this configuration, all applications use by default the entire CPU (device



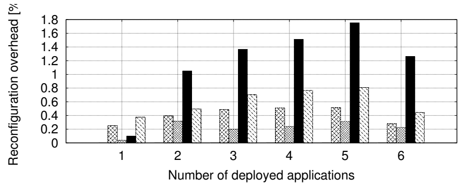
(a) *Average Normalized Actual Penalty (NAP)*



(b) *Throughput degradation w.r.t. to expected*

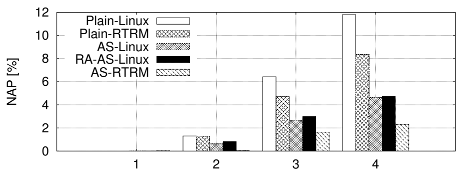


(c) *Normalized output quality loss*

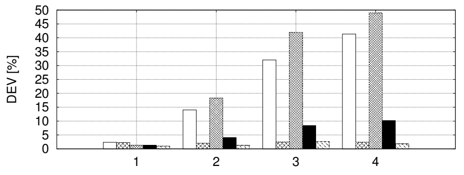


(d) *Reconfiguration time w.r.t. total exec time*

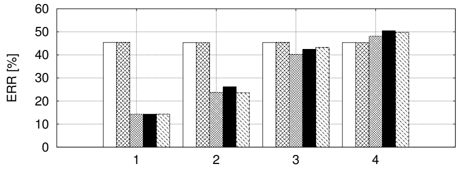
Figure 5.3: *Dynamic workload analysis on the AMD platform.*



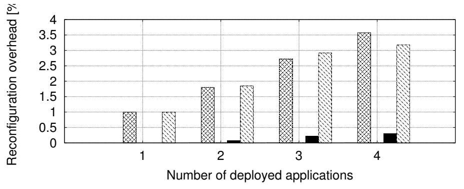
(a) *Average Normalized Actual Penalty (NAP)*



(b) *Throughput degradation w.r.t. to expected*



(c) *Normalized output quality loss*



(d) *Reconfiguration time w.r.t. total exec time*

Figure 5.4: *Dynamic workload analysis on the Intel platform.*

fission is disabled), introducing a high rate of context-switches, which degrades the measured frame-rate. As a consequence of this configuration, the difference between design-time and run-time profiling is the highest. Moreover, this deviation continues to increase when we deploy more concurrent applications. This is an expected result since the OpenCL library relies on the OS scheduler to allocate user time to different applications.

In *Plain-RTRM* the system-wide coordination of resource allocation has the most significant impact on predictability of application performance: indeed, the metric of performance deviation is the lowest for this configuration. The NAP benefits from the execution in a controlled environment since the allocation of CPU cores has the effect of reducing the number of context switches. However, this configuration still fixes the QoS of the application output; thus the reconfiguration options are limited to the computational parallelism.

This is not the case of *AS-Linux*, where the QoS metric (Figure 5.3c and Figure 5.4c) is tuned at run-time to react to variations in the system workload. The error associated with the application output is below *Plain-RTRM* in scenarios with 1-3 instances, above for scenarios with 4-6 instances. The NAP benefits from the wider range of trade-offs, which is lower than in *Plain-RTRM*; however, the predictability of performance metrics is low, as shown by the performance deviation bar (Figure 5.3b and Figure 5.4b).

The proposed *Resource-Aware AS-Linux* overtake this limitation by allowing an application to use computational resources only when they are available. *RA-AS-Linux* performance is better than *Plain-RTRM* in our tests, because *mARGOt* is more reactive than a centralised resource manager; on the other hand, the performance predictability is slightly worse in high contention scenarios. We must notice how the *RA-AS-Linux* approach cannot provide any guarantees on fairness in the resource allocation, nor can support applications with different priority levels whereas the BarbequeRTRM can do.

Finally, the configuration that performs best in all scenarios is the *AS-RTRM*. By combining the benefits of system-wide resource management and application-level autotuning, it is possible to achieve the best performance. However, this configuration requires a more complex software framework, which best-effort applications might not need.

The average Normalized Actual Penalty (NAP) (Figure 5.3a and Figure 5.4a) is the metric that better summarises this analysis. We can observe, as expected, an increasing NAP for all configurations when the workload increases as well. Nevertheless, the adaptive configurations (supported by *mARGOt*) always reduce the NAP with respect to the plain configuration, which in turns enable the application to meet the frame-rate goal much more frequently. However, the NAP metric considers only the processing time and not the run-time management overhead, e.g. the time spent in reconfiguration. Thus Figure 5.3d and Figure 5.4d show also the reconfiguration overhead, with respect to the total execution time of each experiment. The average overhead in the configurations with BarbequeRTRM is 0.4% on the AMD platform and 2% on the Intel platform. In both platforms, we use synchronous OpenCL program build (see Section 5.4.3), because the application execution context is not aware of the system state; thus it cannot control the rescheduling events. Therefore, the difference in reconfiguration overhead depends on the OpenCL run-time libraries: the build of Stereo-Matching kernels takes $154ms$ on AMD and $624ms$ on Intel platform, respectively. On the other hand, the overhead of *AS-Linux* and *RA-AS-Linux* is different between the two platforms for another reason: the Intel platform supports asynchronous OpenCL program build, while AMD does not. This feature can be exploited in the configurations with decentralised resource management because reconfiguration is completely managed by *mARGOt*. On our Intel platform, this results in a 10x reduction of the reconfiguration overhead.

5.6 Summary

In this chapter, we addressed the problem of managing multiple OpenCL applications for server consolidation on multi-core platforms, using the monitors' module of *mARGOt*, as described in Chapter 3. The application we targeted in our tests, Stereo-Matching, can achieve different performance (frame-rate) depending on the computational capabilities of the platform, however more fine-grained control of the resource usage is done through the OpenCL *device fission* API.

We have evaluated different Run-Time Management strategies, in terms of adaptability and predictability in the OpenCL context, reproducing some approaches proposed in the literature [3], [93]. Moreover, we have introduced a light-weight Run-Time Management technique, based on *mARGOt*, which extends the trade-off space of a dynamic application autotuner to resource-usage control. This technique, targeted to compute-intensive applications, allows taking a local decision on resource utilisation at the application level, for efficient resource sharing. Moreover, it enables applications to act as autonomous agents, without coordination among them, differently from known distributed approaches.

Our tests show that the average performance of the proposed approach is very close to the performance achieved with a combined approach based on a centralised resource manager; at the same time, our approach is more portable and less intrusive from an application design point of view.

CHAPTER 6

A Seamless Online Compiler and System Runtime Autotuning Framework

In this chapter, we address the problem of performance portability, concerning options and OpenMP parameters. In particular, we propose a structured approach, named SOCRATES, which uses *mARGOt* to automatically tune the application and an aspect-oriented language to remove the integration effort in the target source code. On the one hand, this chapter aims at mitigating the performance portability problem. On the other hand, it aims at lowering the integration effort of *mARGOt*, as described in Chapter 2, since it targets a well-defined set of software-knobs.

6.1 Introduction

Performance portability across different computing platforms is a challenging problem for application developers working on different computing fields from embedded to HPC systems. The problem is that application performance is strongly dependent on the underlying target platform, system runtime, and input data. Ideally, the solution can be expressed as a *morphable* code capable of adapting to the environmental conditions. However, this approach faces several challenging problems not yet solved. Among them, we can mention that writing such a kind of code would require a flexible high-level language capable of expressing functional aspects, that can be easily manipulated and customised for later compilation and code generation phases.

In the past, customising applications without a complete rewriting of the code, in terms of parallelism and compiler transformations, has been envisioned as a promising path [97, 98]. These approaches are typically based on the tuning of the application, compiler and system runtime knobs before the actual code deployment, thus finding a one-fits-all configuration for the target platform. However, selecting the most suitable configuration can be a hard task, if we consider that the application workload and resource partitioning change dynamically and the energy/power budget can evolve depending on external events. Only a few recent efforts (see, e.g., [3, 60]) are applying strategies once the application has been deployed on the target system. The main problem of runtime solutions for application tuning is that they require a high-level of intrusiveness in the source code. Indeed, the original source code implementing the functional aspects should be enhanced with glue code needed to profile, monitor and configure the application according to extra-functional aspects.

This chapter introduces a structured approach, called SOCRATES, for the runtime selection of the most suitable application configuration in terms of compiler flags and parallelism parameters of the OpenMP runtime. The main contribution of SOCRATES is to offer runtime autotuning features while avoiding any manual intervention by the application developer. The proposed approach uses an aspect-oriented language, LARA [16], to implement the separation of concerns between the functional and extra-functional parts of the application, while an application-level autotuner, *mARGOt*, is integrated for the optimal configuration selection. LARA automatically performs all changes to the application code required by SOCRATES. Furthermore, SOCRATES supports an energy efficient execution by introducing energy consumption as a key variable to be considered at runtime.

6.2 Background

Aspect-Oriented Programming [99] (AOP) provides mechanisms to express and deal with cross-cutting concerns, promoting more modular and less polluted code than alternatives (such as pragmas). One example of such an AOP approach is LARA [16]. LARA differs substantially from annotation-based approaches [97,100,101], as code transformations and compiler mapping strategies are described in a separate file, allowing a high-level of reuse. Moreover, LARA offers finer-grained manipulation over the transformations, giving developers a very precise control of the application at the expression and statement levels. Previous approaches exist for the specification of code transformation and optimisation strategies [102, 103]. These enable the user to write recipes, separated from the original application, specifying a sequence of transformations, but without offering any possibility to select among them at runtime.

Concerning the autotuning of OpenMP parameters, several offline approaches have been presented in literature [104–106] and implemented in commercial tools[1]. While the approaches proposed by Mustafa and Eigenmann [104] and Wang et al. [105] analysed the effect of code transformations and OpenMP parallelization on performance and energy consumption for tuning purposes, Tiwari et al. [106] propose an entire framework for offline autotuning based on Active Harmony [107] and CHiLL [108]. The main limitation of these works is that their effectiveness is strongly dependent on the tuning decisions are taken at profile time to find the one-fits-all solution.

Overall, the presented frameworks focus on transformations for scientific computing and mainly on tuning performance portability. SOCRATES provides such features with a more general approach thanks to the significant flexibility offered by the two key components (LARA DSL and *mARGOt*). The proposed approach enables programmers to customise, in a non-intrusive way, the source code to be then runtime tuned according to the dynamicity of the environmental conditions and application requirements.

6.3 Proposed Methodology

The main goal of the SOCRATES framework is to provide to the application developer an energy-aware framework to enhance the application with a kernel-level compiler autotuning and adaptation layer in a seamless way. In particular, the starting point of the approach is a generic source code that describes the functional behaviour of the application, i.e. $o = f(i)$ where a generic function f computes the desired output o from the given input i. The framework performs two major actions on the original application to reach the adaptivity goal. The first action consists of transforming the application into a *tunable* version, enhancing its structure to take as input a set of knobs $(k_1, k_2, \ldots, k_n)$ that affect its behavior, i.e. $o = f(i, k_1, k_2, \ldots, k_n)$. The idea is that a change in the configuration of the knobs results in a

[1] https://software.intel.com/en-us/articles/intel-software-autotuning-tool

change of the extra-functional property (EFP) of the application f and its output o. Examples of EFPs of the function f might be execution time and power consumption, while EFPs of the output o might be solution accuracy and output file size. The second action consists of enhancing the tunable version of the application with the intelligence needed to configure its knobs dynamically, according to application requirements and environmental conditions. Thus, it enhances the application with an adaptation layer that provides the ability to monitor its behaviour and select the most suitable configuration.

Even if the overall approach is suitable for different contexts, we designed SOCRATES to address the following autotuning space:

Compiler Options (CO) : This knob represents a combination of compiler flags. We used four standard optimization levels from gcc: `Os`, `O1`, `O2`, `O3`, in addition to specific transformations such as: *-funsafe-math-optimizations*, *-fno-guess-branch-probability*, *-fno-ivopts*, *-fno-tree-loop-optimize*, *-fno-inline-functions*, *-funroll-all-loops* derived from [109];

Number of threads (TN) : This knob sets the number of OpenMP threads between 1 and the number of logical cores;

Binding Policy (BP) :This knob sets the OpenMP binding policy: `spread` or `close`. We set the environmental variable OMP_PLACES to `cores`.

Figure 6.1 shows the SOCRATES toolchain. The proposed methodology targets applications with one or more kernels representing different phases of the computation. For reducing the compiler space, the toolchain uses GCC-Milepost [110] to analyse every kernel of the original code and to extract code features. Then, the compiler autotuning framework COBAYN [15] is used to infer and extract the most promising compiler flags for every kernel. We generated several versions of the kernel, according to the autotuning space by using a LARA-controllable toolbox, while *mARGOt* enhances the code with runtime autotuning capability. The enhanced code is then profiled for all the alternatives to create the *application knowledge* required by the final adaptive application binary.

6.3.1 Reducing the compiler space complexity

The first step of SOCRATES consists of pruning the compiler optimisation space. An appropriate methodology is to select efficiently the most promising compiler options given a target application. To this end, we

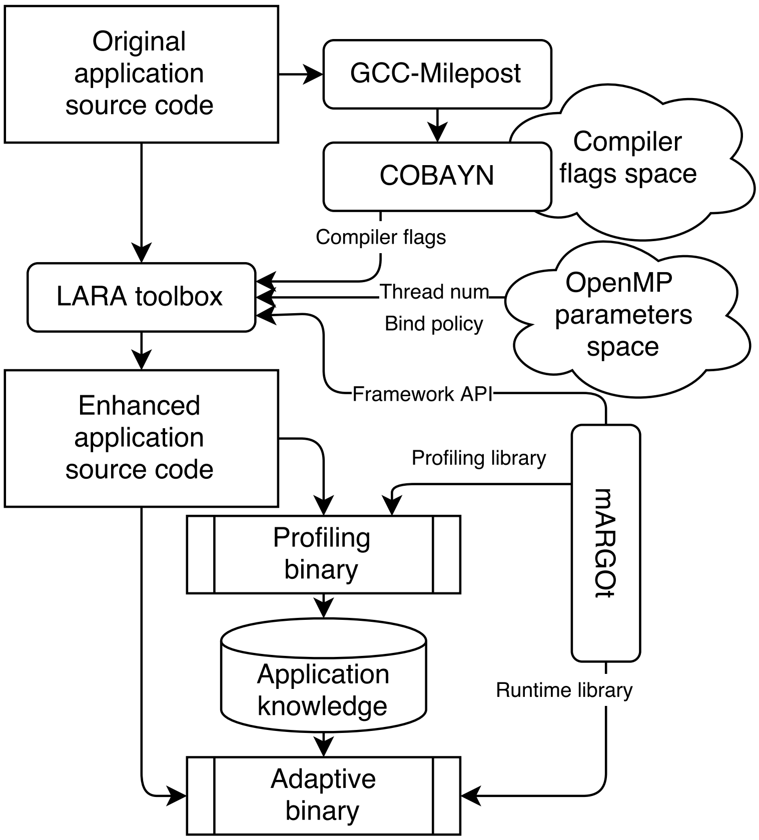


Figure 6.1: *Tool flow of the SOCRATES approach from the original application source code to the generation of the application adaptive binary.*

adopted the COBAYN framework to select the best optimisation passes. COBAYN is an autotuning framework that identifies the most suitable compiler optimisations by using Bayesian Networks (BN). It uses application characterization to induce a prediction distribution by an iterative compilation methodology. This technique identifies a suitable set of compiler optimisations to be applied to the target kernel, thus reducing the cost of the compiler optimisation phase. We used GCC standard optimisation levels and COBAYN predictions as reduced design space for the compiler flags. Application characterization is done by extracting static code features by GCC-Milepost, while COBAYN has been adapted to work at kernel function granularity. In SOCRATES, we used the compiler space adopted in the original COBAYN paper (128 flags combination) by reducing it to four alternatives.

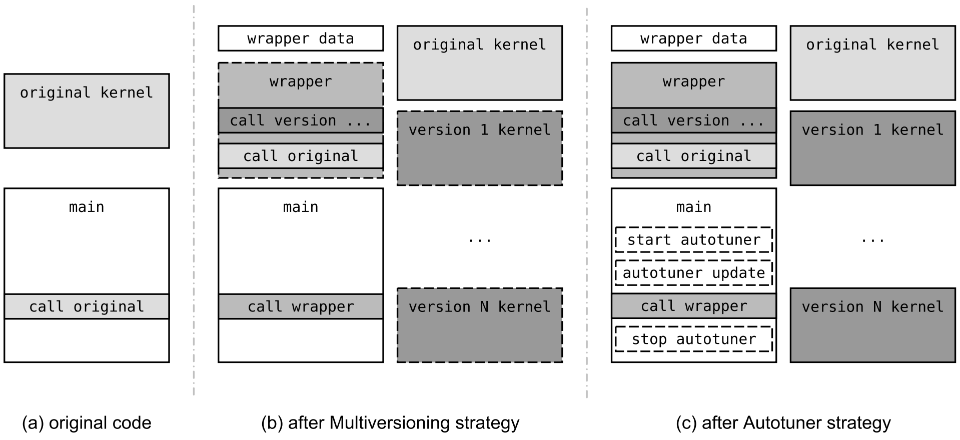


Figure 6.2: *Example of the automatic application code transformation from the original code (a) to the final adaptive code (c).*

6.3.2 Integration issues

LARA strategies are used to enhance automatically the original source code for making the application tunable and to integrate the *mARGOt* framework. In particular, we use code transformation and code insertion strategies specified in LARA aspects to interact with the application source code. MANET [111] is used as a source-to-source compiler to weave the crosscutting concerns described in the aspects in C applications.

There are two main strategies: *Multiversioning* and *Autotuner*. Figure 6.2 shows an example of how the application code evolves during the entire process: from pure functional code to adaptive code, ready to be deployed. The first strategy, *Multiversioning*, generates different versions of the target kernel and a mechanism to choose which version to call at runtime. The autotuning space is composed of GCC compiler flags, binding policy and the number of OpenMP threads. The first two parameters must be statically defined, while the number of OpenMP threads can be controlled dynamically. The first action of the *Multiversioning* strategy clones the kernel several times. Each function clone represents a different version of the kernel in terms of compiler options and binding strategy. No cloned versions have been generated to manage the number of threads variable because it does not require to be known at compile time. For each function clone, the strategy inserts GCC pragmas to set compilation flags (e.g., `#pragma GCC optimize ("O2,no-inline")`) and OpenMP pragmas (e.g., `#pragma omp for num_threads(NT) proc_bind(close)`) to configure the parallelization of the kernels. The strategy also generates a wrapper, which selects the target version of the kernel, according to control variables. Afterwards, the strategy replaces each call of the kernel from application source files, with a call to the wrapper (see Figure 6.2b). The entire process is fully automated.

The second strategy, *Autotuner*, is responsible for integrating *mARGOt* into the application. First, the connection between the generated kernel versions and the autotuner is made by exposing variables containing the current configuration. Then, the strategy inserts the required headers and the initialisation function call at the `main` function. Finally, as shown in Figure 6.2c, the strategy surrounds the call to the wrapper with the *mARGOt* API code to monitor EFPs and to update the most suitable configuration.

6.4 Experimental Results

The platform used for the experiment is a NUMA machine with two Intel Xeon E5-2630 V3 CPUs for a total of 16 cores with hyperthreading enabled and 128 GB of DDR4 memory (@1866 MHz). The experimental campaign is based on 12 apps from the Polybench/C benchmark suite [112]. We used the SOCRATES framework to automatically generate the additional code without any manual intervention on the target applications. In the experimental campaign, we considered the autotuning space presented in Section 6.3. We used *mARGOt* to perform two tasks. The first one profiles the application to perform a Design Space Exploration (DSE) and build the knowledge required by the autotuner. The second task tunes the application at runtime according to application requirements given by the experiment. To evaluate this approach, we used a full-factorial analysis over the design space; however, our approach is agnostic with respect to the used DSE strategy.

Table 6.1 presents some metrics regarding the developed strategy and its application to each benchmark code. *Att* is the number of attributes checked in the LARA strategy about the source code of the application, including function signature information and OpenMP pragma information. *Act* is the number of actions performed on the code, including code insertions, cloning and pragma insertion. The *LOC* columns represent, in order, the number of logical lines of code of the original (*O*-) benchmark, the weaved (*W*-) benchmark and their difference (*D*-). The number of logical lines of source code in the complete LARA strategy is 265. This is used to calculate the *Bloat* metric [113], that roughly estimates how much code is weaved in the original application per line of code in the aspect files.

These data present an overview of how complicated, time-consuming

Table 6.1: *Metrics collected from the application of LARA strategies.*

Benchmark	Att	Act	O-LOC	W-LOC	D-LOC	Bloat
2mm	698	378	136	2068	1932	7.29
3mm	708	378	125	1801	1676	6.32
atax	684	250	81	1071	990	3.74
correlation	1347	410	138	2366	2228	8.41
doitgen	561	218	72	1018	946	3.57
gemver	631	218	94	1008	914	3.45
jacobi-2d	4429	154	145	2918	2773	10.46
mvt	339	154	64	571	507	1.91
nussinov	551	154	78	1356	1278	4.82
seidel-2d	445	154	47	565	518	1.95
syr2k	376	186	66	749	683	2.58
syrk	370	186	62	743	681	2.57
Average	928	237	92	1353	1261	4.10

and error-prone it would be to execute these tasks manually. For instance, take the case of *2mm*, in the first row. The weaver automatically inspects multiple points in the program code, checking the value of 698 attributes and performs transformations (or insertions) on 378 of the inspected points. The resulting code has a number of logical lines of code that is an order of magnitude larger than the original one. From the *Bloat* value for *2mm*, we can see that, on average, we insert 7.29 lines of C code per line of LARA aspect code. The large differences from benchmark to benchmark are explained because their kernels may be very different in size and have different numbers of loops, which are closely related to the number of lines of code and actions performed, respectively.

Figure 6.3 shows the experiment that analyses the trade-off space between power consumption and throughput of the target kernels by using a full-factorial DSE. In particular, it shows the distribution (as boxplot) between the throughput and the average power consumption. The values on the y-axis represent the distribution of the target metrics, for each evaluated application, considering only the Pareto-optimal configurations. Given the large power/performance swing, there is no one-fits-all configuration, thus confirming the importance of the proposed approach.

This experiment aims at assessing the benefits of the proposed approach when autotuning is done statically (compile-time) according to a given power budget. Figure 6.4 shows the results in terms of execution time and

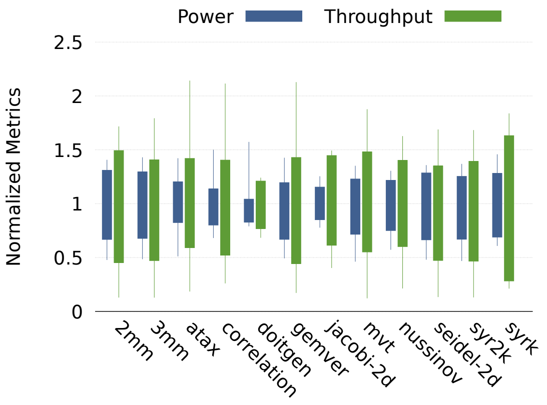


Figure 6.3: *Power/Throughput distribution of the Pareto-optimal software-knobs configuration, leading to an optimal trade-off according to user requirements.*

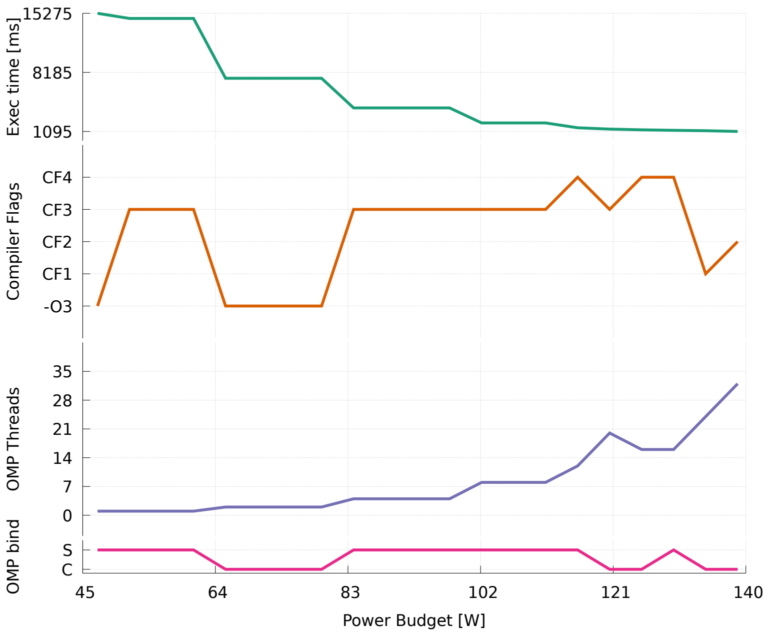


Figure 6.4: *Static analysis of the proposed approach, that aims at minimizing execution time given a constraint on power budget (x-axis).*

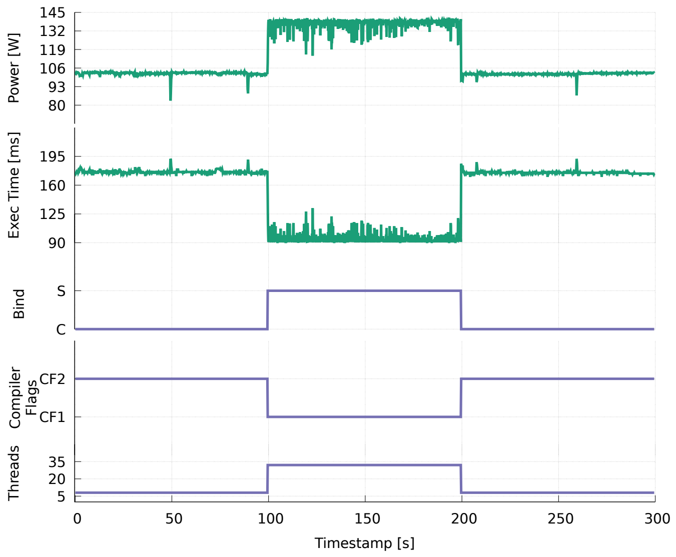


Figure 6.5: *Execution trace of the 2mm application by varying application requirements at runtime each* 100 *seconds.*

the selected configuration (y-axis) while changing the available power budget, for the target application (2mm). The plot shows the power-performance trade-off available in the Pareto curve and also highlight that there is not a clear trend on the selected software-knobs. For this experiment, the custom flag combinations suggested by COBAYN (*CF1-CF4*) are: CF1) *O3, no-guess-branch-probability, no-ivopts, no-tree-loop-optimize, no-inline*; CF2) *O2, no-inline,unroll-all-loops*; CF3) *O2, unsafe-math-optimizations, no-ivopts, no-tree-loop-optimize, unroll-all-loops*; CF4) *O2, no-inline*.

The last experiment shows the runtime effectiveness of SOCRATES. Figure 6.5 reports an execution trace of the target application (*2mm*), when the requirement changes from an energy-efficient policy optimizing Throughput per Watt^2 (Thr/W^2) – in the 0s-100s interval – to a performance-oriented policy optimizing the Throughput – 100s-200s interval – and back to optimizing Thr/W^2 – 200s-300s interval. When changing from an energy-aware to a performance-oriented policy (and vice-versa), we can notice how the parameter sets change dynamically to meet the requirements.

6.5 Summary

The contributions of this chapter are twofold. On the one hand, it addressed the performance portability problem, in terms of compiler options and OpenMP runtime parameters. On the other hand, this chapter aimed at lowering the integration effort from application developers, as mentioned in Chapter 2. The main outcome is an autotuning framework, named SOCRATES. It leverages *mARGOt* for selecting the most suitable software-knobs configuration automatically. While it uses the LARA aspect-oriented language to significantly lower the integration effort from the application developers point of view. SOCRATES has been applied to the OpenMP Polybench suite by varying application requirements. Experimental results show how SOCRATES can reach significant benefits in terms of exploiting runtime energy-performance trade-offs in a dynamic environment.

Part III

Application case studies

CHAPTER 7

Tuning a Server-Side Car Navigation System

In this chapter, we focus on an application domain to show how it is possible to significantly improve the computation efficiency by using *mARGOt*. The analysed application falls in the context of a navigation system, and it is the one used in Section 4.3. However, this chapter describes in more details the relationship between input features and the quality of results. Moreover, we use the LARA aspect-oriented language also in this case study, for hiding the extra-functional concerns from the application source code.

7.1 Introduction

In smart cities, the trend is to combine and automate several common tasks to ease the life of citizens. Among these tasks, traffic estimation and prediction plays a central role: it is used not only to avoid traffic congestion, which allows having predictable travel times but also to reduce car emissions. Considering the rising wave of self-driving cars, the amount of car navigation requests will increase rapidly together with the need for real-time updates and processing on large graphs representing the urban network. This trend imposes larger and more powerful computing infrastructures composed of HPC resources.

Concerning the algorithmic problem, car navigation is one of the main problems of applied theoretical research. The Dijsktra's shortest path algorithm is used for finding the optimal path between two vertices in a weighted graph representing a road network. Apart from single navigation between two points, navigation algorithms are used in various systems for solving larger optimisation problems, like route planning for a fleet of package delivery vehicles, waste collection management or traffic optimisation in a smart city [114]. Definition of the optimal path is based on the type of used weights of the graph edges. The shortest path is based on the geographical distance between two adjacent vertices of a graph. The fastest path is based on the time needed to cross a particular edge. There might be more complex criteria; however, their description is out of the scope of this chapter. Time needed to cross a particular stretch of road can be affected by various elements, such as accidents, traffic congestion, road work and so on. At the basic level, the upper legal limit of speed is used, based on the assumption that each vehicle travels at the same speed. This approach can be vastly inaccurate due to the natural behaviour of traffic.

With the increasing availability of historical traffic monitoring data, there are several research efforts to determine the average speed on road networks by using statistical analysis and various models. However, a single speed value is still not very useful as it does not reflect the stochastic behaviour of the traffic. The probability distribution of the speed at a certain time enables to incorporate low probability real world events that can cause major delays and affect traffic over vast areas. By incorporating probability distribution to the computation, the system can compute the probability of arrival time within a certain time-frame which can be useful for more precise route planning. This problem is called *Probabilistic Time-Dependent Routing* (*PTDR*).

A scalable algorithm for solving the PTDR problem based on Monte Carlo simulations has been presented in [82] [83] and represents the base for our work. In particular, the algorithm uses probability distributions of travel time for the individual graph edges to estimate the distribution of the total travel time and it is integrated into an experimental server-side routing service. This service is deployed on an HPC infrastructure to offer optimal performance for a large number of requests as needed by the smart city context. The *PTDR* algorithm employed in this work simulates a large number of vehicles driving along a determined path in a graph at a particular time of departure. The speed of vehicles on individual roads is sampled from the speed probability distribution (also called speed profile) associated to the graph edge. The number of samples is a parameter that directly affects the informational value of the output as well as its computational requirements. Given a large number of requests to be served, even small changes in the workload can affect the overall HPC system efficiency. While the original version was based on a worst-case tuning of the number of samples [82], and given that a reactive approach [115] is not a viable solution due to the overheads, in this chapter we present a proactive method for dynamically adapting the number of samples for the Monte Carlo (MC) based PTDR algorithm.

In particular, the main contributions of this chapter can be summarised as follows:

- A methodology has been proposed for self-adapting the PTDR algorithm presented in [82] [83] to the input data in a proactive manner, maximising its performance while respecting the output quality level;
- A probabilistic error model has been proposed to correlate the input data characteristics with the number of samples used by the Monte Carlo algorithm;
- An aspect-oriented programming language has been adopted to keep separated the functional version of the application from the code needed to introduce the adaptivity layer.

7.2 Background

Determining the optimal path in a stochastic time-dependent graph is a well-studied problem which has many formulations [116]. Our approach is closest to the *Shortest-path problem with on-time arrival reliability* (SPOTAR) formulation. It can be seen as a variant of the *Stochastic on-time arrival* (SOTA) problem, for which a practical solution exists as shown in [117]. These algorithms have the objective of maximising the probability of arriving within a time budget and are related to optimal routing in stochastic networks. However, there are not many solutions for the time-dependent variant of both of the problems. In [116] authors show practical results for the time-dependent variant of SOTA, simultaneously in [118] authors elaborate on the complexity of existing theoretical solutions of the SPOTAR problem and show how it can be extended with time dependency. There are many other papers which show various theoretical approaches for the SOTA problem, including some practical applications [117] [119] [120] [121]. Solution to the SPOTAR problem based on *policy-based* SOTA as a heuristic is presented [120]. However, the authors assume that the network is time-invariant, which is not true in real cases if considering long paths. The solution is also unusable in on-line systems as its scalability to graphs representing real-world routes is not sufficient.

Our approach follows the same philosophy presented in [82, 83] where the authors provide an approximate solution of the time-dependent variant of the SPOTAR problem based on Monte Carlo simulations. As shown in Section 7.3, our approach uses the k-shortest paths algorithm [122] [123] [124] to determine the paths for which the travel time distribution is estimated. This separation allows us to implement the approach in an on-line system which provides adaptive routing in real-time. Given the Monte Carlo nature of the algorithm, to improve the efficiency of the PTDR calculation, we have two main alternatives [125]. The first is the sampling efficiency, while the second is the sampling convergence. In both cases, the algorithm optimisation is reached by exploiting the iterative nature of the Monte Carlo simulation. Several techniques have been proposed to determine what is the next sample to be evaluated to maximise the gathered knowledge [125, 126] and to improve the sampling efficiency. However, in the implementation under analysis this has been discarded because our goal is to exploit the parallelism of the underlying HPC architecture [83] that excludes any iterative approach to the Monte Carlo. For the same reason also the approaches that require a statistical property evaluation after every iteration [115], checking if the error is acceptable, cannot be considered acceptable. Both approaches would be too time-consuming and, how it has been already analysed in [82], for the specific problem the number of samples has to be chosen a priori in a proactive rather than in a reactive manner.

A two-step approach for solving the Monte Carlo problem has been envisioned in [127]. Similar to our work, the authors suggest to have a first shot of a reduced number of samples to provide an initial approximate solution as fast as possible, and then to refine the output to the required accuracy in successive iterations. In the proposed context, this idea suffers from two main problems. First, it is suitable for scientific work-flows where an intermediate solution is used to trigger next computations, and it is not our case. Second, in the iterative phase, it suggests a reactive approach rather than a proactive one, that we already discussed to be necessary for the specific PTDR problem.

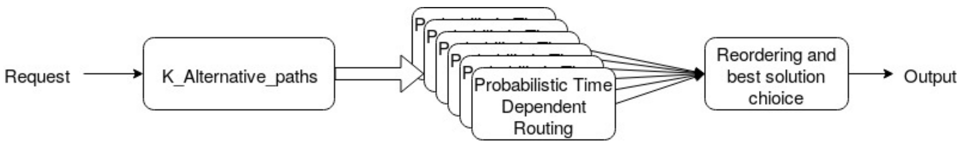

Request
K_Alternative_paths
Probabilistic Time Dependent Routing
Reordering and best solution chioice
Output

Figure 7.1: *The complete navigation infrastructure for serving a single request.*

7.3 Monte Carlo Approach for Probabilistic Time-Dependent Routing

So far, many theoretical formulations and several algorithms have been developed for solving the problem of computing the travel time distribution [116]. In this chapter we consider a *path-based* approach (*SPOTAR*) where the paths are known *a-priori* and travel-time distributions are determined subsequently for each one of the paths [128].

In the context of the complete traffic navigator application illustrated in Figure 7.1, our focus is on the efficient estimation of the arrival time distribution (*PTDR - Probabilistic Time-Dependent Routing* phase). More in detail, the three main steps of the application can be described as follows: (i) The first step consists of determining *K alternative paths* to be passed to the next steps. In the navigation scenario, the identification of the shortest path is not enough to determine a good solution, if no traffic information has been considered. Thus alternative routes derived by algorithms for determining *k-Short Paths* with limited overlap have to be adopted in this step [123] [124] [122]. This first phase is out of the scope of this chapter; (ii) For every path selected by the previous step (K-alternative paths), the computation of the travel time is done using the Probabilistic Time-Dependent Routing module. While the exact solution to the travel-time estimation (PTDR) has exponential complexity, in this work we efficiently approximate the solution of the SPOTAR problem by adopting a Monte Carlo sampling approach [82]; (iii) The final step gathers the timing information provided by the k instances of the PTDR module for every single request and selects the best path to be given back to the user. This phase does not provide a single route but reorders the list of k paths determined by the first step according to the timing distributions determined in the second phase and user preference [129].

This three-step approach of the whole navigation application allows us to implement an approximate solution to the SPOTAR problem, which can be used online in a system to serve a large volume of routing requests.

Our definition of a probabilistic road network is similar to the definition of the stochastic time-dependent network as described by Miller-Hooks [128], except for the segment travel times, which has been substituted by the speed probability distribution (*speed profile*) for a given time of departure within a week. Formally, it can be defined as follows. Let $G = (V, E)$ be a well connected, directed and weighted graph, where V is the set of vertices and E is the set of edges. Each vertex represents a junction or some important point corresponding to geospatial properties of the road, while edges represent the individual road segments between the junctions. Each path selected by the first phase of the application (i.e. K-Alternative paths) can be formally represented as a vector of graph edges $S = (s_1, s_2, \ldots, s_n)$, while $S_p \subseteq E$ and n is the number of road segments in the path.

Using a travel time estimation function, we are interested in estimating the travel time θ as $\hat{\theta}_{S,t,P_S}$ where S is the given path, t is the departure times and P_S are the probabilistic speed profiles for the segments in S. More in detail, $t \in T$ is a departure time within a set of possible departure times which divide a certain timeframe to a set of intervals $T = \{t : t = n \cdot \phi, n \in \mathbb{N}\}$ [130], where the length of the interval ϕ is determined by input data. P is the set of probabilistic speed profiles for the entire graph edges E, where $P_S \subseteq P$. Each speed profile $p \in P$ is represented by a set of discrete speed values and assigned probabilities. The number of speed values depends on the method used for deriving the profiles from historical traffic monitoring data, while the minimum and maximum values represent the congestion speed and the free flow speed respectively.

In our work, the time frame is set to *one week* and $\phi = 900s$ (15 minutes). This approach reflects traffic variations during the various hours of the day and for all the days of a week. By extending the time frame, other factors can be included, such as the seasons or holidays. The number of speed values has been set to 4 levels according to the characteristics of the input data used for the creation of the speed profiles.

Focusing on the SPOTAR problem, we are not interested in a single travel time value θ, but we require to calculate the probability distribution of the arrival time. Given the previous formalisation of the problem, the travel time distribution can be estimated by traversing the path segments together while considering the speed profile distribution. In particular, we can define a tree where each layer represents a segment in the selected path [82]. The tree root is the starting segment, while the end segment is on the leaves. Each node in all layers of the tree has l children, where l is a number of the discrete speed values for each segment, and the tree depth corresponds to the number of selected path segments $|S|$. Each edge in

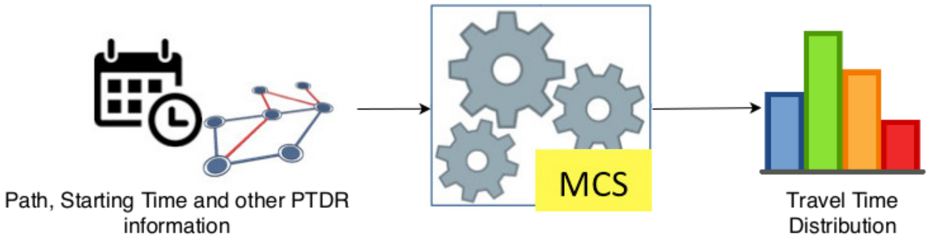


Figure 7.2: *The original approach for PTDR routing based on Monte Carlo simulations to derive the travel time distribution.*

the tree is annotated by the discrete speed value, its probability, and by the length of the considered segment.

Travel time can be computed by a depth-first search (DFS) while selecting an arbitrary child node at each level of the tree. The travel time value is then the sum of the time spent in each segment (length/speed), while the probability of that value is the product of the probability on each edge of the traversal. Each traversal corresponds to a single car travelling along the entire path. The exact solution is obtained by an exhaustive search over all the possible paths between the root node and all the leaves. This approach is clearly not efficient since it scales exponentially with the number of segments in the path.

A Monte Carlo-based approach can be successfully employed in this case. By generating a large number of random tree traversals, enough samples can be obtained to estimate the final distribution. We define this final distribution, which is a collection of θ values ($\theta_1 ... \theta_x$) obtained through the Monte Carlo simulation $MCS(x, i)$, where x is the number of random tree traversals, and i is the input set of the $\hat{\theta}$ function (i.e. S, t, P_S).

Given that travel times usually have a long-tailed distribution due to inherent properties of the traffic (e.g. rare events such as accidents) a large number of samples is needed to estimate the travel time distribution with sufficient precision. Regarding the definition of the number of samples for the Monte Carlo simulation, the particular implementation of the PTDR kernel cannot rely on run-time stability analysis of the output. Each tree traversal (a sample of the Monte Carlo simulation) is independent of the others. Thus this problem is perfectly suitable for parallel computing architectures, such as modern CPUs or accelerators. However, it is necessary to know apriori the number of travel time estimations required to build the final distribution for exploiting this parallelism efficiently.

To summarise, the PTDR algorithm can be seen as in Figure 7.2, where all the information regarding the request are provided to the Monte Carlo simulation (MCS) capable of returning the predicted travel time distribution for the given route.

7.4 The Proposed Approach

The Monte Carlo simulation is designed to use a given number of samples x for every run. Based on a conventional approach, this number is selected according to the worst-case analysis, and it is the lowest number of samples always able to reach a target precision [83]. In this section, we present the proposed technique adopted to select at runtime the number of samples for the Monte Carlo simulation according to the input data characteristics.

Before moving on the methodological part, let us better define the specific context of the problem. In particular, even if we are interested in the travel time distribution, our goal is to know a value τ_i to guarantee with a certain probability that the travel time will be within that value: $P(\theta < \tau_i) \geq y$ where i has been defined as the input set of the travel-time function. The value τ_i is the output of the PTDR phase. In the following, we characterise that value with an additional property $\tau_{i,y}$, where y is the probability that the travel time will be lower than τ.

Using the Monte Carlo simulation, we can estimate the value of $\tau_{i,y}$ using x samples as follows $\hat{\tau}^x_{i,y} = MCS(x, i, y)$. In particular, we estimate the value $\hat{\tau}^x_{i,y}$ by selecting the y-th percentile of the finite-sample distribution obtained from the Monte Carlo simulation (i.e. if $y = 95\%$ then $\hat{\tau}^x_{i,y}$ is the 95^{th} percentile of the distribution).

In the context of this work, we are interested in minimizing the execution time of the function MCS, while limiting the prediction error defined as $error^x_{i,y} = \frac{|\tau_{i,y} - \hat{\tau}^x_{i,y}|}{\tau_{i,y}}$. In particular, the target problem can be expressed as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & cpu\_time^x_i \\ \text{subject to} \quad & error^x_{i,y} \leq \epsilon \end{aligned} \tag{7.1}$$

where ϵ represents the upper bound on the computation error. We want this error to be relative to the output of the MCS, that is the desired percentile of the predicted travel time. In this way, we can abstract from the actual path. Given the tight correlation between the execution time and the used number of samples x, the previous problem can also be simplified by considering the minimisation of x instead of the cpu_time. According to Monte Carlo properties, we can derive that $\tau_{i,y} \equiv \hat{\tau}^{\infty}_{i,y}$, where $\hat{\tau}^{\infty}_{i,y}$ is the output of the MCS function computed using an infinite number of samples. Thus, we can rewrite the error as

$$error^{x}_{i,y} = \frac{|\hat{\tau}^{\infty}_{i,y} - \hat{\tau}^{x}_{i,y}|}{\hat{\tau}^{\infty}_{i,y}} \tag{7.2}$$

Due to the Monte Carlo properties [131], the value $\hat{\tau}^{x}_{i,y}$ is a random variable, asymptotically normally distributed with mean $\mu_{\hat{\tau}^{x}_{i,y}}$ and standard deviation $\sigma_{\hat{\tau}^{x}_{i,y}}$. In particular, according to the central limit theorem [132], while considering enough values of samples the mean value does not depend on the number of Monte Carlo simulations, and the standard deviation decreases by increasing the number of Monte Carlo simulations. Given that, we can define the error as characterized by a normal distribution with mean 0 and a standard deviation $\sigma_{\hat{\tau}^{x}_{i,y}}/\mu_{\hat{\tau}^{x}_{i,y}}$. In the following, we refer to the standard deviation of the error as $\nu_{\hat{\tau}^{x}_{i,y}} = \frac{\sigma_{\hat{\tau}^{x}_{i,y}}}{\mu_{\hat{\tau}^{x}_{i,y}}}$. This expression is the same as the coefficient of variation (relative standard deviation) of the result of the Monte Carlo simulation.

According to the probabilistic nature of the problem, we cannot guarantee that the error will always be below ϵ. However, this can be done by relaxing the error constraint by introducing a confidence interval (CI) level. In particular, given the normal distribution of the error, the selected confidence interval can be correlated with the expected error:

$$P(error^{x}_{i,y} \leq \epsilon) \geq CI \implies err\hat{o}r^{x}_{i,y} \leq n(CI) \times \nu_{\hat{\tau}^{x}_{i,y}} \leq \epsilon \tag{7.3}$$

where $n(CI)$ is a value that express the confidence level (e.g. n(68%)=1, n(95%)=2 and n(99.7%)=3 derived from the 1-3 σ-intervals of the normal distribution). Thus, if we decrease the number of Monte Carlo simulations used to derive $\hat{\tau}^{x}_{i,y}$, on the one hand, we decrease the execution time of the application, but on the other hand we are also reducing the accuracy of the results, having a larger value for the coefficient of variation $\nu_{\hat{\tau}^{x}_{i,y}}$.

An additional problem is derived from the fact that $\hat{\tau}^{x}_{i,y}$ is input dependent. This means that it is not possible to predict the possible Monte Carlo error for unknown paths, according to the number of samples. To deal with this, we found a feature u_i of the inputs i that can be used to quickly estimate the number of samples necessary to keep the error below the threshold ϵ. The idea is to evaluate the error by using u_i instead of the actual i so that we can transform the original problem as

$$error^{x}_{i,y} \leq n(CI) \times \nu_{\hat{\tau}^{x}_{u_i,y}}. \tag{7.4}$$

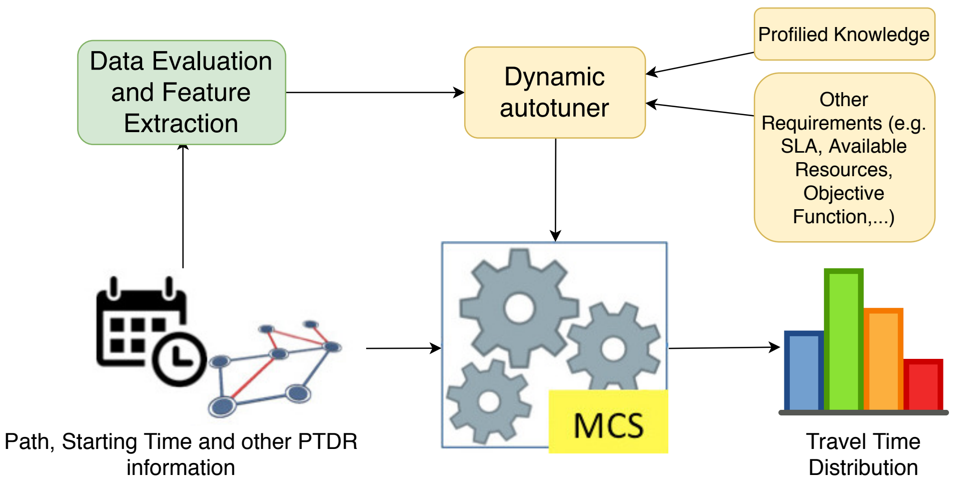

Data Evaluation and Feature Extraction
Dynamic autotuner
Profilied Knowledge
Other Requirements (e.g. SLA, Available Resources, Objective Function,...)
MCS
Path, Starting Time and other PTDR information
Travel Time Distribution

Figure 7.3: *The proposed adaptive approach for PTDR routing based on Monte Carlo simulations and dynamic choice of the number of simulations.*

The feature u_i has been called *unpredictability*, since it represents a set of characteristics of the inputs i (road, starting time,...) that provides information about how complex is the prediction of $\tau_{i,y}$, therefore it is also related on how many samples are required to satisfy a certain error and confidence level. More details on the unpredictability feature are presented in Section 7.4.1.

Given that the error is not anymore related to the specific input set i but only to the feature u_i, the number of samples needed to satisfy the constraint can be easily extracted by $\nu_{\hat{\tau}^x_{u_i,y}} \leq \frac{\epsilon}{n(CI)}$. A profiling phase on a set of representative inputs can be used to extract the values of $\hat{\nu}_{\hat{\tau}^x_{u_i,y}}$, that will be used to determine the correlation between the unpredictability function and the error. More details on the profiling phase including the prediction function are presented in Section 7.4.2.

To summarise, the proposed methodology adds an adaptivity layer on top of the Monte Carlo simulation (see Figure 7.3) to quickly determine at runtime the right number of samples for each request that satisfies the required accuracy. In particular, a feature-extraction procedure estimates the unpredictability value from the input data of the request (path, starting time and segment speed-profiles). The dynamic autotuner combines this data feature with the profiled knowledge and the extra-functional requirements to configure the Monte Carlo simulation.

7.4.1 Unpredictability Feature

Given that the extraction of the data feature from inputs should be done at runtime, its computation should not be a costly operation. Otherwise, the benefit of speeding up the computation phase by reducing the number of Monte Carlo samples would be reduced by the data feature extraction overhead, eventually making the whole approach meaningless.

From the experimental results, we have found that a measure of the unpredictability of the path can be extracted by a simple statistical property of the set of travel times θ extracted by a quick Monte Carlo simulation: the coefficient of variation. Intuitively the more the results are spread out, the more the route is hard to predict, thus to have a precise estimation of the distribution, and in particular the percentiles, we need a higher number of samples.

The unpredictability function is defined as $u_i = \sigma^x_{\theta_i}/\mu^x_{\theta_i}$ where σ_{θ_i} and μ_{θ_i} are evaluated on a MCS done with the minimum number x of samples allowed at runtime. It is important to note that σ_{θ_i} is the variance of the travel times extracted by a single Monte Carlo simulation on the minimum number of samples. We calculate the unpredictability function together with the first set of Monte Carlo samples to further reduce the overhead introduced by the data feature extraction. In particular, we will use this first short Monte Carlo run to determine if there is a need for further samples (and how many) to satisfy the error constraint.

To validate the usage of u instead of i, we performed the Spearman correlation test [133] between the unpredictability value and the value of $\nu_{\hat{T}^x_{i,y}}$ used in the calculation of the expected error for different values of x and y over a wide range of inputs sets i. In all cases, the correlation values were larger than 0.918 showing a p-value equal to 0. These correlations confirm our hypothesis, and the p-values prove that the results are statistically significant.

7.4.2 Error Prediction Function

To predict the expected error for a specific configuration according to the data feature u, we need to extract $\hat{\nu}_{\hat{T}^x_{u_i,y}}$ from profiling data. We run the Monte Carlo simulation several times for each configuration in terms of the number of samples. In particular, we decided to use values ranging from 100 samples up to 3000. The two numbers have been derived from the observation that 100 samples are the minimum to have the estimation of the percentile for the distribution, while 3000 is the number of samples that have already been found good enough to satisfy the worst case conditions

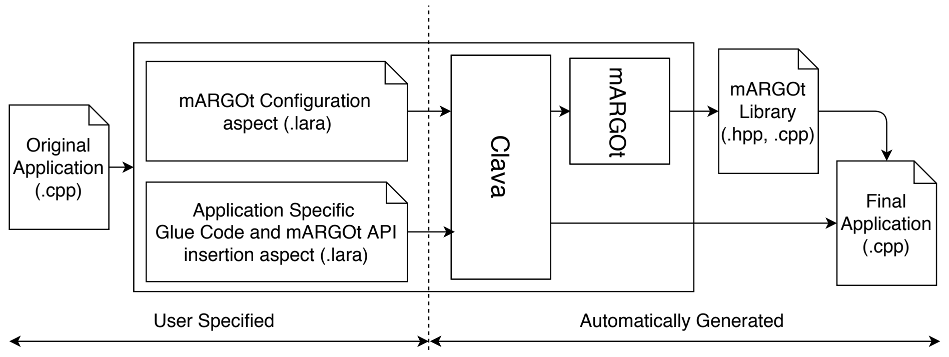

Original Application (.cpp)
mARGOt Configuration aspect (.lara)
Application Specific Glue Code and mARGOt API insertion aspect (.lara)
Clava
mARGOt
mARGOt Library (.hpp, .cpp)
Final Application (.cpp)
User Specified
Automatically Generated

Figure 7.4: *Integration flow outlining the two main LARA aspects and related actions: original code enrichment and autotuner configuration.*

on the previous work [134]. Between the two values, we selected 2 more sampling levels corresponding to 300 and 1000, that have been derived by considering that the Monte Carlo error decreases as $1/\sqrt{n}$ [135]. Thus in our case at each sampling level, we have that the error is almost halved.

We run each set of Monte Carlo simulations with the same configuration in terms of the number of samples on a large set of inputs i (i.e. roads, starting time ...), and we extract $\nu_{\hat{\tau}^x_{u_i,y}}$ and u_i from every single configuration. Then we create a predictor $\hat{\nu}_{\hat{\tau}^x_{u_i,y}}$ as the *quantile regression* [136] over the extracted data. The use of quantile regression enhances the robustness of the model in the context of its use. Indeed, we are not interested in predicting an average value as final result, but we want to use it for the inequality formula $\hat{\nu}_{\tau^x_{u_i,y}} \leq \frac{\epsilon}{n(CI)}$. In this case, a higher value of the quantile with respect to 50^{th} (the purely linear regression), guarantees higher robustness in satisfying the previous inequality. The quantile used for the regression is an additional parameter that can be explored to trade-off robustness and performance.

7.5 Integration Flow

While the previous section introduces the proposed methodology from the end-user perspective, thus considering execution time and elaboration error, this section focuses on the application developer perspective by presenting the integration flow proposed to enhance the target application with limited effort. The proposed integration flow enforces a separation between the functional and extra-functional concerns using an Aspect-Oriented Programming Language to inject the code needed to introduce the adaptivity

```
// Load data
Routing::MCSimulation mc(edgesPath, profilePath);
auto run_result = mc.RunMonteCarloSimulation(samples, startTime);
ResultStats stats(run_result);
Routing::Data::WriteResultSingle(run_result, outputFile);
return 0;
```

Listing 7.1: *Original source code before integrating the adaptivity layer.*

layer in the target source code.

On the one hand, we use *mARGOt* to dynamically tune the application, thus implementing the adaptivity concepts presented in Section 7.4 and thus transforming the target application as highlighted in Figure 7.3. In this context, we use *mARGOt* to select the number of samples that minimises the execution time, provided that they are enough to lead to an error below a certain threshold. In particular, the selection is made by considering the unpredictability value of the current path, and using as application knowledge the design-time model described in Section 7.6.1.

On the other hand, we hide all the complexity for code manipulation to the application developer by using LARA [16] as a language to describe user-defined strategies, and its Clava compiler [1] for source code analysis and transformation. LARA is a Domain Specific Language inspired by Aspect-Oriented Programming concepts. It allows a user to capture specific points in the code based on structural and semantic information, and then analyse and act on those points. This produces a new version of the application, leaving the original unchanged and separating the main functional concerns from those specified in LARA. Clava is a C/C++ source-to-source compiler based on the LARA framework. The compilation analyses and code transformations are described in scripts written in the LARA language.

In this work, we use Clava to perform two main tasks: first, to enrich the original source code with the required autotuner glue code, and second, to configure the autotuner library according to application requirements. Figure 7.4 depicts the transformation process, from the original source code to the final application and highlights the two main LARA aspects used. To further clarify the evolution of the application code and related aspects, Listing 7.1–4, present respectively the original source code, the two LARA aspects used to enhance the target application, and the final enriched code.

In particular, the code in Listing 7.2 shows the aspect needed to configure mARGOt, producing an autotuning library tailored according to the

[1]Project repository: `https://github.com/specs-feup/clava`

```
aspectdef McConfig
  /* Generated Code Structure*/
  output codegen end

  /* mARGOt configuration */
  var config = new MargotConfig();
  var travel = config.newBlock('ptdrMonteCarlo');

  /* knobs */
  ptdrMonteCarlo.addKnob('num_samples', 'samples', 'int');
  /* data features */
  ptdrMonteCarlo.addDataFeature('unpredictability', 'float',
      MargotValidity.GE);
  /* metrics */
  ptdrMonteCarlo.addMetric('error', 'float');
  /* goals */
  ptdrMonteCarlo.addMetricGoal('my_error_goal', MargotCFun.LE, 0.03,
      'error');

  /* optimization problem */
  var problem = ptdrMonteCarlo.newState('problem');
  problem.setStarting(true);
  problem.setMinimizeCombination(MargotCombination.LINEAR);
  problem.minimizeKnob('num_samples', 1.0);
  problem.subjectTo('my_error_goal', 1);

  /* creation of the mARGOT code generator for the following code
      enhancement (McCodegen aspect) */
  margoCodeGen_ptdrMonteCarlo = MargotCodeGen.fromConfig(config, '
      ptdrMonteCarlo');
end
```

Listing 7.2: *LARA aspect for configuring the mARGOt autotuner.*

application requirements. In lines 9–16, we define the *num_samples* tunable software knob, the *unpredictability* feature that we want to observe, the *error* metrics and the goal (i.e. the Service Level Agreement, $error < 3\%$) that in mARGOt is a condition that can be used later to define the optimization problem. Once the knobs, metrics, and data features have been defined, we can proceed with the creation of the multi-objective constrained optimisation problem that the autotuner has to manage (lines 18–23). In mARGOt optimisation problems are called states (line 19). It is because mARGOt gives the possibility to define multiple optimisation problems (only one can be the *default* one, line 22) and also to switch among them according to dynamic conditions. The constraints can be generated as in line 23, where the number represents the priority of the constraint. In case of more than one constraint, if the runtime is unable to satisfy both of them, it will relax the low priority one. Lines 21–22 define the objective function. Given that in this case, the objective is the minimisation of the number of samples, the aspect describes it as a linear combination (line 21) of the *num_samples* knob only by using a linear coefficient equal to 1 (line 22). mARGOt and LARA integration aspects permit to build different types of combined objective functions (e.g. linear or geometric combinations). Finally, line 26 builds the LARA internal structure *margoCodeGen_ptdrMonteCarlo* that is then used to create the mARGOt configuration file and code generator.

The second aspect (shown in Listing 7.3) aims at integrating the proposed methodology in the target application. It takes as input (line 3) the target function call that we want to tune, the mARGOt code generator produced by the previous aspect (Listing 7.2), and the number of samples needed to evaluate the unpredictability feature. In line 6, we query the code to identify the statement (*stmt*) including Monte Carlo function *call* as target join point to be manipulated. Lines 7–17 contain the actual manipulation actions done on the selected join point $stmt$ of the target code. It is composed of mainly two different types of operations. First, to integrate the mARGOt calls for initialising the library and updating the software knob (Lines 10 and 14). Second, to *insert* the glue code (LARA *codedef*) for calculating the unpredictability (line 12 and lines 21–25), and to *replace* the original Monte Carlo call with the optimised one that does not repeat the unpredictability samples (line 16 and lines 28–31).

Overall, in this specific instance of integration, we used 53 lines of LARA to generate 221 lines of C++ code. However, the advantage cannot be only considered from a numerical point of view (>4x in terms of the line of codes). There are three main reasons to justify this approach. First, the user does not need not to worry about the details of the mARGOt

```
aspectdef McCodegen
  /* Target function, mARGOt code generator from McConfig aspect, #
      samples for feature extraction */
  input targetName, margoCodeGen_ptdrMonteCarlo,
      unpredictabilitySamples end

  /* Target function call identification */
  select stmt.call{targetName} end
  apply
    /* Target Code Manipulation */
    /* Add mARGOt Init*/
    margoCodeGen_ptdrMonteCarlo.init($stmt);
    /* add unpredictability code */
    $stmt.insert before UnpredictabilityCode(unpredictabilitySamples)
        ;
    /* Add mARGOt Update */
    margoCodeGen_ptdrMonteCarlo.update($stmt);
    /* Add Optimized Call Code */
    $stmt.insert replace OptimizedCall(unpredictabilitySamples);
  end
end

/* Unpredictability extraction code */
codedef UnpredictabilityCode(unpredictabilitySamples) %{
  auto travel_times_feat_new = mc.RunMonteCarloSimulation([[
      unpredictabilitySamples]], startTime);
  ResultStats feat_stats(travel_times_feat_new, {});
  float unpredictability = feat_stats.variationCoeff;
}% end

/* Optimized MonteCarlo call */
codedef OptimizedCall(unpredictabilitySamples) %{
  auto run_result = mc.RunMonteCarloSimulation(samples - [[
      unpredictabilitySamples]], startTime);
  run_result.insert(run_result.end(), travel_times_feat_new.begin(),
      travel_times_feat_new.end());
}% end
```

Listing 7.3: *LARA aspect for inserting the application-specific glue code (unpredictability extraction) and the required mARGOt calls.*

```
// Load data
Routing::MCSimulation mc(edgesPath, profilePath);
auto travel_times_feat_new = mc.RunMonteCarloSimulation(100,
    startTime);
ResultStats feat_stats(travel_times_feat_new, {});
float unpredictability = feat_stats.variationCoeff;
if(margot::travel::update(samples, unpredictability)) {
  margot::travel::manager.configuration_applied();
}
auto run_result = mc.RunMonteCarloSimulation(samples - 100, startTime
    );
run_result.insert(run_result.end(), travel_times_feat_new.begin(),
    travel_times_feat_new.end());
ResultStats stats(run_result);
Routing::Data::WriteResultSingle(travel_times_new, outputFile);
return 0;
```

Listing 7.4: *Target source code after the integration of the adaptivity layer.*

configuration files and low-level C++ API, but can instead focus on the high-level interface available in LARA that results to be more declarative on the target problem (as shown in Listing 7.2 and Listing 7.3). Second, this approach reuses information between the integration's several steps. There is mARGOt-specific information that should be provided by the user in several places like the configuration files and when using the autotuning API (e.g. the name of the autotuner block and the knobs and data features). By using high-level LARA aspects, users only define this information once, saving time and possibly resulting in fewer production errors. Third, this approach leverages on a separation of concerns between the original code (functional description) and the autotuning code (extra-functional optimisation). All the extra-functional optimisations, including problem definition (optimisation targets and constraints), are kept separated, and users do not have to modify the original source. In this way, the original developer does not need to be involved with all the optimisation process and tools, thus permitting the functional development and extra-functional optimisation to run in parallel.

7.6 Experimental Results

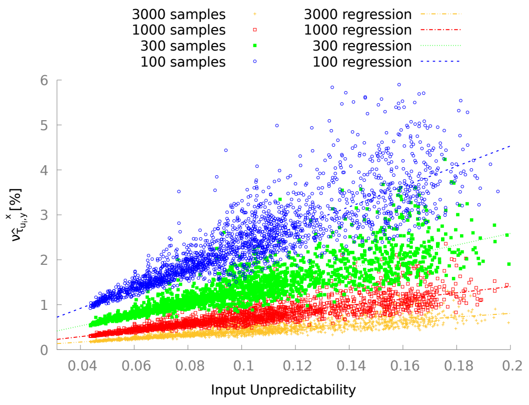
In this section, we show the results of applying the proposed methodology to the PTDR algorithm. The platform used for the experiments is composed by several nodes based on the Intel Xeon E5-2630 V3 CPUs (@2.8 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. First, we show the results of the model training for estimating the expected error (see Section 7.6.1). Then, in Section 7.6.2 we validate the approach by verifying the respect of the error constraint ϵ. We compare the proposed approach with respect to the original version that takes a static decision on the number of samples (see Section 7.6.3). Finally, in Section 7.6.4, we discuss the overhead introduced, while in Section 7.6.5 we evaluate the optimisation impact when considering the entire navigation service at system-level.

7.6.1 Training the Model

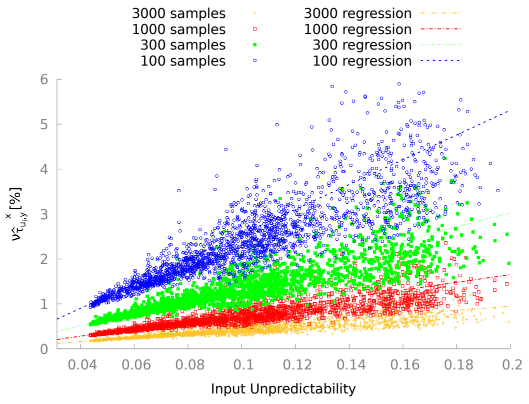
The first actual phase of the methodology is done off-line, and it consists of training the error model ($\hat{error}^{x}_{i,y}$) presented in Section 7.4.2 by using a different number of samples. For training the quantile regression, we used profiling data extracted by running the PTDR algorithm on a training set. This training data set has been built using random requests done on 300 different paths across the Czech Republic in different time-slots, thus considering different speed-profiles for each segment of the paths. All these requests have been made for all the four levels of sampling used in this chapter (i.e. 100, 300, 1000 and 3000, as described in Section 7.4.2). The output of the model training is represented in Figure 7.5. The points in the three plots represent the results obtained from the profiling runs. The lines represent the quantile regression lines, thus the model that will be used at runtime. The three sub-figures are different in terms of the quantile value used for the regression. Figure 7.5a represents the 50th percentile, Figure 7.5b represents the 75th percentile, while Figure 7.5c represents the 95th percentile.

We can see that the three regressions are slightly different since we pass from a more permissive one in Figure 7.5a, where almost half of the points are below the corresponding regression lines, to the most conservative one in Figure 7.5c where only a few points are above. Analysing in depth the data, we can see that the coefficients of the lines of the quantile regression are almost doubled passing from 75th to 95th percentile (e.g. for 100 samples, the coefficients pass from 0.27 to 0.38, while for 3000 samples they pass from 0.049 to 0.071).

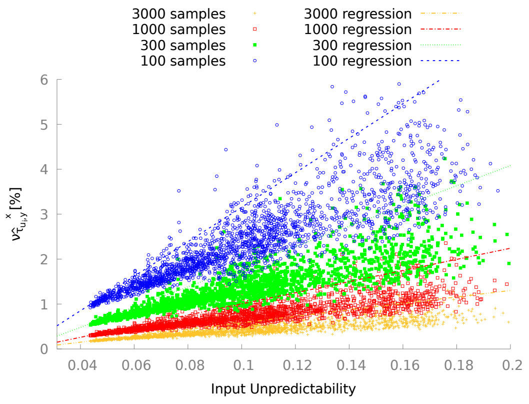
The extracted models are now ready to be used at run-time by the dynamic autotuner to select the minimum number of samples that satisfy the error constraint for the given unpredictability value. The results shown in Section 7.6.2 will demonstrate the effectiveness of the proposed method at runtime.



(a) *Quantile regression using the 50th perc.*

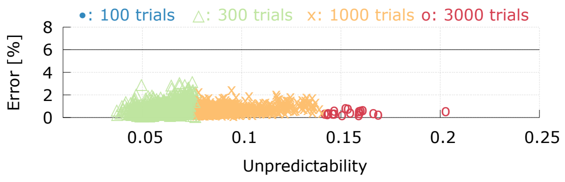


(b) *Quantile regression using the 75th perc.*

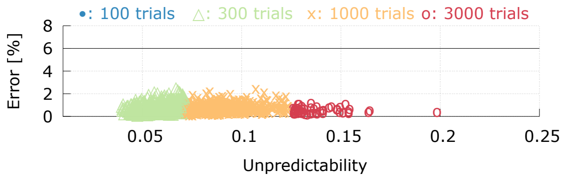


(c) *Quantile regression using the 95th perc.*

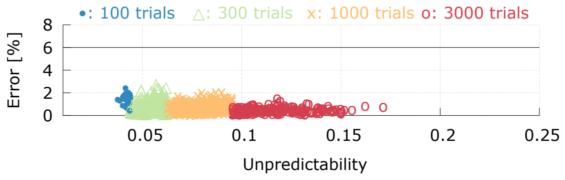
Figure 7.5: *Training of the error model by using different number of samples and quantile regressions.*



(a) *Quantile regression using the 50th perc.*



(b) *Quantile regression using the 75th perc.*



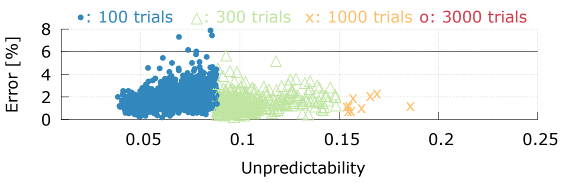
(c) *Quantile regression using the 95th perc.*

Figure 7.6: *Validation of the proposed approach by using 3% as target error and different percentiles for the quantile regression.*

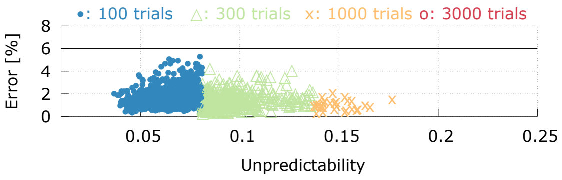
7.6.2 Validation Results

The set of validation results presented in this section are reported to demonstrate how the dynamic tuning of the number of samples satisfies the error constraints. For doing this, we randomly generated 1500 requests to the enhanced PTDR module for routes on the Czech Republic at different starting times. These requests are different from the ones used in the training phase of the model. We validate the approach by using three different quantile regressions (on 50^{th}, 75^{th} and 95^{th} quantile), two different target errors ϵ (3% and 6%) and a confidence interval (CI) for the error constraint equal to 99% (i.e. $n(99\%) = 3$). The error has been derived by considering a run of the Monte Carlo simulation on the same input set by using 1 million of samples, thus enough to be considered a good estimation of the actual travel time distribution. Then, we selected as error the maximum between different key percentiles: 5^{th}, 10^{th}, 25^{th}, 50^{th}, 75^{th}, 90^{th} and 95^{th} percentile.

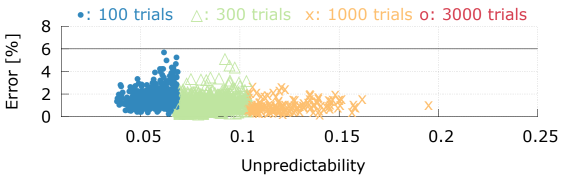
The results are reported in Figure 7.6 and Figure 7.7 respectively for an error constraint ϵ equal to 3% and 6%. The two figures show the error results for each run with respect to the unpredictability feature extracted on the path. Each dot in the plots represents a PTDR request, while its



(a) *Quantile regression using the 50th perc.*



(b) *Quantile regression using the 75th perc.*



(c) *Quantile regression using the 95th perc.*

Figure 7.7: *Validation of the proposed approach by using 6% as target error and different percentiles for the quantile regression.*

shape depends on the number of samples used for the Monte Carlo simulation. In most of the cases, the actual error is below the target error. As it was expected considering the same value of the error constraint ϵ, the more conservative is the quantile regression, the less are the points that violate the constraint error. For the data, we processed, in all cases the number of times the error constraint is not respected is within the selected CI (99%). At the same time, moving from a less conservative quantile regression (e.g. 50^{th} percentile) towards a more conservative one (e.g. 95^{th} percentile), it is possible to note how the threshold values for selecting the same number of samples shifts to the left. As an example, by considering an error constraint $\epsilon = 3\%$ (see Figure 7.6), the maximum unpredictability value for having 300 samples moves from 0.075 to less than 0.06 respectively when considering the quantile regression from the 50^{th} percentile, up to the 95^{th} quantile. Similar is the case when we consider an error constraint $\epsilon = 6\%$ (see Figure 7.7), where the same threshold moves from an unpredictability of 0.15 to 0.14 and 0.11 when using the 50^{th}, the 75^{th} and the 95^{th} as quantile value for the regression. Finally, it is visible the difference in terms of the number of samples between the two cases with different ϵ values. Indeed, while for ϵ equal to 3% (Figure 7.6) only a tiny fraction of the cases use 100 samples and there are a not negligible fraction of cases where 3000 samples are employed. For ϵ equal to 6% (Figure 7.7) in some cases only 100 samples are required.

7.6.3 Comparative Results with Static Approach

In this subsection, we demonstrate the advantages obtained by using the proposed approach with respect to the baseline version [82] where the number of samples is defined *a priori*. To provide a fair comparison, we extracted the number of samples to be used for the baseline version by using the training dataset.

For the 4 levels of sampling used in this chapter (i.e. 100, 300, 1000 and 3000, as described in Section 7.4.2), we analyzed the cumulative distributions of the expected error (see Fgiure 7.8). We selected the minimum sampling level that passes a certain threshold of the cumulative value before reaching the error constraint value ϵ. This threshold value has almost the same *robustness* meaning of the quantile regression value used in our approach. In the following, we are going to compare the proposed approach where the quantile regression model has been built over a certain percentile, with a static tuned version where the same percentile has been used as the threshold for the cumulative. If we use for the proposed approach the quantile regression at 95%, we compare with the statically tuned version where the number of samples has been defined looking at the cumulative curve that reaches at least 95% before to the target error constraint. In particular looking at Figure 7.8, we can notice that for an error constraint $\epsilon = 6\%$ the static tuning is set to 1000 samples for the entire percentile interval between 72^{th} and 98^{th}, while for values larger than 98^{th} and smaller than 72^{th} percentile (down to 7^{th}) we have to consider the configuration using 3000 and 300 samples respectively. On the other side for $\epsilon = 3\%$ we select 3000 samples within the percentile interval 72^{th}-97^{th}, 1000 samples for percentile values smaller than 72^{th} (down to 5^{th}), while we need more than 3000 samples if the request is very tight on a percentile larger than 97^{th}.

Table 7.1 shows the comparative results obtained by using the proposed adaptive technique with respect to the original version (*baseline*) with the statically defined number of samples obtained with the previously described analysis. In particular, Table 7.1 presents the average number of samples and gain with respect to the baseline for different values of error constraint ϵ and different percentiles used to build the predictive model and for the static tuning of the baseline. The results are obtained by running a large exper-

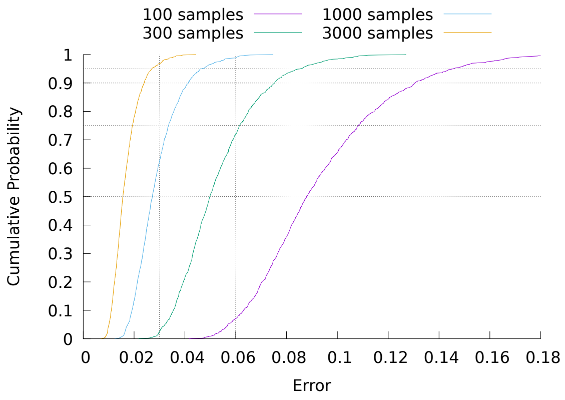


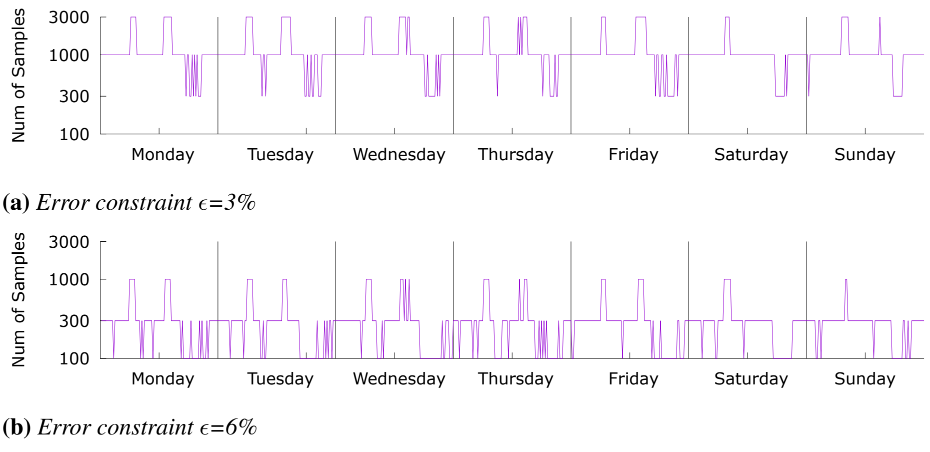
Figure 7.8: *Cumulative distribution of the error by using different numbers of samples over the training set.*

imental campaign over randomly selected pairs of Czech Republic routes and starting times, different from those used for the training. While the routes have been randomly selected, we used a more realistic distribution of the starting time [137] [138].

In all the considered cases, the proposed approach reduces the number of samples by at least 36% and up to 81%. As expected, the average number of samples for the proposed approach is lower when we relaxed either the error constraint (i.e. 6%) or the percentile used for building the model (e.g. 50th percentile). The lower gain for the configurations using the 50th percentile with respect to the cases using the 75th-95th percentile is due to the fact that in the former case the baseline requires a lower number of samples with respect to the latter cases (i.e. 1000 vs 3000 for $\epsilon = 3\%$ and 300 vs 1000 for $\epsilon = 6\%$). When we focus on the absolute numbers, it is possible to detect that even if the percentage gain seems higher with more conservative regressions (75th-95th), the actual average number of samples

Table 7.1: *Average number of samples for the validation set using different quantile regression values (columns) and different error constraints. The results are reported for the* baseline *and proposed* adaptive *versions.*

		Average Number of Samples		
ϵ		50th perc.	75th perc.	95th perc.
3%	baseline	1000	3000	3000
	adaptive	632 (-36%)	754 (-74%)	1131 (-62%)
6%	baseline	300	1000	1000
	adaptive	153 (-49%)	186 (-81%)	283 (-71%)



(a) *Error constraint ϵ=3%*

(b) *Error constraint ϵ=6%*

Figure 7.9: *Number of samples selected by the proposed adaptive method when the same request is performed every 15 minutes during the entire week.*

used is smaller with the more permissive quantile (50^{th}).

The reduction in terms of the number of samples is directly reflected in the execution time reduction since there is a linear dependency, except for the overhead introduced by the dynamic autotuner. In particular, we observed an execution time speed-up between 1.5x and 5.1x. A more detailed analysis of the overhead is presented in Section 7.6.4.

To further show the benefits of the proposed methodology, Figure 7.9 shows the number of samples selected by the adaptive Monte Carlo simulation when the same request in terms of target path is performed every 15 minutes during the entire week. The used temporal interval is derived by the smaller time granularity (ϕ) we had for the database containing the speed profiles. The two plots (a) and (b) have been generated using respectively 3% and 6% as maximum target errors and for both experiments a quantile regression on the 75^{th} percentile.

By looking at the number of samples requested by the adaptive version of the Monte Carlo simulation, we can easily recognise well-known traffic behaviours in both plots. The daily distribution on the weekdays is characterised by two main peaks determined by less predictable situations. The first around 7–8 am and the second around 4–5 pm. During the weekend the morning peak seems to be a bit postponed, while the afternoon one almost disappears. On the opposite, it is also visible how the evening hours result to be the most predictable ones.

This dynamic behaviour that is captured by the enhanced version of the algorithm cannot be exploited by using the original (baseline) version. Following the same philosophy adopted in Figure 7.1, the original version must be tuned by considering 3000 samples for the experiment in Figure 7.9(a) ($\epsilon = 3\%$) and 1000 samples for the experiment in Figure 7.9(b). In both cases, the static tuning results to be the larger number of samples selected from the proposed techniques, that instead is able to use it only when it is strictly required (e.g. during the traffic peaks). Also considering the static tuning to the *average* case (i.e. 1000 samples for the experiment in Figure 7.9(a) ($\epsilon = 3\%$) and 300 samples for the experiment in Figure 7.9(b)) is not a viable solution. This is because there are still many sampling reduction possibilities in predictable moments that will not be captured, and more important, the prediction will not be able to satisfy the algorithm output quality during the most unpredictable periods. Finally, a fixed time-slot policy is also sub-optimal since the unpredictability strongly depends not only on the time of the request but also on the path characteristics (e.g. urban or countryside path, close or far from congested areas) and length (e.g. when it is expected the arrival in a congested area).

7.6.4 Overhead Analysis

While we widely describe in Section 7.5 how we reduced the integration overhead from the application developer point of view, this section clarifies the time-overhead introduced to obtain the proposed adaptivity. In particular, the additional computations that we add are related to the calculation of the $\nu_{\hat{T}^x_{i,y}}$ and to the autotuner calls used to determine the right number of samples to be used. The initial 100 Monte Carlo samples, required to extract the data feature, are not part of the overhead given that they are reused (and thus discounted) to calculate the expected travel time (see Listing 7.4).

Figure 7.10 shows the overhead introduced by the proposed methodology compared to a set of Monte Carlo calculation by using a different number of samples (from 100 to 300, and 1M) over a set of paths among different locations in the three main cities in the Czech Republic. As expected, it is evident that the execution time is strictly correlated to the number of samples used for the travel time computation. The different paths we used are in a range between 300 and 800 segments long. When we fix the number of samples, the different number of segments is the main reason for the variability of the Monte Carlo simulation computing time.

Although the proposed methodology introduces an overhead for every request, it is almost negligible, i.e. more than two orders of magnitude less than the smaller Monte Carlo simulation with 100 samples. In particular, we found that the execution time of the data feature extraction and mAR-

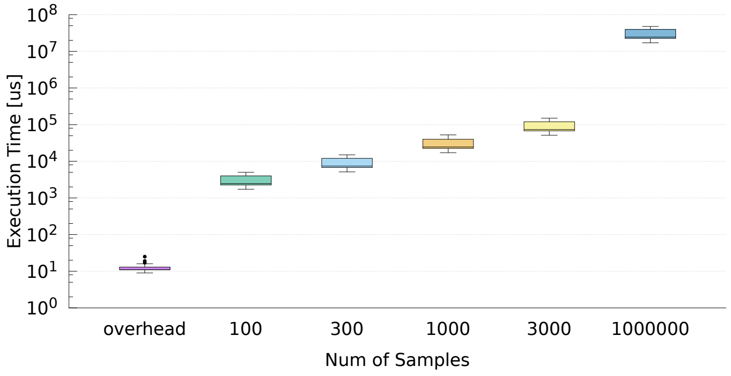


Figure 7.10: *Evaluation of the execution time overhead due to the additional code for the proposed method with respect to the target Monte Carlo simulation by varying the number of samples.*

GOt calls is comparable to the evaluation of a single sample of the Monte Carlo on a road composed of 200 segments.

7.6.5 System-Level Performance Evaluation

To quantify at system-level the effects of the proposed adaptive method, in this section we present an analysis to evaluates the efficiency of the PTDR module when it is included in the full navigation pipeline shown in Figure 7.1. We built a performance model of the navigation pipeline by using the simulation environment Java Modeling Tools (JMT) [139]. JMT is an integrated environment for workload characterization and performance evaluation based on queuing models [140]. It can be used for capacity planning model simulation, workload characterization and automatic identification of bottlenecks. In particular, to build the simulation model of the queuing network, we considered one station for each of the modules that compose the navigation pipeline, and we added a fork-join unit to model the parallel PTDR evaluations of each alternative path found in the first stage.

The model, shown in Figure 7.11, has been annotated with values derived by the profiling of each module (K-Alternative path, PTDR, and reordering) and considering a value for K (the number of alternative paths to evaluate) equal to 10. Moreover, we made a resource allocation according to a load produced by up to 100K cars producing a request every 2 minutes. The latter number is in line with the consideration of having self-driving cars continuously connected with route planner, while the former has been derived by a simple estimation considering a Smart City such as the Milan

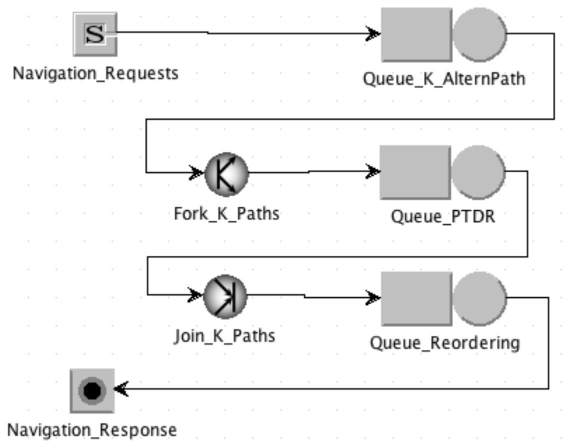


Figure 7.11: *Complete navigation pipeline modeled using JMT.*

urban area. Indeed, in this area the population is composed of around 4 Million people, every day it is estimated to have more than 5 Million trips, and only less than 50% are done by using public transportation [141, 142].

Under these conditions and considering the configuration with $\epsilon = 6\%$ and $95^{th} percentile$, we found out that by adopting the proposed technique we have a 36% reduction in terms of the number of resources needed to satisfy the target workload. In particular, we can differentiate 2 cases. The first one considering the number of resources needed to satisfy the steady-state conditions, and thus that the throughput in terms of input requests should be satisfied by all the stages. In this case, without the proposed optimisation we would have needed at least 777 computing resources (cores). Among them, 400 cores (52% of the entire set) should be dedicated to PTDR. By applying the proposed technique, only 497 cores are needed, reducing to 120 (24% of the entire set) those required for the PTDR stage. The second case considers a more dynamic environment where it is suggested to keep the average utilisation rate of each station below 70%. While respecting this rule of thumb [143], the distribution of the system response time (the time passing from the navigation request to the response) results to be narrow, thus being able to react better to burst of requests. In this second case, without the proposed optimisation we would have needed 1010 cores to allocate the entire pipeline. 572 of them (57% of the entire set) should be dedicated to the PTDR stage. By applying the proposed technique, 646 cores are enough to allocate the pipeline, and out of them, only 172 (26% of the entire set) are dedicated to the PTDR.

7.7 Summary

In this chapter, we focused on Probabilistic Time-Dependent Routing to show how it is possible to improve computation efficiency, by using *mARGOt*. The proposed method quickly samples the input data to extract an unpredictability feature used to determine the number of simulation proactively, while satisfying a certain error threshold. The runtime decision is based on a probabilistic error model – learned offline – correlating the unpredictability feature extracted from the data and the number of samples used by the Monte Carlo algorithm. Experimental results demonstrated that the proposed adaptive approach for the PTDR problem could save a large fraction of simulations (between 36% and 81%) with respect to a static approach while considering different traffic situations, paths and error requirements. Considering the entire navigation pipeline, also composed of the k-alternative path and reordering stages, the adoption of the proposed technique guarantees a significative reduction in terms of computing resources. Finally, we adopted an aspect-oriented programming language (LARA) to reduce the effort necessary on the application developer for introducing the code needed to improve the execution efficiency. Even if we can completely remove the integration effort from the source code point of view, the application developer is still required to interact with LARA. However, by using this approach, it is possible to enforce separation of concerns between functional and extra-functional requirements.

CHAPTER 8

Tuning a Molecular Docking application

In this chapter we use *mARGOt* to autotune a typical HPC application which has a constraint on the time-to-solution, in the context of a drug discovery process. We introduced this application in Section 4.4.2. This chapter analyses the algorithm to identify software-knobs that can expose throughput-quality tradeoffs. Then, we show how it is possible to learn the relation between the exposed tradeoffs and features of the current input. At runtime, *mARGOt* leverages the application knowledge to maximise the accuracy of the current input elaboration within the given time-to-solution.

8.1 Introduction

The goal of a drug discovery process is to find novel drugs starting from a huge exploration space of possible molecules. Typically, this process involves several *in vivo*, *in vitro* and *in silico* tasks ranging from chemical design to toxicity analysis. Molecular docking is one stage of this process [144, 145]. It aims at estimating the three-dimensional pose of a given molecule, named *ligand* when it interacts with the target protein. The ligand is much smaller than the target protein; therefore we focus a small region of the target protein (or receptor), named *pocket* (or binding site). Given the three-dimensional pose of the ligand within the pocket, we can estimate the strength of the chemical and physical interactions between the ligand and the pocket by computing a geometric fitting score.

The evaluation of the pose of each ligand is independent of the evaluation of all the other candidates. Given that in drug discovery the number of ligands that we are interested in analysing is above the billion units, we can consider this problem embarrassingly parallel. However, to find the three-dimensional pose of the ligand when it interacts with the pocket, we have to deal with a large number of degrees of freedom. While we might represent the target pocket as a rigid structure, the ligand is a flexible set of atoms bound together by chemical bonds, i.e. sharing of electron pairs between atoms (covalent bond). From a purely geometrical point of view, it is possible to identify a subset of bonds – named *rotamers* – which can split the ligand into two disjoint non-empty fragments when we remove them. We can independently rotate each of those fragments without altering the chemical connectivity of the ligand. Therefore, we have to consider changes in the shape of the ligand that can be obtained through the rotation of all its rotatable fragments.

As evaluating the chemical and physical interactions between the ligand and the pocket is a computationally intensive problem, state-of-the-art approaches [146–148] suggest splitting the pose prediction task from the virtual screening task. The pose prediction task focuses on providing the best pose for a given ligand within a given binding site, whereas the virtual screening task aims at selecting among a huge database of candidates a small set of promising ligands which best fit the given binding site. The structure of the two tasks is very similar to each other. Indeed, several industrial applications [87, 149] provide both functionalities within the same software module. A remarkable difference between the pose prediction and the virtual screening task lies in the approach to the estimation of the chemical and physical interactions between the ligand and the pocket. It is possible to estimate such interactions with either a geometrical or a pharmacophoric approach. The geometrical approach estimates the ligand-pocket interactions by only using the shape and volume information, while the pharmacophoric approach evaluates the actual chemical and physical interactions.

The latter approach is the most computational-intensive one, and it is regularly exploited on the pose prediction task. Although the best solution according to a pharmacophoric approach also has a very good geometrical score, the best geometrical solution does not guarantee to be either a valid solution or a good solution from a pharmacophoric perspective. Therefore, there is always the need to apply the pharmacophoric approach. The geometrical approach can be exploited for virtual screening before the pharmacophoric evaluation. The scope of this chapter is limited to the geometrical approach for virtual screening.

During the virtual screening process, the time budget is one of the constraints that a molecular docking application has to meet. Nowadays it is common practice to have a domain-expert human in the loop, whose job is to tune the size of the ligands database according to the available time budget. This approach limits the exploration space without any guarantee neither to find a global optimum nor to find a good local optimum. In order to increase the chances to find an interesting solution, we need to enlarge the ligands' input set. Therefore, a reduction in the time spent on evaluating a single pair ligand-pocket enables the end-user to explore a larger set of candidates.

In this chapter, we focus on *GeoDock-MA*, a molecular docking Mini-App for High-Performance Computing (HPC) systems based on the LiGen-Dock module [87]. In general, Mini-Apps can be an important aid for computing architecture and algorithm design space explorations in the early stages of code development. *GeoDock-MA* attempts to capture key computation kernels of the molecular docking application for drug discovery implemented in LiGenDock. By developing *GeoDock-MA* in parallel with the new version of LiGenDock, application developers can work with system architects and domain experts to evaluate alternative algorithms that can either better satisfy the end-user constraints, or better exploit the architectural features. *GeoDock-MA* allows faster performance analysis and optimisation of the key kernels.

The main goal of the proposed approach is to enable tunable approximations to explore performance-accuracy trade-offs during the docking phase. In this chapter, we enhance *GeoDock-MA* with software knobs, and we use them to control time-to-solution in the virtual screening task. In particular,

- the *GeoDock-MA* has been analysed to properly introduce approximate computing techniques on the most significant kernels;
- a performance/accuracy trade-offs have been enabled by exposing tunable software knobs to drive *GeoDock-MA* approximations;
- a *GeoDock-MA* performance model based on the exposed software knobs and input data features has been presented for estimating time-to-solution in a virtual-screening process;
- a *GeoDock-MA* has been enhanced with an autotuning layer capable

of satisfying user-defined time budget according to the workload characteristics.

8.2 Background

Molecular docking is a well-known research topic that is addressed in literature from different perspectives. A large share of work in this field approaches the problem by exploiting random-based algorithms, such as genetic algorithms [150, 151] or Monte Carlo simulations [149, 152]. However, a desirable feature of a molecular docking application is the determinism of the solution. Since the tasks following the *in-silico* step require expensive solution-dependent resources, for several companies having a deterministic and repeatable result is a constraint.

Early work in the literature, such as [153], produce deterministic solutions. However, they consider only rigid movements of the ligand during the docking procedure. Real case scenarios usually require the rotation of portions of the ligand molecule. Therefore, the limitation of rigid movements is likely to prevent the applicability of the solution in the industry. The work by Palma et al. [154] overcomes this issue: they introduce a molecular docking framework which can deal with the flexibility of the ligand molecule. It adopts a model of the electrostatic interactions to finalise the docking. Similar works such as DOCK [155], FlexX [156], FlexX-Scan [157] and sur-flex [158] provide deterministic molecular docking of flexible ligands. They allow the user to exploit several docking algorithms according to the specific use case. All these algorithms also rely on both geometric and pharmacophoric properties in their docking algorithms. All these works implement a different docking procedure with respect to LiGenDock; however, no one has been designed to expose software knobs to enable possible quality-performance trade-offs explicitly. The proposed approach behind *GeoDock-MA* unlocks the possibility to tune the docking procedure according to high-level constraints, such as the allocated time budget for a given size of the ligand database to be virtually screened.

Algorithm-level approximate computing techniques are well known in the literature [159]. In this work, we exploit grid-based optimisations on the geometrical docking kernel. In particular, in computational physics, it is very common to exploit models based on multi-level grids to achieve a fine-grained solution in a restricted area of the whole simulated environment. The size of the grid is a parameter which allows the physicists to trade-off granularity of solution for the increasing/decreasing number of elements to be processed. Geophysics applications exploited for a long time nested grids, such as thermosphere models [160–162] and ocean flows models [163]. The evolution of nested-grids models made the researchers abandon regular grids in favour of variable-size grids. Irregular grids have been exploited in climate forecast models to improve the performance of grid-based models. Authors of [164] demonstrate that a variable-resolution stretched grids lead to longer-term climate forecast with the same accuracy of the nested grid models. In physics, variable-size grids are used to discretise geophysical problems such as advection equations [165].

In the field of image rendering, grid processing has been optimised by selecting which tiles need to be processed first and which later or do not require processing at all. An element is peeled from each tile, and its value is used to decide how to compute the corresponding tile. Depth peeling [166] is a technique which allows splitting an image into several layers. GPUs can render images layer by layer, starting from the closest layer to the viewer. Authors of [167] apply depth peeling to focus computation only on the interesting tiles and selectively skip useless tiles. Several visualisation applications and physics simulations exploited this technique, such as [168].

At compiler-level, GPU-oriented selective skip of instructions has been performed in [169]. Authors implemented approximate-computing techniques via compiler transformations to be applied before the execution of CUDA kernels. Their approach requires to generate in advance a set of kernels with different approximation levels and to switch between them at runtime according to the measured error. In this chapter, we exploit a similar technique on MPI kernels.

8.3 Methodology

This section first introduces *GeoDock-MA*. In particular, we describe its algorithm, and we highlight the application hot spots. Then, we perform a functional analysis to drive the approximation of the elaboration, enabling the accuracy-throughput trade-off. Finally, we describe the exploitation of the trade-off for the application auto-tuning subject to time-to-solution constraints and workload characteristic.

8.3.1 Application Description

In the context of LiGen toolflow [86], LiGenDock [87] is the module that aims at docking one or more ligands into a target protein. It can be used to perform both the pose prediction and the virtual screening tasks. It exploits chemical and geometrical features to dock the ligand through an iterative algorithm. In particular, LiGenDock uses chemical features to set the initial pose of the ligand and to drive the docking process between each iteration. However, it also uses geometrical features to optimise the pose of the ligand in each iteration, by taking into account all the degrees of freedom of the problem space.

The optimisation of the ligand pose is the most computationally intensive part of LiGenDock during the virtual screening task. *GeoDock-MA* includes all the functionalities of LiGenDock that optimise the ligand pose by exploiting the geometric approach. *GeoDock-MA* takes as input the target pocket and a database of ligands, and it produces as output, the score of each pocket-ligand pair, after optimising the pose of the ligand.

GeoDock-MA performs the virtual screening task by using the geometric approach. It estimates the pocket-ligand interactions with the similarity between the shape of the ligand and the three-dimensional shape of the pocket in PASS format [170], which is produced by LiGen PASS [86]. Actually, *GeoDock-MA* scores each ligand with the *overlap score* function. The overlap score, as defined in Equation 8.1, is the reciprocal of the minimum square distance between the ligand and the pocket:

$$o = \frac{l}{\sum_{i=0}^{l} \min_{j=0}^{p} d^2(L[i], P[j])} \tag{8.1}$$

where o is the overlap score, l is the number of atoms in the ligand L, p is the number of 3D points in the pocket P, and d^2 represents the squared distance between the i-th atom of the ligand and the j-th point of the pocket. Hence, higher overlap means better geometric compatibility between pocket and ligand.

Figure 8.1 shows an example of docked a ligand inside a pocket (i.e. 1cvu [171]). The ligand structure is visible in the 3D image, and the bottom left corner highlights its planar representation. Larger bubbles are the atoms L of the ligand while the connections between atoms are the molecule bonds. The dark spots in the figure are the points P representing the PASS version of the target pocket. These points are the centre of the spheres used to model the binding site.

8.3.2 Analysis of *GeoDock-MA*

GeoDock-MA is designed to target an HPC platform. It exploits the machine-level parallelism through the MPI master/slave paradigm. In particular, the master process reads the ligands' input database and dispatch those ligands

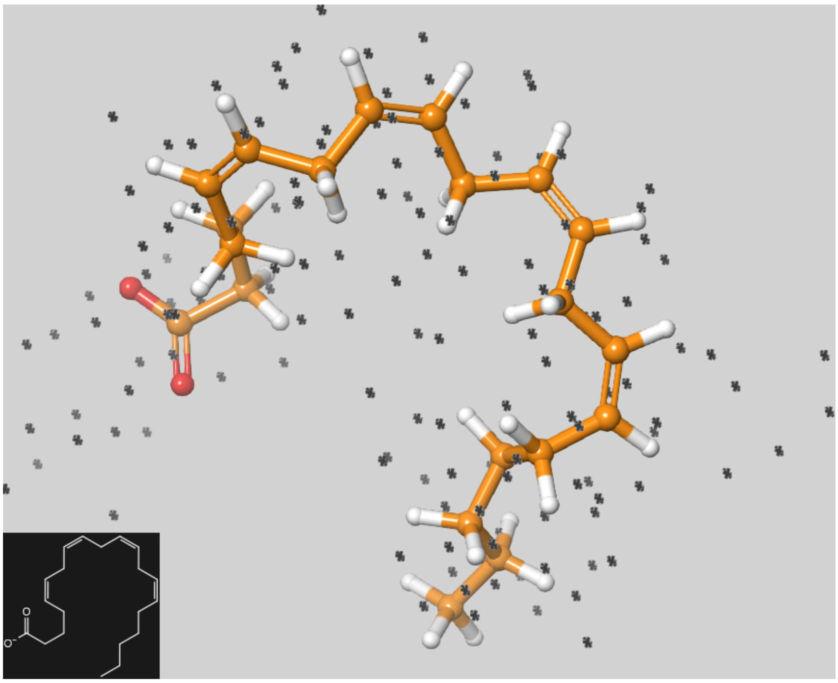


Figure 8.1: *3D visualization of a docked ligand (connected structure) inside a PASS version of the target pocket (dark spots).*

to any free slave. Each slave computes the overlap score of a given ligand with respect to the pocket of the target molecule. At the end of the computation, each slave process notifies the master about the best overlap score found and waits for new data to be processed. *GeoDock-MA*, as well as the original LiGenDock, avoids the parallelism at the level of each slave task as it falls under the embarrassing parallel class. Indeed, given that the huge number of ligands to be processed, the parallelism is widely exploited at a higher level.

We profile the application in order to understand which are the critical sections of the code by using GPROF[1]. Figure 8.2 shows the Call Graph report. It groups the individual functions by the caller.

The application spends a significant fraction of the execution time on `MatchProbesShape`. This kernel is responsible for the optimisation of the shape of the ligand, using a steepest descent algorithm to deal with all the internal degrees of freedom of the ligand. In this chapter, we focus on the introduction of possible software knobs through approximation techniques to tune the time-to-solution of this functionality.

[1]GNU gprof `https://sourceware.org/binutils/docs/gprof/`

```
99.9% - MPISlaveTask
 └─ 98.7% - Molecule::MatchProbesShape
     ├─ 89.2% - Molecule::MeasureOverlap
     │
     └─ 08.2% - Fragment::CheckBumps
```

Figure 8.2: *Application Call Graph profile. Functions taking less than 2% of the overall execution time are omitted.*

Algorithm 3 shows the pseudo code of the target kernel. At first, the algorithm identifies the set of *rotamers* (line 1), thus selecting all the possible sources of flexibility in the ligand shape. Then, it iterates over the set of these bonds to find the best shape of the ligand (lines from 2 to 20). In particular, the body of the algorithm grows a left and right ligand fragments, with respect the two extremes of the bond (line 3). The left and the right fragments are rotated independently. The first to be processed is the left fragment. It is rotated step by step up to a 360 degree angle (lines from 4 to 5); At each step, we check whether the ligand shape is valid. There is a non-null possibility of internal bumping of the molecule (line 6), which invalidates the shape of the ligand. If the ligand shape is valid, the overlap score of the ligand is considered during the check for possible improvements (lines from 7 to 9). At the end of the whole 360 degrees of exploration, we rotate once more the left fragment to match the angle that maximises the overlap score (line 11). The same procedure is applied to the right fragment (lines from 12 to 19).

The kernel applies the same computation of the left and right fragment of each rotamer. For this reason, in the rest of the chapter, we do not differentiate between the two fragments.

Figure 8.2 shows also that the computation of the overlap score of each pose (`Molecule::MeasureOverlap`) is the most expensive operation. The implementation of `Molecule::MeasureOverlap` is relatively simple, and its optimisation is not trivial given that much effort has been spent in the past in terms of performance tuning. Our contribution aims at reducing the number of invocations performed by its caller. In particular, we want to avoid the computations which are very likely to do not lead to any improvement.

Figure 8.3 shows an example of how the rotation of a fragment affects the overlap score of the ligand. The x-axis represents the rotation space, while the blue line shows the overlap score of the ligand according to the position of the fragment. The empty spaces are due to the fact that some

Algorithm 3: Pseudo-code of the `MatchProbesShape` kernel, which changes the shape of the ligand to maximize the overlap score.

```
Data: the pocket and the 3D structure of the ligand
Result: the overlap score of the ligand
get the list of rotamers;
foreach rotamer do
    grow the right and left fragment;
    for angle in 0-360 degrees with step 1 degree do
        rotate left fragment to angle;
        if the ligand shape is feasible then
            measure the overlap of the ligand;
            check if the overlap is improved
        end
    end
    set the left fragment to best angle found;
    for angle in 0-360 degrees with step 1 degree do
        rotate right fragment to angle;
        if the ligand shape is feasible then
            measure the overlap of the ligand;
            check if the overlap is improved
        end
    end
    set the right fragment to best angle found;
end
return the overlap score of the ligand;
```

of the generated poses of the ligands are not valid because of the internal bumps of the ligand atoms. We define as *delta overlap* the difference between the minimum and maximum overlap of a single rotation. We define as *peak* the set of contiguous and valid rotation angles whose overlap is higher than 50% of the delta.

We analyse the behaviour of the application to find exploitable patterns for reducing the number of evaluations for each fragment rotation, thus creating possible application knobs (see Figure 8.4).

Figure 8.4a correlates the size of a fragment with its impact on the final score of the ligand. In particular, the x-axis represents the relative size of the fragment with respect to the size of the ligand. The y-axis represents the delta overlap normalised with respect to the final score of the ligand. We can notice that small fragments have small deltas, which means that such fragments usually have a limited impact on the numeric value of the final score of the ligand.

Figure 8.4b correlates the width of a *peak* (in degree) with its height

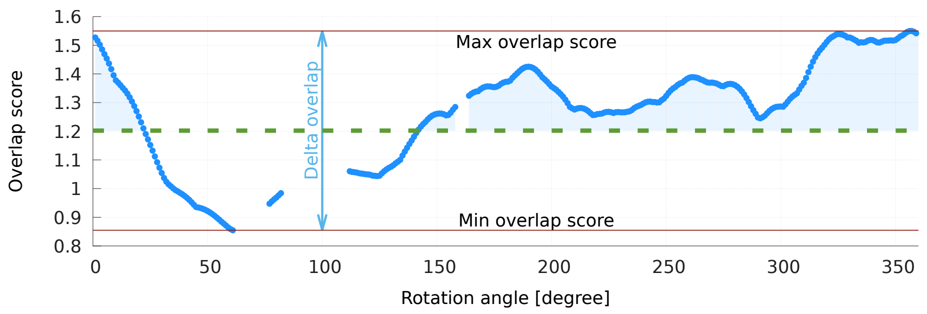


Figure 8.3: *Changes in the overlap score by rotating a fragment of the ligand. The x-axis represents the angle of the rotation, while the y-axis represents the overlap score of the ligand.*

normalized with respect to the delta overlap. From this plot, we can notice that the peaks that contain the maximum overlap are usually higher than 50 degrees, while narrow peaks rarely reach the maximum height. We can conclude that the behaviour of the overlap score is rather smooth since small peaks are also narrow.

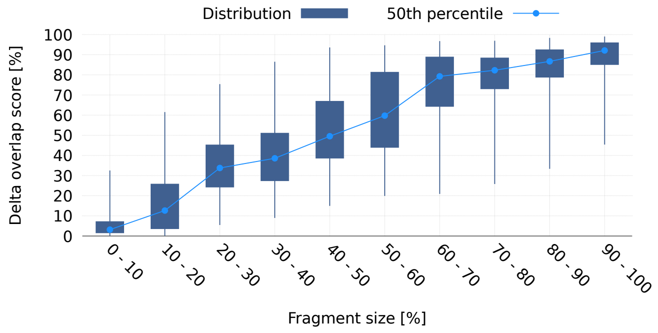
Figure 8.4c shows on the y-axis the number of peaks which are contained in a fragment, by changing the fragment size on the x-axis. We can notice how larger fragments usually have only one peak, while smaller ones tend to have more.

Besides the functional behaviour of the overlap score, Figure 8.5a shows the detailed composition of the time spent by the application to find the best rotation angle of a fragment of the ligand (y-axis) according to the size of the fragment (x-axis). From the execution time, we highlight the time spent by measuring the overlap score (MeasureOverlap) and the time spent on checking if the evaluated angle is admissible (CheckBumps). From the plot, we see that computing the overlap score is independent of the size of the fragment. We expected this result since it involves the evaluation of the whole ligand.

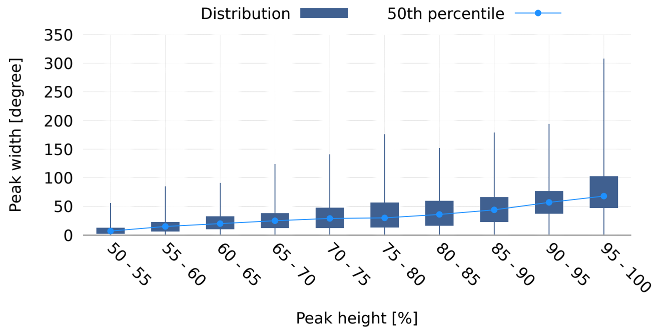
Moreover, Figure 8.5b shows the frequency of the size of a fragment. Due to the definition of the ligand database, smaller and larger fragments are slightly more frequent with respect to the other sizes.

8.3.3 Exposing Tunable Application Knobs

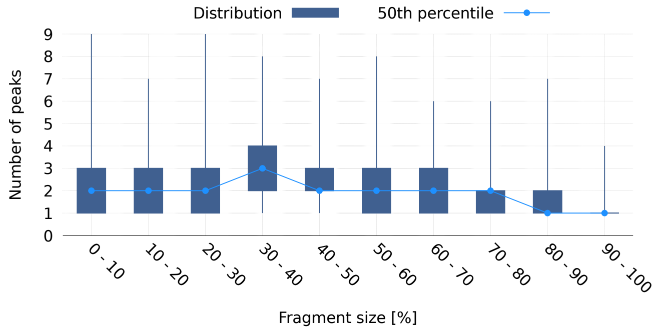
In the original implementation of LiGenDock [87], the authors listed several parameters that alter the behaviour of the docking algorithm. However, most of them are chemical-specific parameters that do not impact the ex-



(a) *Distribution of the delta overlap.*



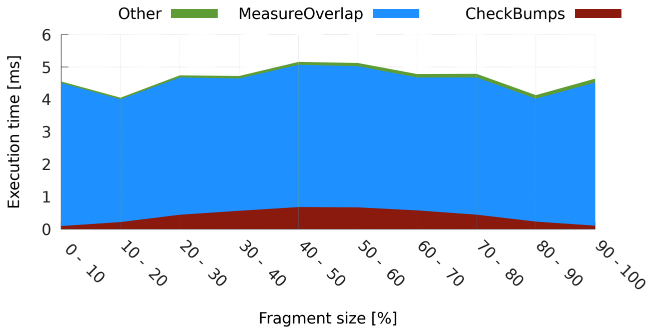
(b) *Distribution of the peak geometry.*



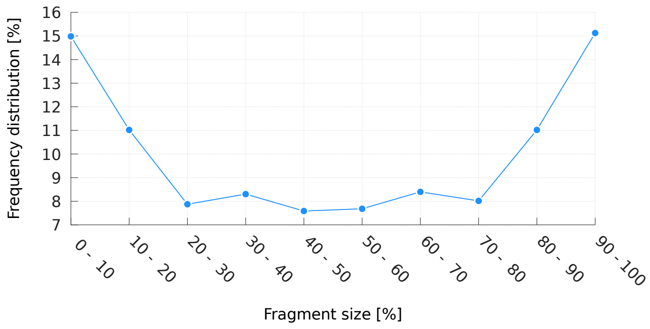
(c) *Distribution of the number of peaks.*

Figure 8.4: *Analysis on the peaks of overlap across different fragments. Each plot shows the minimum value, the 25th, 50th, 75th percentile and the maximum value.*

ecution time of *GeoDock-MA*. The only exception is a constant parameter which performs loop perforation [5] on the loops that rotates a fragment of the ligand (lines from 4 to 5 in Algorithm 3). In particular, the baseline version uses a step of 1°, while it is possible to increase the step size to skip iteration, thus reducing the number of evaluations, increasing the



(a) `MatchProbesShape` *execution time composition.*



(b) *Frequency distribution of the fragments.*

Figure 8.5: *Analysis of the execution time and the frequency of fragments, grouped by their relative size.*

performance of the application.

In addition to this first step, based on the analysis done in the previous section, we can exploit domain knowledge to define more aggressive software-knobs which approximate the *GeoDock-MA* results. In particular, we know from Figure 8.4a that small fragments have a limited impact on the delta overlap. Therefore, instead of applying a flat loop perforation, as in the original application, we introduce a *parametric loop perforation* which allows us to focus only on the most important fragments of the ligand. Whenever the size of the current fragment is below a given THRESHOLD, we use a coarse-grain rotation step (LOW-PRECISION STEP), otherwise, we use a fine-grain rotation step (HIGH-PRECISION STEP).

On the other hand, since `MatchProbesShape` is a greedy algorithm, we might improve the overlap score by repeating the whole procedure, thus considering multiple time each fragment. In particular, the more we repeat the procedure, the more we increase the probability to find a better pose for the target ligand. Therefore, we define the tunable software knob REPETI-

TIONS as the number of times to repeat the procedure. This step may seem counterintuitive; however, we argue that it is better to perform more times `MatchProbesShape` using aggressive approximations with respect to run `MatchProbesShape` once with fewer approximations.

Furthermore, we can extract other important information about the overlap score from the peak analysis we discussed in the previous section. In particular, we can rely on the smoothness of the overlap score through the entire rotation space, which means that each fragment has a limited number of peaks and that the most important peak is usually wide (the median is 68°). For each fragment above THRESHOLD we partition the $360°$ rotation space into several tiles of fixed-size x. Then, we peel and evaluate only one element for each tile (the central one). In the following iteration, we evaluate only the tile corresponding to the most promising peeling element, using HIGH-PRECISION STEP. Given this policy, the number of evaluated rotations (y) is function of the tile size (x) and HIGH-PRECISION STEP, as described in Figure 8.2.

$$y = \frac{360°}{x} + \frac{x}{\text{HIGH-PRECISION STEP}} \tag{8.2}$$

We are interested in minimising the number of pose evaluation while preserving a high probability to identify the most important peak. The minimization of Figure 8.2 has a unique solution, its value $\hat{x}$ is defined in Figure 8.3.

$$\hat{x} = 6 * \sqrt{10} * \sqrt{\text{HIGH-PRECISION STEP}} \tag{8.3}$$

For example, if we set HIGH-PRECISION STEP at the same accuracy of the original algorithm (1°), the optimal tile size is 18°, which means that we have a high probability of identifying the most important peak with the peeling element. In particular, Figure 8.6 shows, for each tile size (x-axis), the probability that the width of the most important peak is greater than the evaluated tile size (y1-axis, blue line) and the number of evaluated iterations (y2-axis, green line). The red line highlights the optimal value. As a consequence of Figure 8.3, we observe that an increment of HIGH-PRECISION STEP implies an increment in the value of the optimal tile size and a decrement of the probability of finding the best peak.

To summarize, starting from the original algorithm described in Figure 3, we have introduced five tunable software-knobs: HIGH-PRECISION STEP, LOW-PRECISION STEP, THRESHOLD, REPETITIONS and ENABLE REFINEMENT, to approximate the application by reducing the number of ligand evaluations. The main idea is to focus the elaboration only when it is required, according to the functional behaviour analysed in Figure 8.3.2.

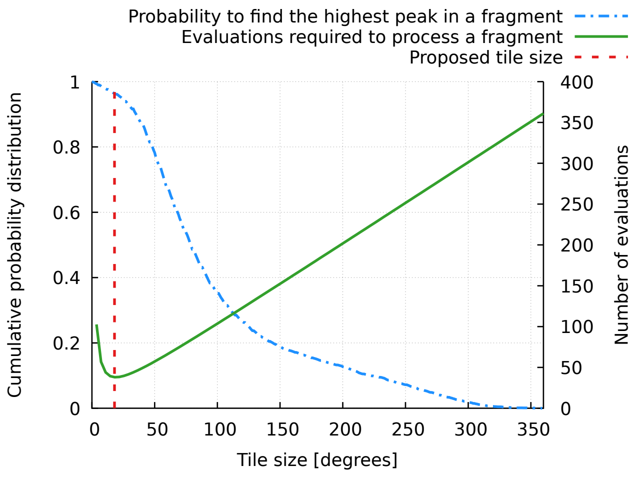


Figure 8.6: *For each tile size (x-axis), the relation between the number of evaluated rotations (y2-axis) and the probability that the width of the best peak is greater than the given size (y1-axis).*

In particular, Algorithm 4 shows the parametric algorithm of `MatchProbesShape`. The outer loop (line 2) contains all the original algorithm, which repeat the pose optimisation according to REPETITIONS. The optimization of the pose of left fragment is described between line 5 and line 16. According to the relative size of the fragment and to THRESHOLD (line 5), we perform either a coarse grained exploration using LOW-PRECISION STEP (line 6) or a fine grained exploration (lines 9-15). In the latter case, we either perform a two-step optimisation using iterative refinements, or we perform a flat exploration using HIGH-PRECISION STEP, according to ENABLE REFINEMENT. The two-step optimization evaluates the peeling elements of the rotation (line 10) and then it refines the exploration of the most promising tile using HIGH-PRECISION STEP (line 11). Due to the symmetry of the problem, the same procedure is applied to the right fragment (lines 17-28).

8.3.4 Application Autotuning

The software knobs defined in the previous subsection aim at narrowing the exploration space of ligand poses, decreasing the time-to-solution of the application and the accuracy of the results as well. However from the end-user point of view, a manual selection of the application configuration it is a nontrivial task. Therefore we use the *mARGOt* autotuning framework to select the software-knobs configuration that maximises the accuracy of the result given a time budget.

Algorithm 4: The tunable pseudo-code of the `MatchProbesShape` kernel.

```
Data: the pocket and the 3D structure of the ligand
Result: the overlap score of the ligand
get the list of rotamers;
for the number of REPETITIONS do
    foreach rotamer do
        grow the right and left fragment;
        if relative size of left fragment $\leq$ THRESHOLD then
            place the left fragment in the best angle found with step LOW-PRECISION STEP;
        end
        else
            if ENABLE REFINEMENT then
                evaluate the peeling element for each tile;
                place the left fragment in the best angle found in the best tile using step HIGH-PRECISION STEP;
            end
            else
                place the left fragment in the best angle found with step HIGH-PRECISION STEP;
            end
        end
        if relative size of right fragment $\leq$ THRESHOLD then
            place the right fragment in the best angle found with step LOW-PRECISION STEP;
        end
        else
            if ENABLE REFINEMENT then
                evaluate the peeling element for each tile;
                place the right fragment in the best angle found in the best tile using step HIGH-PRECISION STEP;
            end
            else
                place the right fragment in the best angle found with step HIGH-PRECISION STEP;
            end
        end
    end
end
return the overlap score of the ligand;
```

The autotuner must know or predict the performance of the configurations for the actual input for selecting the most suitable configuration. As the accuracy is platform-independent and it is used only to sort the configurations in terms of software knobs, it is possible to run an error profiling campaign only once, averaging the results over a representative set of pockets and ligands.

On the other hand, to complete the execution of the application within a given time budget, we must estimate the time-to-solution given the target architecture and the actual input dataset (pocket and ligands database). As the target problem is embarrassing parallel, without the need of synchronisation, the overhead of the MPI environment has been found to be negligible even scaling over a large set of nodes. Therefore, assuming homogeneous resources, we predict the time to solution of the serial case and then split it according to the available resources.

If we focus on a single software-knobs configuration, it is possible to use input data features to estimate the time to solution of the given input. To this end, we model the entire database as a set of ligands with the same *average* data features. In particular, we use a multivariate linear regression with interaction to estimate the time-to-solution t_{la} for the average ligand. The vector of predictors $\overline{x}$ is composed by the number of 3D points of the pocket xp_p, the average number of atoms in a ligand xl_a, the average number of rotamers in a ligand xl_r, and all the possible interactions among them (i.e. $xp_p \cdot xl_a$, $xp_p \cdot xl_r$, $xl_a \cdot xl_r$, and $xp_p \cdot xl_a \cdot xl_r$). Thus, the target model is simply composed of $t_{la} = \overline{\alpha} \cdot \overline{x} + \beta$, where $\overline{\alpha}$ is the vector of predictor coefficient, while β is the intercept.

To generalise the approach, we consider the parameters of the regression as a function of the proposed software knobs since the impact of the data features on the execution time is strongly dependent on the software-knobs configuration. Using this information, we can build a model that estimates the time-to-solution as stated in Equation 8.4, where $\overline{k}$ is the vector of software knobs and ν is the number of ligands to dock, in the input database.

$$t = \nu \cdot (\overline{\alpha}(\overline{k}) \cdot \overline{x} + \beta(\overline{k})) \tag{8.4}$$

Opposed to the accuracy characterisation, we should train the performance model every time we change the computing platform. However, in both cases, the experiment described in Section 8.5.1 suggests that a small database of ligands is enough to define the accuracy-performance behaviour.

To recap, we enhanced the original algorithm of the application by exposing software knobs that enables performance-accuracy trade-offs. We used an application autotuner to automatically configure the application according to simple user-oriented parameters: the number of available computational resources and the available time-budget. The data features of the actual input can be either included by the user or directly extracted by a preliminary input data analysis.

8.4 Experimental Setup

To assess the benefits of using the approximation techniques described in this chapter, we need to define the data sets used in the experiments, the metrics of interest and the platform that executes the application.

8.4.1 Data Sets

To evaluate the functional and extra-functional performance of the proposed approximation techniques, we used a database of ligands composed of 113K ligands. The molecules are different in terms of the number of atoms (between 28 and 153) and rotamers (between 2 and 53). We used 6 pockets protein-ligand complexes derived from the RCSB Protein Data Bank (PDB) [171]: 1b9v, 1c1v, 1cvu, 1c2, 1dh3, 1fm9. In particular, the PASS [170] version of the pockets has also been used together with the database of ligands. The PASS (Putative Active Sites with Spheres) version uses spheres to represent binding sites, unlike the classic grid representation of the pocket. This solution has been widely used in the context of fast docking [170].

8.4.2 Metrics of Interest

To measure the performance of *GeoDock-MA* we consider its throughput and the time to solution. In particular, we define the application throughput as the number of ligand's atoms processed in one second, while the time to solution is the time taken by the application to elaborate the input.

We use the metric *overlap degradation* to quantify the mean loss of accuracy introduced by approximation techniques with respect to the baseline, which is the configuration that leads, on average, to the better overlap score: HIGH-PRECISION STEP $= 1°$, THRESHOLD $= 0$, REPETITIONS $= 3$ and ENABLE REFINEMENT $= false$. The *overlap degradation* is defined as described in Figure 8.5,

$$score_{degradation} = (1 - \frac{overlap_{approx}}{overlap_{original}}) \times 100 \tag{8.5}$$

where $overlap_{approx}$ is the mean overlap score of the top 1% ligands of the evaluated configuration, while $overlap_{original}$ is the mean overlap score of the top 1% ligands of the non-approximated version of the application (baseline).

8.4.3 Target Platform

The platform used to execute the experiments is composed of two dedicated supercomputer NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@2.8 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The experiments are performed by using the GALILEO platform located at CINECA supercomputing center[2].

8.5 Experimental Results

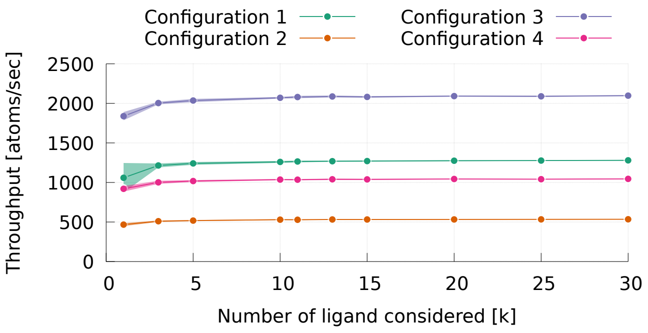
In this section, we evaluate the benefits of the proposed approach using four different experiments. Since *GeoDock-MA* is a data-dependent application, the first experiment aims at assessing data sensitivity by changing the number of ligands to use for evaluating a configuration. The second experiment aims at assessing the benefits of applying the approximation techniques, with respect to the original version of the application. We show the enabled accuracy-performance trade-off for a virtual screening procedure, and we also evaluate the effect of the overlap degradation on a single ligand docking procedure. The third experiment validates the accuracy of the time-to-solution model. Finally, the fourth experiment aims at showing the benefits of the proposed approach for the end-user on two different application scenarios.

8.5.1 Data Dependency Evaluation

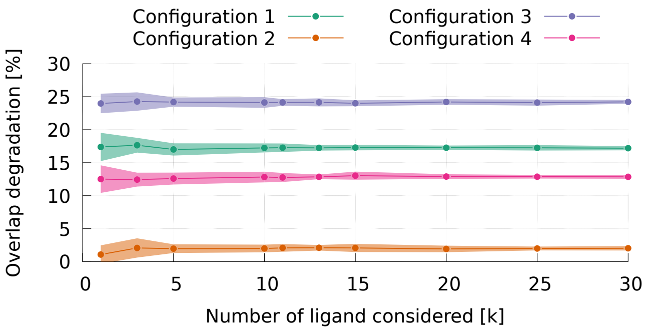
In this chapter, we aim at enhancing the geometrical docking module of LiGenDock with approximation techniques, to trade off the quality of results for throughput. Therefore we are interested in finding the set of configurations in the Pareto front, that results to be non-dominated solutions considering both target metrics. However, this application needs to find the most promising ligands across a heterogeneous data set. As a consequence, the performance might depend on which ligands the application is evaluating.

This experiment aims at assessing how much the performance of the application is dependent on changes in the dataset, to avoid a profiling phase of the alternative software-knobs configurations for each evaluated database of ligands. To this end, we evaluate four different configurations

[2]https://www.cineca.it/en



(a) *Throughput per process*



(b) *Overlap score degradation*

Figure 8.7: *Application analysis in terms of throughput per process and overlap score degradation by varying the number of ligands. For each configuration we show the average values (dot) and the standard deviation (colored area).*

of the enhanced version of *GeoDock-MA* in terms of tunable knobs. For each configuration, we characterise the application behaviour in terms of throughput and overlap degradation, by varying the number of considered ligands. The set of ligands considered to evaluate each configuration has been selected by randomising 20 times over the full set of 113K elements, thus emulating new datasets.

Figure 8.7 shows the results of the experiment. In particular, Figure 8.7a focuses on the application throughput (y-axis), while Figure 8.7b focuses on the overlap degradation (y-axis). For both of them, each dot represents the mean performance of the evaluated configuration by varying different databases of ligands. The transparent solid curve represents the uncertainty of the mean, using the standard deviation of the measures. The x-axis indicates the number of ligands considered in the evaluation.

From these results we see in Figure 8.7a that the average application throughput has a minimal dependency on both on the number of ligands in

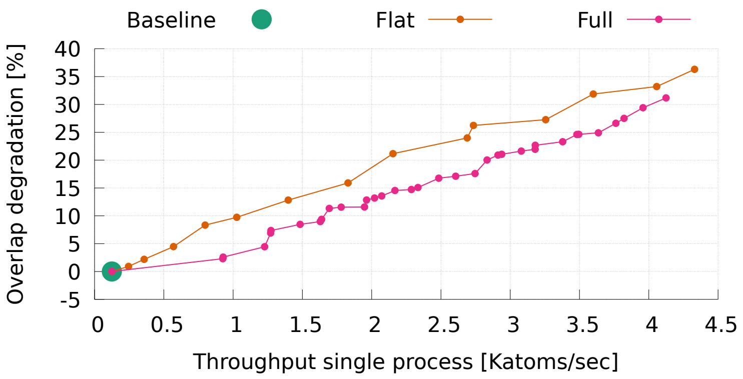


Figure 8.8: *Pareto front of* GeoDock-MA *in terms of overlap score degradation and throughput:* Flat *vs* Full.

the target database and on the input data (i.e. very small standard deviation). We expected this result, since the throughput definition we considered is related to the number of atoms of the database instead of the number of ligands, thus providing a normalised measure. On the other hand, Figure 8.7b shows how the overlap degradation is a bit more data dependent than throughput. In particular, we need to consider at least $5K$ ligands to have a steady average value. This behaviour is due to the overlap degradation definition that makes the value dependent on the top 1% ligands and therefore on which ligands are selected. However, we can determine that less than 10K ligands are enough to characterise the configurations of the enhanced version of *GeoDock-MA* for both throughput and overlap degradation.

8.5.2 Trade-off Analysis

This experiment aims at defining the performance-accuracy trade-off when we apply the approximation techniques described in Section 8.3.3. Figure 8.8 shows the Pareto front of the design space exploration carried out in a single node of Galileo using 20k ligands, targeting a single pocket.

Figure 8.8 compares the performance of the application by using a *flat* sampling on the rotation angles, as proposed in the LiGenDock paper [87], with the performance of the application using the *full* set of software knobs proposed in this chapter. In particular, the *flat* design space is the following: HIGH-PRECISION STEP $[1°, 2°, 3°, 5°, 10°, 15°, 45°, 60°]$, REPETITIONS [1, 2, 3]. While the *Full* design space, which aims at evaluating all the software knobs proposed in this chapter, is the following: HIGH-PRECISION STEP $[1°, 2°, 3°, 5°]$, LOW-PRECISION STEP $[45°, 90°]$, THRESHOLD [0, 0.3, 0.6,

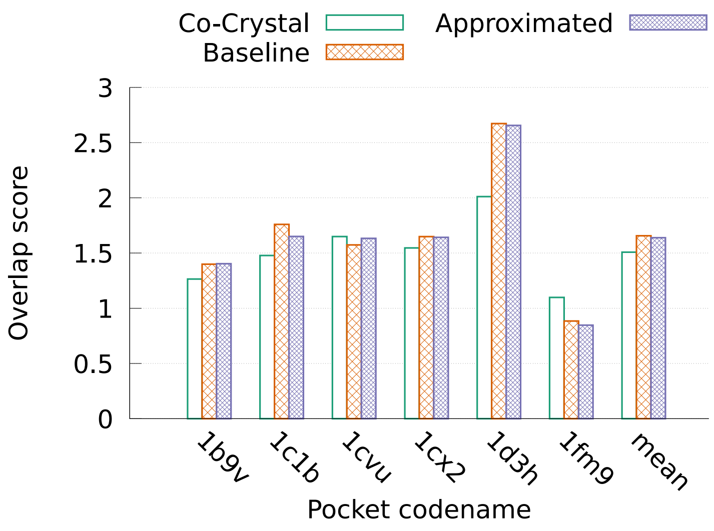


Figure 8.9: *Overlap score by varying the target pocket for the co-crystallized ligand, for the baseline and proposed approximated version.*

0.8], REPETITIONS [1, 2, 3], ENABLE REFINEMENT [$true$, $false$]. For both of them, we use a full factorial Design of Experiments. We consider as *baseline* configuration the most precise one, that is actually the same version even if derived from *flat* and *full* – i.e. HIGH-PRECISION STEP=1° and REPETITIONS=3 for the *flat* version, and HIGH-PRECISION STEP=1°, LOW-PRECISION STEP=*, THRESHOLD=0, REPETITIONS=3, and ENABLE REFINEMENT=$false$ for the *full* version.

As expected, from the results in Figure 8.8, the Pareto front derived by the *full* version dominates the one derived by the *flat* sampling. It is possible to notice how only by enabling the iterative refinement (first configuration on the *full* curve after the *Baseline*) we can significantly improve the throughput of the application (7.4X) with a limited overlap degradation (2.3%) with respect to the *baseline* version.

The Protein Data Bank (PDB) [171] contains the three-dimensional structural data of biological molecules, providing also the pose of the ligand when co-crystallized within the target pocket (i.e. the actual pose of the best ligand for that pocket). Therefore, we decided to use some pocket-ligand pairs to see the effects of accuracy degradation better. Figure 8.9 shows for each pocket-ligand pair: a) the overlap score of the crystal, b) the overlap score of the docked ligand using the *baseline* version, and c) the overlap score of the approximated version using only the iterative refinement (i.e. 7.4x speedup). We may notice that the investigated configuration

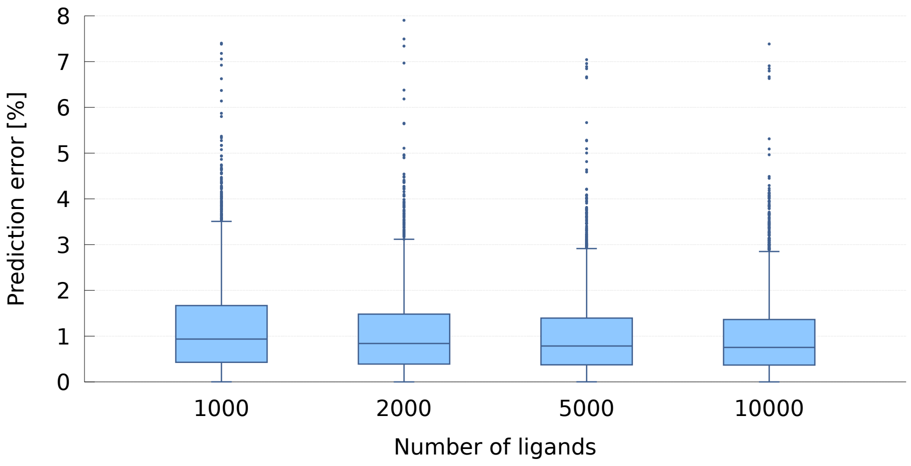


Figure 8.10: *Prediction error of the time to solution, by varying the number of ligand to dock in the target pocket.*

of the enhanced *GeoDock-MA* has a small degradation of the overlap score not only in the average case but also when considering a single target ligand. The overlap score of the co-crystallized ligand is usually lower with respect to the computed ones because the real pose of the ligand takes into consideration also chemical features which are not captured by the geometrical score.

8.5.3 Time-to-solution Model Validation

This experiment aims at validating the time-to-solution model described in Section 8.3.4. In particular, the model is defined within the design space of the *full* version described in Section 8.5.2. To compute the coefficients of the linear regression for each configuration (see Equation 8.4), we run the application several times using $1K$ ligands per pocket for each configuration. The extracted models for each configuration have an average adjusted R^2 value equal to 0.977.

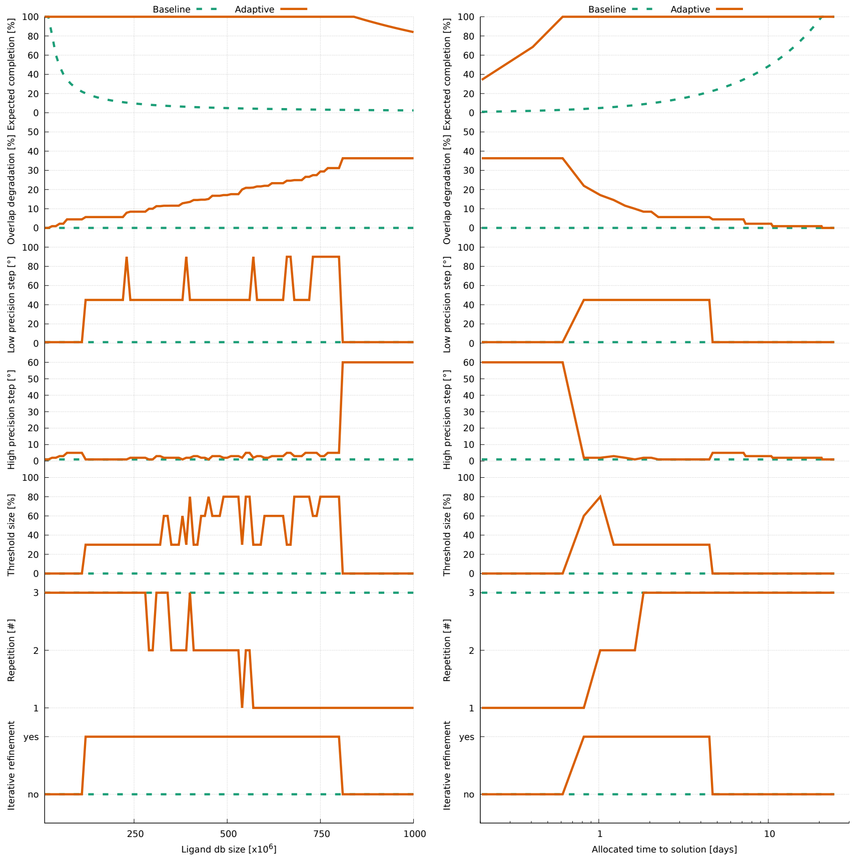
We run an experiment campaign to further validate the accuracy of the time-to-solution prediction of the model, by using a leave-one-out scheme on the pockets and using a different set of ligands with respect to those used for training. For each pocket and configuration stated in Section 8.5.2, we execute the application with three different databases randomly selected over the entire set and composed of $1k$, $2k$, $5k$ and $10k$ ligands. We do not use very small numbers for the target database size because our goal is to predict the time-to-solution during a virtual-screening process, thus composed of a large number of target ligands. For each run, we stored the predictor value of the input and the observed time-to-solution to extract the prediction error. Figure 8.10 shows the distribution of the prediction error of our model, by varying the number of ligands in the experiment. We notice that the average error is below 1% of the observed time to solution for the entire range of considered ligand-database size, while the value of outliers (we validate the application using more than $15K$ runs) reaches a maximum of 7.9%, with a trend which is stable by increasing the number of docked ligands.

8.5.4 Use-case Scenarios

The last experiment aims at evaluating the benefits of the proposed approach for the end-user, i.e. a pharmaceutical company that aims at screening a large set of ligands within a specific time budget. We envision two use cases to exploit the performance-accuracy trade-off. In the first scenario, we consider a fixed time budget for the computation, and we would like to understand what is the effect of incrementing the size of the database to investigate, to increase the chances to find a better drug. In the second one, we plan the opposite scenario. We fix the size of the database, and we observe the effects in varying the time budget, thus varying the cost of the experiment as well. We can summarise these two scenarios as attempts to provide the user with two high-level knobs: in the first case the number of ligands to be screened, while in the second one the time budget, and thus the cost of the experiment. The time-to-solution model will be used to set the right low-level application knobs included in *GeoDock-MA* while satisfying the constraints.

Figure 8.11 shows the expected behaviour of the application, assuming that the end-user is using eight nodes of the Galileo machine (see Section 8.4.3). On the y-axis are represented the expected performance of the application (top two plots) and the selected configuration of the software knobs (bottom five plots). We define the performance of the application in terms of the expected completion percentage of the planned ligand database and the related overlap degradation of the result. The completion percentage is the ratio between the number of ligands docked within the allocated time budget and the size of the target ligand database. Each plot includes two lines, one for the baseline version of *GeoDock-MA* the other for the adaptive version presented in this chapter.

The x-axis represents the high-level parameter tuned by the end-user, according to the scenario. On one side, in Figure 8.11a the end-user would



(a) *Scenario 1. Varying the size of the ligand database allocating one day to the time budget.*

(b) *Scenario 2. Varying the allocated time budget given the ligand database size equal to* 500×10^6.

Figure 8.11: GeoDock-MA *behavior in terms of expected percentage of ligand database completion, expected overlap degradation, and the selected configuration (a) by varying the size of the input, and (b) the time budget, when using 8 nodes of Galileo.*

like to tune the size of the database of ligands to complete the job within a time budget of one day. On the other side, in Figure 8.11b the end-user would like to tune the time budget of the application to dock a database of 500×10^6 ligands.

In both cases, from the results, we notice that the proposed software knobs enable a swing of several orders of magnitude for the end-user to tune the parameters of the job, i.e. problem size and time to solution. Moreover, by using an autotuner together with the time to solution model, we can alleviate the burden of the manual selection of the software knobs from the end-user, exposing more straightforward parameters. While the average trend of the application knobs values for both scenarios can be derived from their meaning, the actual values and when to switch among the configurations according to the high-level constraints (i.e. problem size and time to solution) is something that is hard to know without any automatic support. A clear example of this is the behaviour of the THRESHOLD-value within the middle range (300–700×10^6) of the problem size considered in the experiment shown by Figure 8.11a.

In terms of application performance, we can notice how in the scenario where we fix the time to solution (see Figure 8.11a), while the proposed *adaptive* version can keep the completion rate of the ligand database equal to 100% up to 850×10^6, the baseline version rapidly decreases the rate to very low values when enlarging the database size. This behaviour is due to the capability of the proposed *adaptive* version, not present in the *baseline*, to trade-off accuracy and performance. Moreover, the second plot in Figure 8.11a also demonstrates how the deep parametrisation introduced in the target Mini-App provides a smooth degradation of the application quality. On the other scenario shown in Figure 8.11b, where we fixed the size of the target database, and we varied the allocated time budget, we observed similar behaviour. By increasing the time budget, the proposed *adaptive* version rapidly reaches (< 1 day) the value of 100% completion at the cost of low accuracy, while for the baseline we had to allocate more than 20 days.

Finally, in both scenarios, the experimental results show how the *adaptive* approach can provide an output with a limited overlap degradation (less than 10%), while the baseline can process only the 10% of input data set. This result demonstrates the effectiveness of the extracted low-level knobs in *GeoDock-MA*.

8.6 Summary

In this chapter, we have analysed *GeoDock-MA* as a representative batch job application which runs in HPC centre. From an analysis campaign of the application domain, we identified five software-knobs that enable an accuracy-performance trade-off, by focusing the computation only where it is likely to have a significant impact on the output. The adaptive version of *GeoDock-MA* provides different accuracy levels according to the needs of the virtual screening experimental campaign, automatically managed by *mARGOt*. In particular, experimental results demonstrated how, by varying the quality of the results, the application could complete a virtual screening campaign over a given ligand database, within a wide range of time-to-solution. These results represent a considerable advantage for pharmaceutical industries in a context in which the use of HPC system and software in drug discovery have become a valuable asset to find novel drugs. Due to the large number of theoretical molecules that may be evaluated, this procedure can lower the cost of the drug discovery process or evaluate a larger number of ligands, increasing the chances of finding better candidates. With respect to other case studies described in previous chapters, this one is more similar to a static tuning. The dynamicity is due to the fact that we do not configure the application with a one-fits-all configuration, but we rely on application knowledge and input features to select the most suitable configuration for the actual input. Finally, thanks to the analysis derived from the work presented in this chapter, the complete version of the geometrical docking application (not the MiniApp), has been optimized and tuned to run a very large run on the whole MARCONI machine from CINECA (>250Kcores, >10PetaFlops system and position number 17 on the top 500). In particular, this experiment will perform one of the largest virtual screening campaign for the ZIKA virus considering a database of 1.2B ligands. Results on this run are not shown in the current version of the thesis since it is going to take place during the first days of November.

CHAPTER 9

Conclusions

In this thesis, we addressed the problem on how to enhance a target application with an adaptation layer that provides *self-optimization* capabilities. The main outcome is a dynamic autotuning framework that exposes mechanisms to adapt reactively and proactively, and it enables to learn application knowledge at runtime, in a distributed fashion. We have experimentally evaluated the proposed approach and its exploitation in two different scenarios. Moreover, we described how it is possible to leverage approximate techniques in two application case studies that belong to entirely different contexts. The remainder of this chapter summarises the finding and limitation of the proposed approach and provides recommendations for future works.

9.1 Main contributions

The main results of the work carried out in this thesis might be summarised as follows:

1. The features exposed by the adaptation layer has been evaluated in different scenarios, ranging from embedded to High-Performance Computing. Experimental results show how leveraging feedback infor-

mation from monitors provides a mechanism to adapt the application knowledge according to the observed behaviour, under the assumption of linear error propagation. Given the *mARGOt* flexibility for expressing application requirements, it is possible to further improve efficiency by adapting the requirements according to external events, for example using information from the current input. Moreover, it is possible to identify and seize optimisation opportunities by leveraging features of the current input. Furthermore, by learning the application knowledge at runtime, it is possible to learn complex relations between software-knobs configuration, EFPs of interest and the current input, increasing the computation efficiency.

2. The orthogonality between resource managers and application autotuning has been evaluated, by applying different adaptation schemes in a dynamic workload with co-running applications. Our tests show that the average performance of using *mARGOt* as a lightweight resource manager is very close to the performance achieved with a combined approach based on a centralised resource manager. Moreover, our approach is more portable and less intrusive from an application design point of view. However, it does not provide any guarantee of *fairness* nor *optimality* in resource allocation.

3. An approach has been proposed to combine the adaptation mechanisms of *mARGOt*, with source-to-source transformations of the LARA aspect-oriented language [16], and with insight provided by the COBAYN compiler autotuner [15], to provide to application developers a seamless online compiler and system runtime autotuning framework. The proposed approach can provide self-optimization capabilities to the target application, in terms of compiler options and OpenMP parameters, in a transparent way to application developers.

4. In the context of smart cities, we focused on a time-dependent probabilistic routing algorithm, by analysing the relationship between end-user requirements, application software-knob and features of the input. Experimental results show how it is possible to drastically improve computation efficiency, by leveraging input features.

5. In the context of a drug discovery process, we focused on a geometrical docking miniapp, by analysing the effect of approximation techniques for the extra-functional properties of interest. Experimental results show how it is possible to increase the application throughput by one order of magnitude, with an accuracy degradation less than

30%. By using *mARGOt*, end-user can harness this tradeoff to satisfy a time-to-solution constraint.

9.2 Recommendation for future works

Experimental evaluations of the proposed framework show promising results; however, there are still open questions to investigate. In our opinion, the most challenging point to solve are the following:

1. *mARGOt* uses feedback information from monitor to adapt the knowledge base by assuming a linear error propagation, as a local reaction mechanism for the application. However, there are cases where this assumption is wrong, typically when co-running applications share computational resources. Therefore, this reaction mechanism should be improved. For example, it is possible to leverage a change point detection mechanism [172] to re-trigger a Design Space Exploration to update the application knowledge.

2. Even if *mARGOt* can manage different application regions of code, it considers them independently. As demonstrated in previous work [39], if different regions of code share common software-knobs, an independent tuning of them might lead to sub-optimal performance.

3. In the current implementation, *mARGOt* relies on end-users and application developers to define error metrics and input features since they are utterly application-specific. However, previous work [7,59] shows that generic error metrics and input features might apply to different applications. It may be of interest to investigate more their effectiveness to decrease the integration effort of the approach.

4. The *mARGOt* framework can handle input features to adapt proactively. However, it does not provide any mechanism to extract such features from the current input automatically. It is possible to envision a new module that can extract common information from “iterable” data structures, such as the average, standard deviation, or autocorrelation [59].

We hope that the work discussed in this thesis, on the one hand, will help application developers to improve computation efficiency. On the other hand, we hope that it will help researchers to advance toward an autonomic manager. To this end, we publicly released the *mARGOt* source code [14], along with user manuals and Doxygen documentation.

Publications

Articles published or under review in international journals

1. Ralph Görgen, Kim Grüttner, Fernando Herrera, Pablo Peñil, Julio Medina, Eugenio Villar, Gianluca Palermo, William Fornaciari, Carlo Brandolese, Davide Gadioli, Sara Bocchio, Luca Ceva, Paolo Azzoni, Massimo Poncino, Sara Vinco, Enrico Macii, Salvatore Cusenza, John Favaro, Raùl Valencia, Ingo Sander, Kathrin Rosvall, Davide Quaglia "CONTREX: Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties." Microprocessors and Microsystems 51 (2017): 39-55.

2. Davide Gadioli, Emanuele Vitali, Gianluca Palermo, Cristina Silvano "mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing" IEEE Transactions on Computers, accepted, in publishing

3. Davide Gadioli, Gianluca Palermo, Stefano Cherubin, Emanuele Vitali, Giovanni Agosta, Candida Manelfi, Andrea R. Beccari, Carlo Cavazzoni, Nico Sanna, Cristina Silvano "Tunable Approximations for Controlling Time-to-Solution in an HPC Molecular Docking Mini-App", IEEE Transactions on Computers, under peer review

4. Davide Gadioli, Emanuele Vitali, Gianluca Palermo, Cristina Silvano "mARGOt: a Dynamic Autotuning Framework Targeting Adaptivity and Controllable Approximation" SoftwareX, under peer review

5. Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Martin Golasowski, João Bispo, Pedro Pinto, Jan Martinovič, Kateřina Slaninová, João M. P. Cardoso, Cristina Silvano “An Efficient Monte Carlo-based Probabilistic Time-Dependent Routing Calculation Targeting a Server-Side Car Navigation System” IEEE Transactions on Emerging Topics in Computing, under peer review

6. Tomáš Martinovič, Davide Gadioli, Gianluca Palermo, Cristina Silvano “On-line Application Autotuning Exploiting Ensemble Models” IEEE Transactions on Computers, under peer review

Articles published in proceedings of international conferences

1. Edoardo Paone, Davide Gadioli, Gianluca Palermo, Vittorio Zaccaria, Cristina Silvano ”Evaluating orthogonality between application autotuning and run-time resource management for adaptive OpenCL applications“ IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2015

2. Davide Gadioli, Simone Libutti, Giuseppe Massari, Edoardo Paone, Michele Scandale, Patrick Bellasi, Gianluca Palermo, Vittorio Zaccaria, Giovanni Agosta, William Fornaciari, Cristina Silvano ”OpenCL application auto-tuning and run-time resource management for multi-core platforms” IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2014

3. Davide Gadioli, Gianluca Palermo, Cristina Silvano “Application autotuning to support runtime adaptivity in multicore architectures” IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015

4. Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, João Bispo, João M. P. Cardoso, Rui Abreu, Pedro Pinto, Carlo Cavazzoni, Nico Sanna, Andrea R Beccari, Radim Cmar, Erven Rohou ”The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems“ Proceedings of the ACM International Conference on Computing Frontiers, 2016

5. Cristina Silvano, Giovanni Agosta, Jorge Barbosa, Andrea Bartolini, Andrea R. Beccari, Luca Benini, João Bispo, João M. P. Cardoso, Carlo Cavazzoni, Stefano Cherubin, Radim Cmar, Davide Gadioli, Candida Manelfi, Jan Martinovič, Ricardo Nobre, Gianluca Palermo,

Martin Palkovič, Pedro Pinto, Erven Rohou, Nico Sanna and Kateřina Slaninová "The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems" IEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017

6. Davide Gadioli, Ricardo Nobre, Pedro Pinto, Emanuele Vitali, Amir H Ashouri, Gianluca Palermo, João M. P. Cardoso, Cristina Silvano "SOCRATES-A seamless online compiler and system runtime autotuning framework for energy-aware applications" Design, Automation and Test in Europe Conference and Exhibition (DATE), 2018

7. Cristina Silvano, Gianluca Palermo, Giovanni Agosta, Amir H Ashouri, Davide Gadioli, Stefano Cherubin, Emanuele Vitali, Luca Benini, Andrea Bartolini, Daniele Cesarini, João M. P. Cardoso, João Bispo, Pedro Pinto, Riccardo Nobre, Erven Rohou, Loïc Besnard, Imane Lasri, Nico Sanna, Carlo Cavazzoni, Radim Cmar, Jan Martinovič, Kateřina Slaninová, Martin Golasowski, Andrea R Beccari, Candida Manelfi "Autotuning and adaptivity in energy efficient HPC systems: the ANTAREX toolbox" Proceedings of the ACM International Conference on Computing Frontiers, 2018

8. Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M. P. Cardoso, Carlo Cavazzoni, Stefano Cherubin, Davide Gadioli, Martin Golasowski, Imane Lasri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Nico Sanna, Kateřina Slaninová, Emanuele Vitali "ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing" Euromicro Conference on Digital System Design, 2018

Articles published in proceedings of international workshops

1. Davide Gadioli, Gianluca Palermo, Cristina Silvano "Application Adaptation at Runtime through Dynamic Knobs Autotuning." RES4ANT@ DATE, 2016

2. Ahmet Erdem, Davide Gadioli, Gianluca Palermo, Cristina Silvano "Design Space Pruning and Computational Workload Splitting for Autotuning OpenCL Applications" Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, 2018

3. Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Cristina Silvano “Accelerating a Geometric Approach to Molecular Docking with OpenACC” Proceedings of the 5th International Workshop on Parallelism in Bioinformatics, 2018

Posters published/presented in poster sessions co-located with international conference

1. Jan Martinovič, Kateřina Slaninová, Martin Golasowski, Radim Cmar, João M. P. Cardoso, João Bispo, Gianluca Palermo, Davide Gadioli, Cristina Silvano “DSL and Autotuning Tools for Code Optimisation on HPC Inspired by Navigation Use Case” Supercomputing, 2016

Bibliography

[1] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[2] Marc Duranton, Koen De Bosschere, Christian Gamrat, Jonas Maebe, Harm Munk, and Olivier Zendra. The hipeac vision 2017, 2017.

[3] 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.

[4] Edoardo Paone, Gianluca Palermo, Vittorio Zaccaria, Cristina Silvano, Diego Melpignano, Germain Haugou, and Thierry Lepley. An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 503–512. ACM, 2012.

[5] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[6] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[7] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 25–34. ACM, 2010.

[8] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[9] Qian Zhang and Qiang Xu. Approxit: A quality management framework of approximate computing for iterative methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

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

[11] 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.

[12] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.

[13] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136 – 2145, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[14] mARGOt framework git repository. `https://gitlab.com/margot_project/core`, 20018.

[15] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2):21:1–21:25, 2016.

[16] João M. P. Cardoso, José G. F. 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. *Softw. Pract. Exper.*, 2016.

[17] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[18] Eric M Dashofy, André Van der Hoek, and Richard N Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM, 2002.

[19] Rema Ananthanarayanan, Mukesh Mohania, and Ajay Gupta. Management of conflicting obligations in self-protecting policy-based systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 274–285. IEEE, 2005.

[20] Markus C Huebscher and Julie A McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[21] Sara Mahdavi-Hezavehi, Vinicius HS Durelli, Danny Weyns, and Paris Avgeriou. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology*, 90:1–26, 2017.

[22] William E Walsh, Gerald Tesauro, Jeffrey O Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77. IEEE, 2004.

[23] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 65–73. IEEE, 2006.

[24] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[25] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. Seec: a general and extensible framework for self-aware computing. 2011.

[26] Juan A Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B Bartolini, Nitesh Mor, et al. Tessellation: refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, page 76. ACM, 2013.

[27] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau. Resource-aware programming and simulation of mpsoc architectures through extension of x10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, pages 48–55. ACM, 2011.

[28] Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, and Giuseppe Lipari. Aquosa-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, 2009.

[29] Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. Performance portability across heterogeneous socs using a generalized library-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(2):21, 2014.

[30] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *European Conference on Parallel Processing*, pages 9–20. Springer, 2009.

[31] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, et al. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In *International Workshop on Applied Parallel Computing*, pages 328–342. Springer, 2012.

[32] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, 2010.

[33] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[34] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):88, 2013.

[35] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. *ACM SIGPLAN Notices*, 49(4):35–50, 2014.

[36] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May OŔeilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.

[37] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. Automatically exploring tradeoffs between software output fidelity and energy costs. *IEEE Transactions on Software Engineering*, 2017.

[38] Ari Rasch, Michael Haidl, and Sergei Gorlatch. Atf: A generic auto-tuning framework. In *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*, pages 64–71. IEEE, 2017.

[39] Juan J Durillo, Philipp Gschwandtner, Klaus Kofler, and Thomas Fahringer. Multi-objective region-aware optimization of parallel programs. *Parallel Computing*, 2018.

[40] 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.

[41] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[42] 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.

[43] 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.

[44] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 195–202. IEEE, 2015.

[45] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and auto-tuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[46] Shoaib Ashraf Kamil. *Productive high performance parallel programming with auto-tuned domain-specific embedded languages*. University of California, Berkeley, 2012.

[47] Michael J Voss and Rudolf Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 163–170. IEEE, 2000.

[48] Joseph P. Bigus, Don A. Schlosnagle, Jeff R. Pilgrim, W Nathaniel Mills III, and Yixin Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[49] Sujay Parekh, Neha Gandhi, Joseph Hellerstein, Dawn Tilbury, T Jayram, and Joe Bigus. Using control theory to achieve service level objectives in performance management. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 841–854. IEEE, 2001.

[50] Yixin Diao, Joseph L Hellerstein, Sujay Parekh, Rean Griffith, Gail Kaiser, and Dan Phung. Self-managing systems: A control theory foundation. In *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pages 441–448. IEEE, 2005.

[51] Haipeng Guo. A bayesian approach for automatic algorithm selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI03), Workshop on AI and Autonomic Computing, Acapulco, Mexico*, pages 1–5, 2003.

[52] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

[53] 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.

[54] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 13–24. IEEE, 2013.

[55] 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.

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

[57] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 13–24. ACM, 2015.

[58] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 154–168. ACM, 2018.

[59] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. *ACM SIGPLAN Notices*, 51(6):161–176, 2016.

[60] 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.

[61] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[62] Joshua San Miguel and Natalie Enright Jerger. The anytime automaton. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 545–557. IEEE Press, 2016.

[63] 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.

[64] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 85–96. IEEE Computer Society, 2011.

[65] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OŔeilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, volume 50, pages 379–390. ACM, 2015.

[66] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May OŔeilly, 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.

[67] Vittorio Zaccaria, Gianluca Palermo, Fabrizio Castro, Cristina Silvano, and Giovanni Mariani. Multicube explorer: An open source framework for design space exploration of chip multi-processors. In *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pages 1–7. VDE, 2010.

[68] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pages 257–269. Springer, 2005.

[69] Vincent M Weaver, Daniel Terpstra, Heike McCraw, Matt Johnson, Kiran Kasichayanula, James Ralph, John Nelson, Philip Mucci, Tushar Mohan, and Shirley Moore. Papi 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013.

[70] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[71] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 185–194. ACM, 2006.

[72] Zhenghua Nie and Jeffrey S Racine. The crs package: Nonparametric regression splines for continuous and categorical predictors. *R Journal*, 4(2), 2012.

[73] Delphine Dupuy, Céline Helbert, and Jessica Franco. DiceDesign and DiceEval: Two R Packages for Design and Analysis of Computer Experiments. *Journal of Statistical Software*, 2015.

[74] Jerome H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics*, 1991.

[75] Charles J. Stone, Mark H. Hansen, Charles Kooperberg, and Young K. Truong. Polynomial splines and their tensor products in extended linearmodeling. *Annals of Statistics*, 1997.

[76] R. WEBSTER and T. M. BURGESS. OPTIMAL INTERPOLATION AND ISARITHMIC MAPPING OF SOIL PROPERTIES III CHANGING DRIFT AND UNIVERSAL KRIGING. *Journal of Soil Science*, 1980.

[77] Leo Breiman. Bagging predictors. *Machine Learning*, 1996.

[78] Leo Breiman. Stacked regressions. *Machine Learning*, 24(1):49–64, jul 1996.

[79] S original by Berwin A. Turlach R port by Andreas Weingessel <Andreas.Weingessel@ci.tuwien.ac.at>. *quadprog: Functions to solve Quadratic Programming Problems.*, 2013. R package version 1.5-5.

[80] Brian. Everitt and Anders. Skrondal. *The Cambridge dictionary of statistics*. 2010.

[81] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE transactions on circuits and systems for video technology*, 19(7):1073–1079, 2009.

[82] Radek Tomis, Lukáš Rapant, Jan Martinovič, Kateřina Slaninová, and Ivo Vondrák. Probabilistic time-dependent travel time computation using monte carlo simulation. In *International Conference on High Performance Computing in Science and Engineering*, pages 161–170. Springer, 2015.

[83] Martin Golasowski, Radek Tomis, Jan Martinovič, Kateřina Slaninová, and Lukáš Rapant. Performance evaluation of probabilistic time-dependent travel time computation. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 377–388. Springer, 2016.

[84] To Thanh Binh. A Multiobjective Evolutionary Algorithm - The Study Cases. *INSTITUTE FOR AUTOMATION AND COMMUNICATION*, 1999.

[85] Frank Kursawe. A variant of evolution strategies for vector optimization. In *Lecture Notes in Computer Science*, 1991.

[86] Andrea R Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. Ligen: a high performance workflow for chemistry driven de novo design. *Journal of Chemical Information and Modeling*, 53(6):1518–1527, 2013.

[87] Claudia Beato, Andrea R Beccari, Carlo Cavazzoni, Simone Lorenzi, and Gabriele Costantino. Use of experimental design to optimize docking performance: The case of ligendock, the docking module of ligen, a new de novo design program, 2013.

[88] Khronos Group. OpenCL Specification. 2012.

[89] Stefan Wildermann, Tobias Ziermann, and Jurgen Teich. Game-Theoretic Analysis of Decentralized Core Allocation Schemes on Many-Core Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1498–1503, New Jersey, 2013. IEEE Conference Publications.

[90] Holger Endt and Kay Weckemann. Remote Utilization of OpenCL for Flexible Computation Offloading using Embedded ECUs, CE Devices and Cloud Servers. In *PARCO*, pages 133–140, 2011.

[91] Yanzhi Wang, Shuang Chen, Hadi Goudarzi, and Massoud Pedram. Resource allocation and consolidation in a multi-core server cluster using a Markov decision process model. In *International Symposium on Quality Electronic Design (ISQED)*, pages 635–642. IEEE, March 2013.

[92] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Power Optimization in Embedded Systems via Feedback Control of Resource Allocation. *IEEE Transactions on Control Systems Technology*, 21(1):239–246, January 2013.

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

[94] Chantal Ykman-Couvreur, Philipp A. Hartmann, Gianluca Palermo, Fabien Colas-Bigey, and Laurent San. Run-time resource management based on design space exploration. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '12*, page 557, New York, USA, October 2012. ACM Press.

[95] Giovanni Mariani, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. ARTE: An Application-specific Run-Time managEment framework for multi-cores based on queuing models. *Parallel Computing*, 39(9):504–519, September 2013.

[96] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-Based Local Stereo Matching Using Orthogonal Integral Images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, July 2009.

[97] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *IEEE IPDPS*, 2009.

[98] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Programming Language Design and Implementation*, 2009.

[99] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. 1997.

[100] Apan Qasem, Guohua Jin, and John Mellor-crummey. Improving performance with integrated program transformations. Technical report, 2003.

[101] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing*, 2006.

[102] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Workshop on Languages and Compilers for Parallel Computing*, 2010.

[103] Qing Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software-Practice & Experience*, 2012.

[104] Dheya Mustafa and Rudolf Eigenmann. Portable Section-level Tuning of Compiler Parallelized Applications. In *High Performance Computing, Networking, Storage and Analysis*, 2012.

[105] Wei Wang, John Cavazos, and Allan Porterfield. Energy Auto-tuning using the Polyhedral Approach. In *Workshop on Polyhedral Compilation Techniques*, 2014.

[106] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. *Auto-tuning for Energy Usage in Scientific Applications*. 2011.

[107] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Symposium on Parallel Distributed Processing*, 2009.

[108] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.

[109] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.*, 9(3):21:1–21:30, October 2012.

[110] Grigori Fursin et al. Milepost-gcc: Machine learning enabled self-tuning compiler. *Intern, Journal of Parallel Programming*, 2011.

[111] P. Pinto, R. Abreu, and J. M. P. Cardoso. Fault Detection in C Programs using Monitoring of Range Values: Preliminary Results. *ArXiv*, 2015.

[112] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.

[113] Cristina Videira Lopes and Gregor Kiczales. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.

[114] Paolo Toth and Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*, volume 18. SIAM, 2014.

[115] Michael J. Gilman. A brief survey of stopping rules in monte carlo simulations. In *Proceedings of the Second Conference on Applications of Simulations*, pages 16–20. Winter Simulation Conference, 1968.

[116] Anton Agafonov and Vladislav Myasnikov. Reliable routing in stochastic time-dependent network with the use of actual and forecast information of the traffic flows. In *Intelligent Vehicles Symposium (IV), 2016 IEEE*, pages 1168–1172. IEEE, 2016.

[117] Samitha Samaranayake, Sebastien Blandin, and A Bayen. A tractable class of algorithms for reliable routing in stochastic networks. *Procedia-Social and Behavioral Sciences*, 17:341–363, 2011.

[118] Yu Marco Nie and Xing Wu. Shortest path problem considering on-time arrival probability. *Transportation Research Part B: Methodological*, 43(6):597–613, 2009.

[119] Maleen Abeydeera and Samitha Samaranayake. Gpu parallelization of the stochastic on-time arrival problem. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–8. IEEE, 2014.

[120] Mehrdad Niknami and Samitha Samaranayake. Tractable pathfinding for the stochastic on-time arrival problem. In *International Symposium on Experimental Algorithms*, pages 231–245. Springer, 2016.

[121] Evdokia Nikolova, Jonathan Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. *Algorithms–ESA 2006*, pages 552–563, 2006.

[122] Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In Daniele Frigioni and Sebastian Stiller, editors, *ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems - 2013*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 108–122, Sophia Antipolis, France, September 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[123] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Alternative routing: k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 68:1–68:4, 2015.

[124] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Exact and approximate algorithms for finding k-shortest paths with limited overlap. pages 414–425, 2017.

[125] Hans Janssen. Monte-carlo based uncertainty analysis: Sampling efficiency and sampling convergence. *Reliability Engineering & System Safety*, 109:123 – 132, 2013.

[126] Q. Xu, M. Sbert, M. Feixas, and J. Sun. A new adaptive sampling technique for monte carlo global illumination. In *2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 191–196, Oct 2007.

[127] J. S. Miguel and N. E. Jerger. The anytime automaton. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 545–557, June 2016.

[128] Elise Miller-Hooks and Hani Mahmassani. Path comparisons for a priori and time-adaptive decisions in stochastic, time-varying networks. *European Journal of Operational Research*, 146(1):67–82, 2003.

[129] Jan Martinovič, Václav Snášel, Jiří Dvorský, and Pavla Dráždilová. Search in documents based on topical development. In Vaclav Snášel, Piotr S. Szczepaniak, Ajith Abraham, and Janusz Kacprzyk, editors, *Advances in Intelligent Web Mastering - 2*, pages 155–166, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[130] Mohammad Asghari, Tobias Emrich, Ugur Demiryurek, and Cyrus Shahabi. Probabilistic estimation of link travel times in dynamic road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 47. ACM, 2015.

[131] JM Juritz, JWF Juritz, and MA Stephens. On the accuracy of simulated percentage points. *Journal of the American Statistical Association*, 78(382):441–444, 1983.

[132] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley and Sons, 2003.

[133] D. Zwillinger and S. Kokoska. *CRC Standard Probability and Statistics Tables and Formulae*. Chapman & Hall, 2000.

[134] Radek Tomis, Jan Martinovič, Kateřina Slaninová, Lukáš Rapant, and Ivo Vondrák. Time-dependent route planning for the highways in the czech republic. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 145–153. Springer, 2015.

[135] Phelim P. Boyle. Options: A monte carlo approach. *Journal of Financial Economics*, 4(3):323–338, 1977.

[136] Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005.

[137] Federal Highway Administration US Department of Transportation. *US Department of Transportation, Federal Highway Administration – Traffic Report*. 2014.

[138] gov.uk UK Department for Transport. *Average annual daily flow and temporal traffic distributions*. 2017.

[139] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: Performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, March 2009.

[140] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[141] Milano Agenzia Mobilita' Ambiente e Territorio. *Annual Mobility Report*.

[142] Marco Bedogni, Milano Agenzia Mobilita' Ambiente e Territorio. *Road Traffic Measures in The City of Milan*.

[143] Marco Gribaudo, Pietro Piazzolla, and Giuseppe Serazzi. Consolidation and replication of vms matching performance objectives. In Khalid Al-Begain, Dieter Fiems, and Jean-Marc Vincent, editors, *Analytical and Stochastic Modeling Techniques and Applications*, pages 106–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[144] Evanthia Lionta, George Spyrou, Demetrios K. Vassilatis, and Zoe Cournia. Structure-based virtual screening for drug discovery: Principles, applications and recent advances. *Current Topics in Medicinal Chemistry*, 14(16):1923–1938, 2014.

[145] Andrea R. Beccari, Marica Gemei, Matteo Lo Monte, Nazareno Menegatti, Marco Fanton, Alessandro Pedretti, Silvia Bovolenta, Cinzia Nucci, Angela Molteni, Andrea Rossignoli, Laura Brandolini, Alessandro Taddei, Lorena Za, Chiara Liberati, and Giulio Vistoli. Novel selective, potent naphthyl trpm8 antagonists identified through a combined ligand- and structure-based virtual screening approach. In *Scientific reports*, 2017.

[146] Douglas B. Kitchen, Hélène Decornez, John R. Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery*, 3, Nov 2004. Review Article.

[147] Paul D Lyne. Structure-based virtual screening: an overview. *Drug Discovery Today*, 7(20):1047 – 1055, 2002.

[148] Jayashree Srinivasan, Angelo Castellino, Erin K. Bradley, John E. Eksterowicz, Peter D. J. Grootenhuis, Santosh Putta, and Robert V. Stanton. Evaluation of a novel shape-based computational filter for lead evolution: Application to thrombin inhibitors. *Journal of Medicinal Chemistry*, 45(12):2494–2500, 2002. PMID: 12036357.

[149] Richard A. Friesner, Jay L. Banks, Robert B. Murphy, Thomas A. Halgren, Jasna J. Klicic, Daniel T. Mainz, Matthew P. Repasky, Eric H. Knoll, Mee Shelley, Jason K. Perry, David E. Shaw, Perry Francis, and Peter S. Shenkin. Glide: A new approach for rapid, accurate docking and scoring. 1. method and assessment of docking accuracy. *Journal of Medicinal Chemistry*, 47(7):1739–1749, 2004. PMID: 15027865.

[150] René Thomsen and Mikael H Christensen. Moldock: a new technique for high-accuracy molecular docking. *Journal of medicinal chemistry*, 49(11):3315–3321, 2006.

[151] Gareth Jones, Peter Willett, Robert C Glen, Andrew R Leach, and Robin Taylor. Development and validation of a genetic algorithm for flexible docking. *Journal of molecular biology*, 267(3):727–748, 1997.

[152] Ming Liu and Shaomeng Wang. Mcdock: a monte carlo simulation approach to the molecular docking problem. *Journal of computer-aided molecular design*, 13(5):435–451, 1999.

[153] Fan Jiang and Sung-Hou Kim. "soft docking": matching of molecular surface cubes. *Journal of molecular biology*, 219(1):79–102, 1991.

[154] P Nuno Palma, Ludwig Krippahl, John E Wampler, and José JG Moura. Bigger: a new (soft) docking algorithm for predicting protein interactions. *Proteins: Structure, Function, and Bioinformatics*, 39(4):372–384, 2000.

[155] Todd JA Ewing, Shingo Makino, A Geoffrey Skillman, and Irwin D Kuntz. Dock 4.0: search strategies for automated molecular docking of flexible molecule databases. *Journal of computer-aided molecular design*, 15(5):411–428, 2001.

[156] Bernd Kramer, Matthias Rarey, and Thomas Lengauer. Evaluation of the flexx incremental construction algorithm for protein-ligand docking. *Proteins: Structure, Function, and Bioinformatics*, 37(2):228–241, 1999.

[157] Ingo Schellhammer and Matthias Rarey. Flexx-scan: Fast, structure-based virtual screening. *PROTEINS: Structure, Function, and Bioinformatics*, 57(3):504–517, 2004.

[158] Ajay N Jain. Surflex-dock 2.1: robust performance from ligand energetic modeling, ring flexibility, and knowledge-based search. *Journal of computer-aided molecular design*, 21(5):281–306, 2007.

[159] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.

[160] TJ Fuller-Rowell. A two-dimensional, high-resolution, nested-grid model of the thermosphere: 1. neutral response to an electric field "spike". *Journal of Geophysical Research: Space Physics*, 89(A5):2971–2990, 1984.

[161] TJ Fuller-Rowell. A two-dimensional, high-resolution, nested-grid model of the thermosphere: 2. response of the thermosphere to narrow and broad electrodynamic features. *Journal of Geophysical Research: Space Physics*, 90(A7):6567–6586, 1985.

[162] Wenbin Wang, Tim L Killeen, Alan G Burns, and Raymond G Roble. A high-resolution, three-dimensional, time dependent, nested grid model of the coupled thermosphere–ionosphere. *Journal of Atmospheric and Solar-Terrestrial Physics*, 61(5):385–397, 1999.

[163] Lie-Yauw Oey and Ping Chen. A nested-grid ocean model: With application to the simulation of meanders and eddies in the norwegian coastal current. *Journal of Geophysical Research: Oceans*, 97(C12):20063–20086, 1992.

[164] Michael S. Fox-Rabinovitz, Lawrence L. Takacs, Ravi C. Govindaraju, and Max J. Suarez. A variable-resolution stretched-grid general circulation model: Regional climate simulation. *Monthly Weather Review*, 129(3):453–469, 2001.

[165] Paul A Ullrich and Christiane Jablonowski. An analysis of 1d finite-volume methods for geophysical problems on refined grids. *Journal of Computational Physics*, 230(3):706–725, 2011.

[166] Cass Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[167] FÃ¡bio F. Bernardon, Christian A. Pagot, JoÃ£o L. D. Comba, and ClÃ¡udio T. Silva. Gpu-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, 11(4):1–16, 2006.

[168] Cheng-Kai Chen, Chris Ho, Carlos Correa, Kwan-Liu Ma, and Ahmed Elgamal. Visualizing 3d earthquake simulation data. *Computing in Science & Engineering*, 13(6):52–63, 2011.

[169] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[170] G Patrick Brady and Pieter FW Stouten. Fast prediction and visualization of protein binding pockets with pass. *Journal of computer-aided molecular design*, 14(4):383–401, 2000.

[171] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Res*, 28:235–242, 2000.

[172] Cesare Alippi, Giacomo Boracchi, and Manuel Roveri. Change detection tests using the ici rule. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–7. IEEE, 2010.