Chapter 7
Dynamic Application Autotuning for Self-aware Approximate Computing

Davide Gadioli

Abstract The energy consumption limits the application performance in a wide range of scenarios, ranging from embedded to High-Performance Computing. To improve computation efficiency, this Chapter focuses on a software-level methodology to enhance a target application with an adaptive layer that provides self-optimization capabilities. We evaluated the benefits of dynamic autotuning in three case studies: a probabilistic time-dependent routing application from a navigation system, a molecular docking application to perform virtual-screening, and a stereo-matching application to compute the depth of a three-dimensional scene. Experimental results show how it is possible to improve computation efficiency by adapting reactively and proactively.

7.1 Introduction

The increasing demand for computation power shifted the optimization focus towards efficiency in a wide range of energy-constrained systems, from embedded platforms to High-Performance Computing (HPC) [1]. A promising way to improve energy efficiency is approximate computing [2], which aims at finding a *good enough* solution, avoiding the unnecessary computation effort. It is possible to approximate the computation at different levels: from approximate hardware [3] to software techniques such as loop perforation [4]. Moreover, a large class of applications exposes software parameters that define an accuracy-throughput trade-off, especially in the multimedia field [5]. In this chapter, we focus at the software-level, where an application exposes *software-knobs* [6] that can alter its extra-functional behaviour. In this context, the definition of application performance includes several extra-functional properties (EFPs) in conflict to each other.

The value of an EFP might depend on the underlying architecture configuration, the system workload, and the features of the current input. Since this information usually changes at runtime, it is not trivial to find a one-fits-all configuration of

D. Gadioli (✉)
Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milan, Italy
e-mail: davide.gadioli@polimi.it

© The Author(s) 2020

B. Pernici (ed.), *Special Topics in Information Technology*, PoliMI SpringerBriefs,
https://doi.org/10.1007/978-3-030-32094-2_7

the application software-knobs at design-time. This is a well-known problem in the autonomic computing field [7], where researchers investigate approaches to provide self-optimization capabilities to a target application, for identifying and seizing optimization opportunities at runtime.

In this context, we propose a methodology to enhance an application with an adaptation layer that exposes reactive and proactive mechanisms to provide continuously the most suitable software-knobs configuration according to application requirements. The decision process is based on the application knowledge, which describes the relationship between software-knob configurations and EFPs. It is possible to learn the application knowledge either at design-time, by using well-known Design Space Exploration techniques [8], or at runtime by using an external component that coordinates a distributed DSE [9]. The benefits of the latter approach are the following: (1) the application can observe its behaviour with the same execution environment of the production run, (2) it can leverage features of the production input, and in the context of a distributed computation, (3) it can leverage the number of application instances to lower the learning time.

We evaluate the benefits of the methodology implementation, named *mARGOt*, in three real-world case studies. In particular, we use a stereo-matching application [5] to assess the benefits of reactive adaptation mechanisms, with respect to changes of both application requirements and performance. We use a probabilistic time-dependent routing stage in a navigation car system [10] to evaluate the benefits of the proactive adaptation mechanisms. Finally, we use a molecular docking application for virtual screening, to assess the benefits of learning the application knowledge at runtime.

The remainder of the Chapter is organized as follows. First, Sect. 7.2 provides an overview of autonomic computing, focusing on application autotuning and highlighting the main contributions of the proposed methodology. Then, in Sect. 7.3, we formalize the problem and describe the *mARGOt* framework. Section 7.4 discusses the benefits of the proposed methodology. Finally, Sect. 7.5 concludes the chapter.

7.2 Autonomic Computing and Application Autotuning

In the context of autonomic computing [7], we perceive a computing system as an ensemble of autonomous elements capable of self-management. The main idea is to enable the system to perform autonomously a task which is traditionally assigned to a human, to cope with the increasing complexity of computation platforms and applications. For example, if the system is able to incorporate new components whenever they become available, the system has the self-configuration ability. To qualify for the self-management ability, a system must satisfy all the related self-* properties. How to provide these properties is still an open question and previous surveys [11, 12] summarize the research effort in this area.

In this chapter, we focus on the self-optimization property at the software-level, which is the ability to identify and seize optimization opportunities at runtime. The methodologies to provide self-optimization properties are also known in the literature

as *autotuners*. It is possible to categorize autotuners in two main categories: static and dynamic.

Static autotuners aim at exploring a large space of software-knobs configuration space to find the most suitable software-knob configuration, assuming a predictable execution environment and targeting software-knobs that are loosely input-dependent. Among static autotuners, we might consider the following works. AutoTune [13] targets multi-node applications and it leverages the Periscope framework [14] to measure the execution time. It targets application-agnostic parameters exposed by the computation pipeline such as communication buffers and OpenHMPP/MPI parameters. QuickStep [15] and Paraprox [16] perform code transformations to automatically apply approximation techniques for enabling and leveraging an accuracy-throughput trade-off. OpenTuner [17] and the ATF framework [18] explicitly address the exponential growth in the complexity of exploring the parameters space, by using an ensemble of DSE techniques and by taking into account dependencies between software-knobs. Although very interesting, these approaches work at design-time and they usually target a different set of software-knobs with respect to dynamic autotuners.

The methodology proposed in this chapter belongs to the category of dynamic autotuners, which aim at changing the software-knobs configuration during the application runtime according to the system evolution. Therefore, they focus on providing adaptation mechanisms, typically based on application knowledge. Among dynamic autotuners, we might consider the following works. The Green framework [19] and PowerDial [6] enhance an application with a reactive adaptation layer, to change the software-knob configurations according to a change on the observed behaviour. The IRA framework [20] and Capri [21] focus instead on providing proactive adaptation mechanisms to select the software-knob configuration according to the features of the current input. On the other hand, Petabricks [22] and Anytime Automaton [23] are capable to adapt the application reactively and proactively. However, they require a significant integration effort from the application developers. Moreover, they are capable to leverage only accuracy-throughput trade-off. All these previous works are interesting and they have significantly contributed to the field, according to their approach on how to provide the self-optimization capabilities. The main contributions of *mARGOt* is to provide a single methodology to provide an adaptation layer with the following characteristics:

- The flexibility to express the application requirements as a constrained multi-objective optimization problem, addressing an arbitrary number of EFPs and software-knobs. Moreover, the application might change the requirements at runtime.
- The capability to adapt the application reactively and proactively, where the user might observe the application behaviour continuously, periodically or sporadically. Moreover, the reaction policies are not only related to the throughput, but they are agnostic about the observed EFP.
- To minimize the integration effort, we employ the concept of separation of concerns. In particular, application developers define extra-functional aspects in a

configuration file and the methodology is capable to generate an easy-to-use interface to wrap the target region of code.

Furthermore, being possible to define the application knowledge at runtime, *mARGOt* can use an external component to orchestrate a distributed DSE at runtime, during the production phase.

7.3 The mARGOt Autotuning Framework

This section describes how the proposed methodology can enhance the target application with an adaptation layer. At first, we formalize the problem, the application requirements and how the *mARGOt* interacts with the application. Then, we describe the main components of the framework and how they can adapt the application reactively and proactively.

7.3.1 Problem Definition

Figure 7.1 shows the overview of the *mARGOt* and how it interacts with an application. To simplify, we assume that the application is composed of a single computation kernel g that reads a stream of input i to produce the required stream of output o. However, *mARGOt* is capable to manage several regions of code independently. Moreover, the target kernel exposes a set software-knobs $\overline{x}$ that alter the extra-functional behaviour. Since a change of the configuration of these knobs might lead to a different quality of the results, we might define the application as $o = g(i, \overline{x})$.

Given this definition, the application requirements are expressed as a constrained multi-objective optimization problem described in Eq. 7.1:

$$\begin{aligned} \max(\min)\ & r(\overline{x}; \overline{m} \mid \overline{f}) \\ \text{s.t. } & C_1 : \omega_1(\overline{x}; \overline{m} \mid \overline{f}) \ \propto \ k_1 \ \ with\ \alpha_1\ confidence \\ & C_2 : \omega_2(\overline{x}; \overline{m} \mid \overline{f}) \ \propto \ k_2 \\ & \dots \\ & C_n : \omega_n(\overline{x}; \overline{m} \mid \overline{f}) \ \propto \ k_n \end{aligned} \tag{7.1}$$

where r denotes the objective function to maximize (minimize), $\overline{m}$ is the vector of metrics of interest (i.e. the EFPs), and $\overline{f}$ is the vector of input features. Let C be the set of constraints, where each constraint C_i is expressed as the function ω_i over $\overline{m}$ or $\overline{x}$, that must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$, with a confidence α_i, if ω_i targets a statistical variable. If the application is input-dependent, ω_i and r depend on its features.

In this context, the goal of the autotuner is to solve the optimization problem by inspection, using the application knowledge. The application always needs a

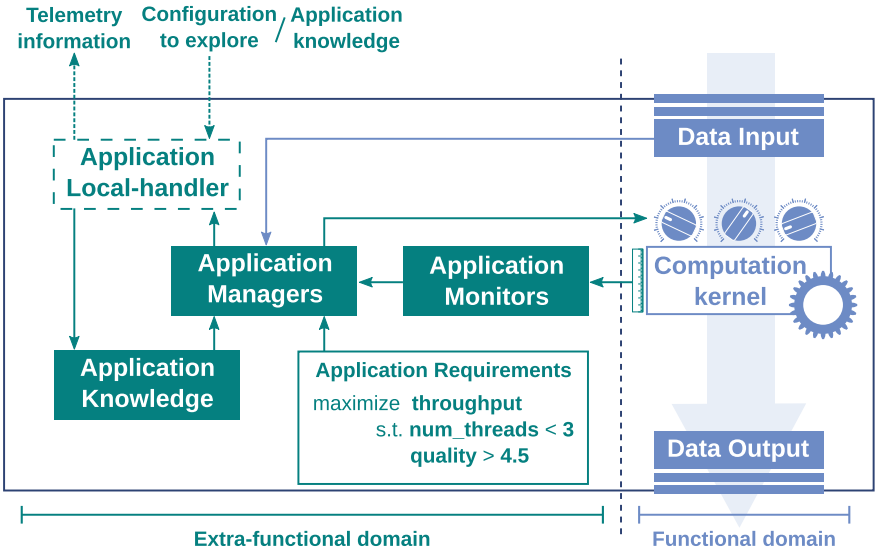


Fig. 7.1 The overview of the proposed autotuning framework. Green elements represent framework components, while blue elements represent application components (Color figure online)

software-knob configuration, therefore if the problem is unfeasible, *mARGOt* can relax the constraints, according to their priority, until it finds a valid solution.

We implemented *mARGOt* as a standard C++ library that should be linked to the target application. Being the time spent by the framework to select the most suitable software-knob configuration stolen from the application, the *mARGOt* implementation has been designed to be lightweight and to minimize the introduced overhead. We publicly released the framework source code [24].

7.3.2 Application-Knowledge

To solve the optimization problem we need a model of the application behaviour. However, the relationship between software-knobs, EFPs, and input features is complex and unknown. Therefore, *mARGOt* defines the application-knowledge as a list of *Operating Points* (OPs). Each OP θ corresponds to a software-knob configuration, together with the reached EFPs. If the application is input-dependent, the OP includes also the information on the related input features: $\theta = \{\overline{x}, \overline{m}, \overline{f}\}$.

This representation has been chosen for the following reasons: it provides a high degree of flexibility to describe the application behaviour, *mARGOt* can solve efficiently the optimization problem described in Eq. 7.1 and it prevents the possibility to select an invalid software-knob configuration.

The application-knowledge is considered an input of *mARGOt*, and there are several tools that can explore the Design Space efficiently. In particular, *mARGOt* uses the XML format of Multicube Explorer [25] to represent the application-knowledge. Moreover, it is possible to learn it at runtime, as described in Sect. 7.3.4.

7.3.3 Interaction with the Application

The core component of *mARGOt* is the application manager, which is in charge of solving the optimization problem and of providing to the application the most suitable software-knob configuration. The application is able to change the application requirements at runtime according to the system evolution. Moreover, if the EFPs are input-dependent, the application can provide features of the actual input to adapt proactively [9].

To enable the reactive adaptation, *mARGOt* must observe the actual behaviour of the application and compare it with the expected one. In particular, we compute a coefficient error for each observed EFP as $e_{m_i} = \frac{expected_i}{observed_i}$, where e_{m_i} is the error coefficient for the i-th EFP. Under the assumption of linear error propagation among the OPs, *mARGOt* is able to adapt reactively.

The methodology implementation provides to application developers a suite of monitors to observe the most common metrics of interest, such as throughput or performance events using PAPI [26]. The customization of a monitor to observe an application-specific EFP, such as the accuracy, is straightforward.

7.3.4 Runtime Application-Knowledge Learner

The *mARGOt* flexibility enables the possibility to change the application knowledge at runtime. Therefore, it is possible to learn the application knowledge directly at runtime, during the production phase. Although this approach is platform independent, it focuses on the High-Performance Computing Scenario. The idea is to perform a distributed Design Space Exploration, leveraging the parallelism level of the platform to lower the exploration time. In particular, when we spawn an application, it notifies its existence to a central coordinator, along with information about the exploration, such as the software-knobs domains, and the list of EFPs. According to a Design of Experiments, the central coordinator dispatches software-knobs configuration to evaluate at each application instance, which provides as feed-back telemetry information. Once the central coordinator collects the required observations, it uses learning techniques to derive the application-knowledge to broadcast to the application instances.

Figure 7.2 shows an overview of the central coordinator. In particular, it uses a thread pool of *application remote-handlers* to interact with *application local-handler* of the application instances. The communication uses the lightweight MQTT or

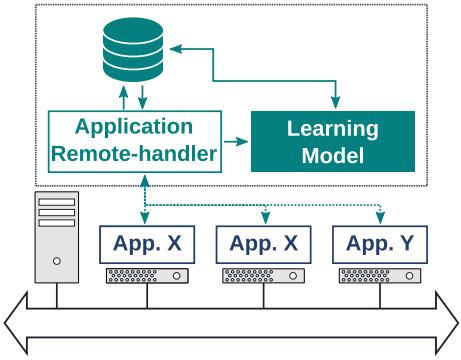


Fig. 7.2 The proposed approach to perform a distributed on-line Design Space Exploration, using a dedicated server outside of the computation node

MQTTs protocols, while we use the Cassandra database to store the required information. The learning module leverages a well-known approach [27] to interpolate application performance, implemented by the state-of-the-art R package [28].

7.4 Experimental Results

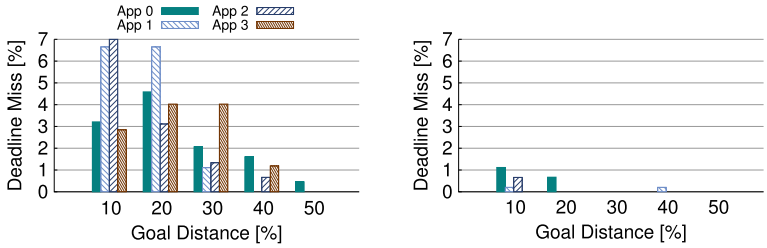
This section assesses the benefits of the proposed adaptation layer, by deploying the *mARGOt* framework in three different case studies. Each case study highlights a different characteristic of *mARGOt*.

7.4.1 Evaluation of the Reactive Adaptation

This experiment aims at assessing the benefits of the reactive adaptation mechanisms. We focus on a Stereo-matching application deployed in a quad-core architecture [29]. The application takes as input a pair of stereo images of the same scene and it computes the disparity map as output. The algorithm exposes application-specific software-knobs that define an accuracy-throughput trade-off. Moreover, it is possible to change the number of software threads that the application can use to carry out the computation.

We create the application-knowledge at design-time, evaluating each configuration in isolation. In this experiment, we execute four instances of stereo-matching, overlapping their execution: each application has an execution time of 200 s, and we spawn them with a delay of 50 s between each other. The application designer would like to maximize the accuracy of the elaboration, provided that the application must sustain a throughput of 2 fps.

In this experimental setup, we compare two adaptation strategies. On one hand, we consider as *baseline* the reaction policy that monitors the throughput of the



(a) Baseline approach (b) *mARGOt* approach

Fig. 7.3 Distribution of deadline misses with respect to the constraint on the throughput, according to the distance from the target value (2 fps)

application and that reacts accordingly, since it is commonly used in literature. In particular, if the monitors observe that the actual throughput of the application is lower than the expected one, *mARGOt* will choose a configuration with an higher expected throughput to compensate. On the other hand, we exploit the *mARGOt* flexibility to consider also the CPU usage in the requirements. In particular, we define a constraint on the available resource usage on top of the one on the throughput, to limit the number of software threads according to the available resources. The value of the constraint is continuously updated at runtime as follows:

$$CPU_{available} = \Gamma - \gamma + \pi_{measured}$$

where Γ is the maximum CPU quota available in the platform, γ is the monitored CPU quota used by the system, and $\pi_{measured}$ is the monitored CPU quota assigned to the application by the Operating System. The *mARGOt* capability to consider an arbitrary number of EFPs enables this adaptation strategy.

From the experimental results, we can observe how the two strategies are capable to satisfy the application requirements on average. However, Fig. 7.3 shows the distribution of the deadline misses for the two strategies. The baseline strategy relies on the scheduler for a fair assignment of the CPU quota, therefore the contention on the resources reduces the predictability of the throughput. Conversely, the *mARGOt* approach avoids this contention by observing the CPU usage. However, we are not able to guarantee a fair allocation of the CPU quota by using this approach.

7.4.2 Evaluation of the Proactive Adaptation

This experiment aims at assessing the benefits of the proactive adaptation mechanisms. We focus on a phase of a car navigation system: the probabilistic time-dependent routing (PTDR) application [10]. It takes as input the starting time of the travel and the speed profiles of all the segments that compose the target path. The

Table 7.1 Number of independent route traversal simulations by varying the requested maximum error and the statistical properties of interest

Approach	Error (%)	Simulations 50th percentile	Simulations 75th percentile	Simulations 95th percentile
Baseline	3	1000	3000	3000
	6	300	1000	1000
Adaptive	3	632	754	1131
	6	153	186	283

speed profiles of a segment vary during the week, with a fifteen minutes granularity. To estimate the arrival time distribution, the PTDR application uses a Monte Carlo approach that simulates multiple times an independent route traversal. The output of the application is a statistical property of the arrival time distribution such as the 50th or 75th percentile. This application has been already optimized to leverage the target HPC node [30], therefore it exposes as software-knob the number of route traversal simulations.

The application designer would like to minimize the number of route traversal simulations, given that the difference between the value computed with the selected configuration and with 1 M samples are below a given threshold. The threshold value might vary according to the type of user that generates the request. In this experiment, we consider 3% for premium users and 6% for free users.

To evaluate the benefits of the proactive adaptation, we compare two adaptation strategies. On one hand, we fix the number of samples according to the worst path in a representative input set [30]. We use this strategy as a baseline. On the other hand, we use an adaptive strategy that extracts a feature from the actual input to select the number of simulations according to the target path [10].

Table 7.1 shows the experimental results of a large experimental campaign with randomly selected pairs of Czech Republic routes and starting times. The adaptive approach can significantly reduce the number of required simulations according to the target statistical property and the maximum error level. Moreover, we modelled the car navigation system with the simulation environment Java Modeling Tools to measure the benefits of the adaptive strategy at the system level. With a load of 100 k requests every 2 min, an error threshold of 6%, and assuming that we are interested in the 95th percentile, the adaptive strategy can reduce the number of nodes by 26%. These parameters estimate the requested generated by a Smart City with the size of the Milan urban area.

7.4.3 Evaluation of the Runtime Learner

This experiment aims at evaluating the benefits of learning the application knowledge at runtime. In the context of the early stages of the drug discovery process, we focus

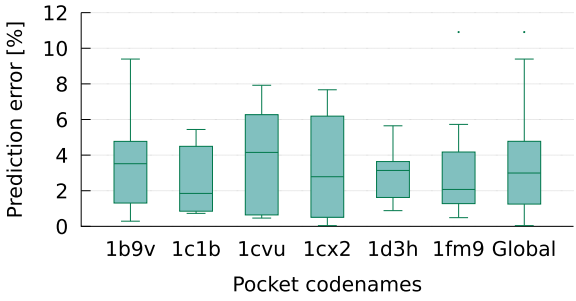


Fig. 7.4 Distribution of the prediction error on the time-to-solution, by varying the target pocket

on a molecular docking application for virtual screening [31]. It takes as input two information. On one hand, the docking site of the target molecule, named *pocket*. On the other hand, a huge library of possible solutions, named *ligands*. The output of the application is a small set of ligands which may have a strong interaction with the target pocket, to forward to the later stages of the process. The application exposes application-specific software-knobs that expose an accuracy-throughput trade-off.

Due to the complexity of estimating a ligand-pocket interaction, and due to the embarrassingly parallel nature of the problem, this application is a perfect match for HPC computation. In this scenario, the application designer would like to maximize the quality of the elaboration, given an upper bound on the time-to-solution. The relationship between the throughput and the software-knobs configuration depends on the characteristic of the actual input, especially of the pocket. Therefore, in this experiment we use *mARGOt* to learn at runtime such relationship, to be exploited in the remainder of the production run.

Figure 7.4 shows the distributions of the prediction error on the time-to-solution with six pocket from the RCSB Protein Databank (PDB) [32]. We use a chemical library with heterogeneous ligands [9]. For example, the number of their atoms range from 28 to 153. Since the prediction error is limited ($<10\%$), *mARGOt* is able to improve the computation efficiency.

7.5 Conclusion

This chapter focuses on a methodology to enhance the target application with an adaptation layer, based on the application-knowledge, that provides the most suitable software-knobs configuration according to the application requirements.

We assessed the benefits of *mARGOt* in three case studies that belong to different application domains. Experimental results show how it is possible to improve drastically the computation efficiency by adapting reactively and proactively. Moreover, it is possible to learn the relationship between EFPs, software-knobs, and input features using the input of the production run, identifying and seizing optimization opportunities.

References

1. Duranton M, De Bosschere K, Coppens B, Gamrat C, Gray M, Munk H, Ozer E, Varganega T, Zendra O (2019) Hipeac vision 2018
2. Sasa M, Stelios S, Henry H, Martin R (2010) Quality of service profiling. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 1. ACM, pp 25–34
3. Hadi E, Adrian S, Luis C, Doug B (2012) Neural acceleration for general-purpose approximate programs. In: Proceedings of the 2012 45th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 449–460
4. Henry H, Sasa M, Stelios S, Anant A, Martin R (2009) Using code perforation to improve performance, reduce energy consumption, and respond to failures
5. Edoardo P, Davide G, Gianluca P, Vittorio Z, Cristina S (2014) Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive openCL applications. In: Application-specific Systems, architectures and processors (ASAP). IEEE, pp 161–168
6. Henry H, Stelios S, Michael C, Sasa M, Anant A, Martin R (2011) Dynamic knobs for responsive power-aware computing. In: ACM SIGPLAN notices, vol 46. ACM, pp 199–212
7. Jeffrey O Kephart and David M Chess (2003) The vision of autonomic computing. Computer 36(1):41–50
8. Bergstra J, Pinto N, Cox D (2012) Machine learning for predictive auto-tuning with boosted regression trees. In: 2012 innovative parallel computing (InPar), pp 1–9, May 2012
9. Gadioli D, Vitali E, Palermo G, Silvano C (2018) Margot: a dynamic autotuning framework for self-aware approximate computing. IEEE transactions on computers
10. Emanuele V, Davide G, Gianluca P, Martin G, João B, Pedro P, Jan M, Kateřina S, João MPC, Cristina S (2019) An efficient monte carlo-based probabilistic time-dependent routing calculation targeting a server-side car navigation system. IEEE transactions on emerging topics in computing
11. Markus CH, Julie AMcC (2008) A survey of autonomic computing—degrees, models, and applications. ACM Comput Surv (CSUR) 40(3):7
12. Sara M-H, Vinicius HSD, Danny W, Paris A (2017) A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. Inf Softw Technol 90:1–26
13. Renato M, Gilles C, Anna S, Eduardo C, Michael G, Houssam H, Carmen N, Siegfried B, Martin S, Laurent M et al (2012) Autotune: a plugin-driven approach to the automatic tuning of parallel applications. In: International workshop on applied parallel computing. Springer, pp 328–342
14. Shajulin B, Ventsislav P, Michael G (2010) Periscope: an online-based distributed performance analysis tool. In: Tools for high performance computing 2009. Springer, pp 1–16
15. Misailovic S, Kim D, Rinard M (2013) Parallelizing sequential programs with statistical accuracy tests. ACM Trans Embed Comput Syst (TECS) 12(2s):88
16. Mehrzad S, Davoud AJ, Janghaeng L, Scott M (2014) Paraprox: pattern-based approximation for data parallel applications. ACM SIGPLAN Not 49(4):35–50
17. Jason A, Shoaib K, Kalyan V, Jonathan R-K, Jeffrey B, Una-May O, Saman A (2014) Opentuner: an extensible framework for program autotuning. In: 2014 23rd international conference on parallel architecture and compilation techniques (PACT). IEEE, pp 303–315
18. Ari R, Michael H, Sergei G. Atf: a generic auto-tuning framework. In IEEE 19th international conference on high performance computing and communications; IEEE 15th international conference on smart city; IEEE 3rd international conference on data science and systems (HPCC/SmartCity/DSS). IEEE, pp 64–71
19. Woongki B, Trishul MC (2010) Green: a framework for supporting energy-conscious programming using controlled approximation. In: ACM Sigplan Notices, vol 45. ACM, pp 198–209
20. Michael AL, Parker H, Mehrzad S, Scott M, Jason M, Lingjia T (2016) Input responsiveness: using canary inputs to dynamically steer approximation. ACM SIGPLAN Not 51(6):161–176

21. Xin S, Andrew L, Donald SF, Keshav P (2016) Proactive control of approximate programs. ACM SIGOPS Oper Syst Rev 50(2):607–621
22. Yufei D, Jason A, Kalyan V, Xipeng S, Una-May O, Saman A (2015) Autotuning algorithmic choice for input sensitivity. In: ACM SIGPLAN notices, vol 50. ACM, pp 379–390
23. Joshua SM, Natalie EJ (2016) The anytime automaton. In: ACM SIGARCH computer architecture news, vol 44. IEEE Press, pp 545–557
24. mARGOt framework git repository (2018). https://gitlab.com/margot_project/core
25. Vittorio Z, Gianluca P, Fabrizio C, Cristina S, Giovanni M (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: 23th international conference on architecture of computing systems 2010. VDE, pp 1–7
26. Dan T, Heike J, Haihang Y, Jack D (2010) Collecting performance data with papi-c. In: Matthias SM, Michael MR, Alexander S, Wolfgang EN (eds) Tools for high performance computing 2009, Berlin, Heidelberg, 2010. Springer, Berlin, Heidelberg, pp 157–173
27. Benjamin CL, David MB (2006) Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: ACM SIGOPS operating systems review, vol 40. ACM, pp 185–194
28. Zhenghua N, Jeffrey SR (2012) The crs package: nonparametric regression splines for continuous and categorical predictors. R J 4(2)
29. Davide G, Gianluca P, Cristina S (2015) Application autotuning to support runtime adaptivity in multicore architectures. In: 2015 international conference on embedded computer systems: architectures, modeling, and simulation (SAMOS). IEEE, pp 173–180
30. Radek T, Lukáš R, Jan M, Kateřina S, Ivo V (2015) Probabilistic time-dependent travel time computation using monte carlo simulation. In: International conference on high performance computing in science and engineering. Springer, pp 161–170
31. Claudia B, Andrea RB, Carlo C, Simone L, Gabriele C (2013) Use of experimental design to optimize docking performance: the case of ligendock, the docking module of ligen, a new de novo design program
32. Helen MB, John W, Zukang F, Gary G, Bhat TN, Helge W, Ilya NS, Philip EB (2000) The protein data bank. Nucleic Acids Res 28:235–242

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.