Scientific Programming 22 (2014) 285–297
DOI 10.3233/SPR-140394
IOS Press

From single- to multi-objective auto-tuning of programs: Advantages and implications

Juan Durillo [*] and Thomas Fahringer
Institute of Computer Science, University of Innsbruck, Innsbruck, Austria
E-mails: {juan, tf}@dps.uibk.ac.at

Abstract. Automatic tuning (auto-tuning) of software has emerged in recent years as a promising method that tries to automatically adapt the behaviour of a program to attain different performance objectives on a given computing system. This method is gaining momentum due to the increasing complexity of modern multicore-based hardware architectures. Many solutions to auto-tuning have been explored ranging from simple random search to more sophisticate methods like machine learning or evolutionary search. To this day, it is still unclear whether these approaches are general enough to encompass all the complexities of the problem (e.g. search space, parameters influencing the search space, input data sensitivity, etc.), or which approach is best suited for a given problem. Furthermore, the growing interest in auto-tuning a program for several objectives is increasing this confusion even further. The goal of this paper is to formally describe the problem addressed by auto-tuning programs and review existing solutions highlighting the advantages and drawbacks of different techniques for single-objective as well as multi-objective auto-tuning approaches.

Keywords: Program auto-tuning, single-objective optimization, multi-objective optimization

1. Introduction

Over the last decade we have observed major changes in computing systems [11]. The penetration of multi- and many-core processors in all segments of the computing market has implied a programming paradigm shift from serial to parallel computing. The dominance of desktops, laptops and server PCs has faded and has been replaced by new computing systems of smart embedded systems, mobile devices and Cloud infrastructures. The rise of heterogeneous parallel computers where parts of a computing device is shut off to stay within a power limit, has been among others caused by the profound effect of energy consumption on computing systems. Cyber physical systems are becoming more and more popular, where data can be gathered, for instance, from users or embedded systems, processed on Cloud computers, and results are delivered to users through their mobile devices. As a consequence of this wide spectrum of different computing systems, optimizing an application for performance has become a challenging task. Applications optimized for one computing system may not be optimal for another, thus optimization of programs commonly has to be repeated for every computing system. In the last years, a new optimization technique known as *software auto-tuning* or simply *auto-tuning* has emerged to deal with this challenge. The idea of auto-tuning is to automatically generate or modify programs to optimize for one or several non-functional behavior(s) on a given target architecture. Auto-tuning has been implemented as part of programming environments, compilers, operating systems or libraries.

Auto-tuning is commonly applied to optimize programs for very specific objectives. For instance, programs on small-scale embedded systems are being optimized for program size and memory footprint. Energy consumption or temperature is important to be minimized for mobile systems. Execution times and power limits are first order goals for applications on high performance or supercomputing systems. Economic costs are of paramount interest for programs that are running on Cloud infrastructures. Time and safety are crucial for critical and mixed critical systems, and security for desktop computers connected to the Internet. Few systems are sophisticated enough to simultaneously deal with two or more objectives. Providing auto-tuning methods that can deal with multiple objectives at the same time becomes more and more important as different non-functional behavior such as execution time, economic costs, security, energy consump-

[*]Corresponding author. E-mail: juan@dps.uibk.ac.at.

tion and others may all be important with some kind of priority for an increasing number of domains in science, engineering and commerce.

Although the number of works in the field of auto-tuning has considerably increased, there is a lack of studies that systematically elaborate and compare auto-tuning techniques for optimizing programs. The goal of this paper is to define the software auto-tuning problem and to describe what optimization techniques can be applied for solving a single- and multi-objective optimization problem, what are their advantages and drawbacks.

Outline. This paper is structured as follows. The next section formally presents the problem of automatic software tuning and different mechanisms to modify the non-functional program behavior. Section 3 describes different approaches for single-objective auto-tuning; Section 4 is devoted to the multi-objective optimization problem. In Section 5, we include an empirical comparison of several single and multi-objective auto-tuners. Finally, Section 6 indicates the main advantages and drawbacks of some of the most important software auto-tuning methods, as well as some guidelines for future work.

2. Automatic tuning of programs

In [27], the term Software Automatic Tuning is defined as a technology intended to automatically adapt the execution of a program P to attain optimal non-functional behavior (e.g. execution time) on a given computing system.

The process of software adaptation can be realized in two basic non-orthogonal ways. Firstly, we can tune P without modifying its code by finding the optimal configuration of the computing system on which P will be executed or the runtime system which is linked with P. Examples of system parameters that influence the execution time of a program are the number of threads, the affinity or mapping of threads onto physical cores, the frequency at which the core is clocked (tunable via DVFS [18]), or the work group/grid size in the case of a GPU application written in OpenCL/Cuda. A program P that is executed on a specific computing system can be parameterized with a set of tunable system parameters $P(x_{s_1}, \dots, x_{s_n})$, where $x_{s_1}, \dots, x_{s_n}$ are defined prior to the execution of P.

Secondly, we can tune a program P by generating a semantically equivalent but syntactically different version of P by using transformation systems such as compiler or by applying them manually. This can be achieved by *code transformations* such as loop unrolling, blocking, tiling, software pipelining, data layout changes, to mention a few. Given a program P, a code transformation T produces a new program P' as output. We denote the operation of applying a code transformation as $P \otimes T \to P'$. Notice that the result of applying a code transformation is a new program to which successive code transformations can be applied. For example the application of two code transformations T_1 and T_2 would be denoted as $(P \otimes T_1) \otimes T_2$; obviously, the order in which these transformations are applied matters, since the second transformation is not applied to the original program P but to a modified version (P') of P. In most of the cases, code transformations expose some additional parameters which need to be set. We refer to them as transformation parameters. For example the loop unrolling code transformation exposes the unrolling factor (number of times the loop body should be replicated). Another parameter is the tile size for loop tiling. In these cases, a code transformation T can be denoted as $T(x_{p_1}, \dots, x_{p_m})$, where $x_{p_1}, \dots, x_{p_m}$ are the m program parameters exposed by T.

Given a program P and a computing system exposing the system parameters $x_{s_1}, \dots, x_{s_n}$, auto-tuning of P can be viewed as the process of determining the optimal values for these parameters, the optimal sequence of t successive transformations to be applied $T_1, \dots, T_t$ and the optimal values for these transformations parameters in order to attain the best value of a specific non-functional behavior f. Every execution of P with a specific value setting for system and program parameters as well as specific sequence of transformations refers to an alternative (solution) of P. To the best of our knowledge, no auto-tuning work simultaneously addressed the search for

- a sequence of transformations,
- the program parameters for these transformations, and
- the system parameters.

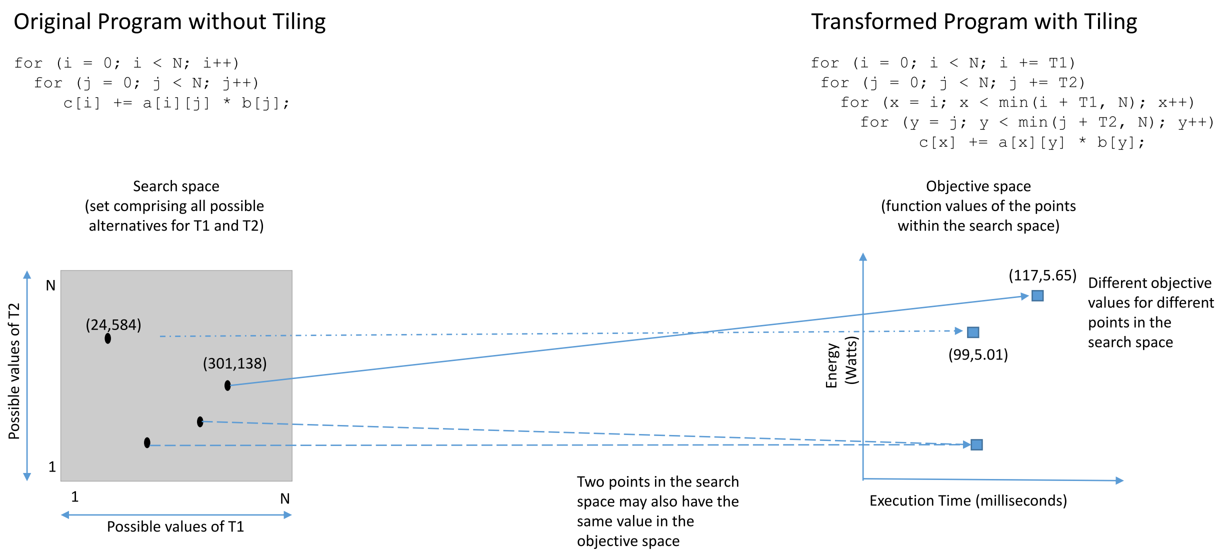
Instead existing work often tries to prune the search space, for example by searching only the system parameters or search only for an optimized sequence of transformations but ignoring program parameters, or searching only for program parameters without considering alternative transformation sequences, etc.

Different non-functional behaviors, such as execution time, energy consumption, reliability, or economic costs, can be the target of the optimization process. If our interest is to tune a program for several of these non-functional behaviors simultaneously, represented by a vector of optimization functions $\vec{f} = \langle f_1, \ldots, f_r \rangle$, then software auto-tuning becomes a multi-objective optimization problem, since many of these behaviors may conflict with each other. Many auto-tuning works do not consider auto-tuning as a multi-objective problem, and are limited to optimize a single objective at a time. In addition, many works considering several optimization goals, usually transform these objectives to a single objective function to be optimized.

Single- and multi-objective auto-tuning face an optimization problem, which can be defined in terms of a search: finding the alternative(s) optimizing the desired non-functional behavior(s). In every search it is important to distinguish between two terms: the search space, and the objective space. The search space defines the set comprising all possible alternatives (i.e., the domain of the function to be optimized). The objective space consists of the function values for $\vec{f}$ of the alternatives in the search space (i.e., the co-domain of the function to be optimized). The objective space is single-dimensional in the case of a single-objective auto-tuning problem, and r-dimensional in the case of the multi-objective auto-tuning problem where r is the number of objectives. Figure 1 depicts the search and objective spaces of an auto-tuning problem to find the best tile sizes for a matrix-vector multiplication code, where the goal is to minimize the execution time and the energy consumption of the code.

It should be noted that auto-tuning can be applied to the code regions $R_1, \ldots, R_q$ composing a program P or to the whole program P. In the first case, auto-tuning tries to find the system parameters and program transformations which optimize each of those code regions independently. In the second case, the system parameters and program transformations are searched in order to optimize the entire program. Notice that the solutions to both problems may be different since system parameters and transformations which optimize a single code region can negatively impact other regions and therefore the entire program.

Solving an auto-tuning problem is a challenging task due to several reasons. Firstly, there is no obvious solution to this problem due to complex and counter-intuitive relations between transformations and parameter values; and second, exhaustive search techniques are not feasible since it is not possible to evaluate all the possible value combinations of the computing system parameters, transformations and values of their exposed parameters within a reasonable amount of time. To overcome these problems, different techniques to prune and navigate the search space in a smart way have been proposed. Many of these techniques rely on generating different alternatives by assigning values to the system parameters or applying a sequence of transformations (with specific program parameters) to the input program P. These alternatives are then executed on the target computing system. For each evaluated alternative we measure the corresponding values for ev-


Original Program without Tiling

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
     c[i] += a[i][j] * b[j];
```

Transformed Program with Tiling

```
for (i = 0; i < N; i += T1)
 for (j = 0; j < N; j += T2)
   for (x = i; x < min(i + T1, N); x++)
     for (y = j; y < min(j + T2, N); y++)
           c[x] += a[x][y] * b[y];
```

Search space
(set comprising all possible alternatives for T1 and T2)
Possible values of T2
N
1
1
N
Possible values of T1
(24,584)
(301,138)
Two points in the search space may also have the same value in the objective space
Objective space
(function values of the points within the search space)
Energy (Watts)
Execution Time (milliseconds)
(117,5.65)
(99,5.01)
Different objective values for different points in the search space

Fig. 1. Search and objective spaces for a matrix-vector multiplication code example for different tile sizes. The goals are to minimize the execution time and energy consumption. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140394.)

ery function in $\vec{f}$ whose outcome is then used to decide which additional alternatives are to be generated. This procedure is known as *iterative compilation* [21] in the literature.

Auto-tuning of programs can be further classified as to when it is applied. Offline auto-tuning means that a program is tuned during an offline phase before the program is executed in "production mode" on the target computing system. In contrast, online auto-tuning refers to tuning the program while it is being executed in production mode. The latter approach is more challenging than the former mainly because of two reasons. Firstly, auto-tuning implies additional overhead that increases the overall execution time of a program. Secondly, selecting a poor performing alternative can deteriorate some values of $\vec{f}$ of a running program, which cannot be acceptable for some applications. Offline auto-tuning does not suffer by these problems. However, online approaches have an important advantage over offline auto-tuning, as they can tune a program for the current load of a computing systems which is difficult to consider for an offline tuning approach. In both cases, the suitability of a method for offline or online auto-tuning may depend on the number of times the code regions of a program (to be tuned) are executed. Only if these code regions are executed reasonably often during a single program run, then online auto-tuning can explore a large enough part of the search space to effectively optimize the program.

In the following sections we analyse different techniques for single- and multi-objective auto-tuning highlighting their main differences, advantages and disadvantages.

3. Techniques for software auto-tuning

In this section we describe different techniques for performing software auto-tuning. Figure 2 depicts different techniques that have been applied in this field. These techniques are presented in a hierarchy structure. The first level of the hierarchy distinguishes between techniques which always compute local or global optima (where applicable) and techniques which do not fulfil that goal. Among the former group of techniques, we separate exact techniques from approximation techniques. The first group assures to determinate the global optimal configuration. Techniques within this group are rarely applicable. On the other hand, approximation techniques only guarantee to compute a local optimal configuration of the program. This group is further divided in two classes: heuristic and metaheuristics. Heuristics are usually faster than metaheuristics but are problem dependent and can only be applied under certain restrictions; in other words, there are programs for which some heuristic methods cannot be applied. Metaheuristics may take longer than heuristics to compute a solution, but

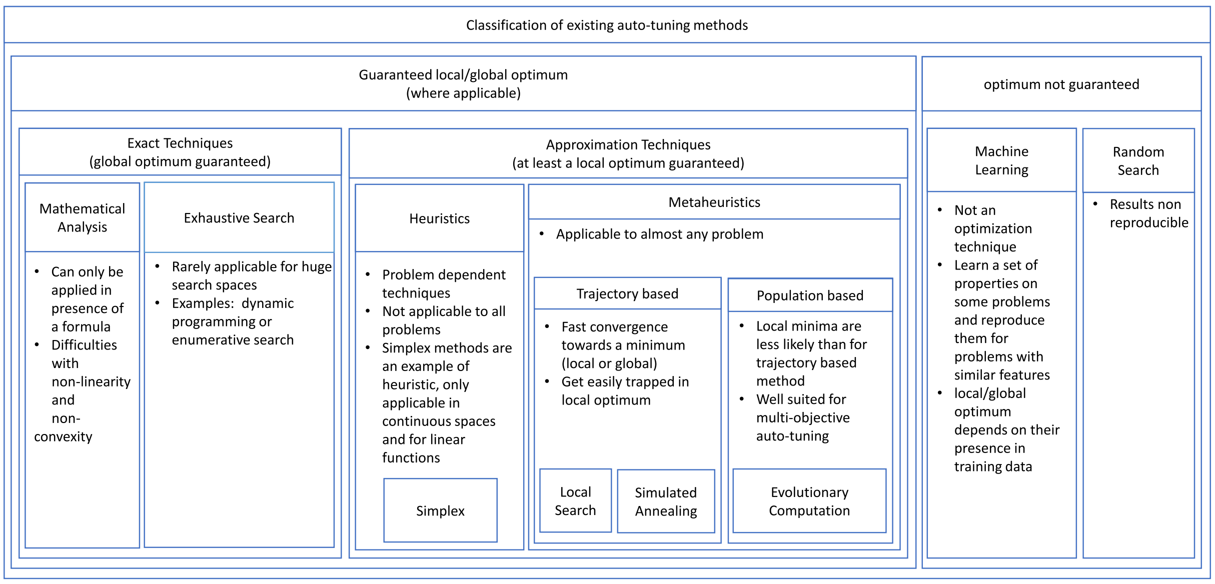

Classification of existing auto-tuning methods
Guaranteed local/global optimum (where applicable)
optimum not guaranteed
Exact Techniques (global optimum guaranteed)
Approximation Techniques (at least a local optimum guaranteed)
Machine Learning
Random Search
Mathematical Analysis
Exhaustive Search
Heuristics
Metaheuristics
• Applicable to almost any problem
• Can only be applied in presence of a formula
• Difficulties with non-linearity and non-convexity
• Rarely applicable for huge search spaces
• Examples: dynamic programming or enumerative search
• Problem dependent techniques
• Not applicable to all problems
• Simplex methods are an example of heuristic, only applicable in continuous spaces and for linear functions
Simplex
Trajectory based
• Fast convergence towards a minimum (local or global)
• Get easily trapped in local optimum
Local Search
Simulated Annealing
Population based
• Local minima are less likely than for trajectory based method
• Well suited for multi-objective auto-tuning
Evolutionary Computation
• Not an optimization technique
• Learn a set of properties on some problems and reproduce them for problems with similar features
• local/global optimum depends on their presence in training data
• Results non reproducible

Fig. 2. Hierarchical presentation of software auto-tuning techniques. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140394.)

do no impose hard restrictions on the problems to be solved.

In the following, for each of the approaches included in the figure we discuss their working principle, their main benefits and drawbacks. We do not claim to cover all possible single-objective optimization techniques but focus on some of the most prominent and effective techniques which have been considered previously in the auto-tuning field.

3.1. Mathematical analysis

Mathematical analysis is based on the idea of computing the first and second order derivatives of a function whose analytical formulation is known. That information is used to determine whether a function has an optimum value and whether this is a maximum or a minimum. Mathematical analysis is rarely applied for software auto-tuning. One of the main difficulties is the absence of a mathematical function modelling the behavior of the computing system in many cases. The computing system appears in these cases to the user as a black box where the input is a program and the output is the execution time (or another optimization criterion) of that program. In some cases, even if a formulation for these black boxes exist, the complexity governing the behaviour of a computer system components, like the memory hierarchy, would probably correspond to a non-linear functions or non-convex search spaces, making difficult the application of mathematical analysis in this context.

3.2. Exhaustive search

Exhaustive search consists in a systematic evaluation of every possible alternative within a search space and the selection of the best performing with respect to an optimization criterion. This approach renders its application impossible for many auto-tuning instances due to the size of the search space, since it may take even years to complete.

3.3. Local search or hill climbing methods

Local search techniques, such as the hill climbing method (see [6]), represent another option to solve optimization problems for which exhaustive search is impractical. Local search methods start with an alternative which is iteratively modified until no further improvements are possible. These modifications consist of small variations of the current alternative. An example of local search applied in the context of auto-tuning is described in [29], where transformations to a program are applied by setting compiler flags. The method described in that paper starts by applying all transformations to the input program (i.e., all the compilation flags are set). The method then removes the transformation (i.e., unset a flag), if any, that has the highest negative impact on the optimization criterion. Obviously, to determine the transformation with the highest negative impact, compiler flags are unset one after the other, the program is compiled and executed without that flag, and the resulting alternative has to be evaluated on the target computing system and compared. If the method does not find a transformation anymore that negatively affects a solution, then the current set of flags that are set is returned.

In theory, local search methods can be applied to determine the parameters of a computer system and/or the sequence of transformations to be applied together with their parameters. In practice, however, local search methods suffer from two major drawbacks: (1) the computed alternative highly depends on the chosen initial solution; and, (2) a huge search space may require evaluating a prohibitively large set of alternatives. These drawbacks are particularly critical in the context of auto-tuning of programs where interactions between different transformations and parameter values are expected to generate irregular and complex search spaces.

3.4. Simulated annealing

Simulated annealing [20] is basically an extension of local search techniques that reduces the probability of getting trapped in local minima. This technique still suffers from the aforementioned drawbacks (1) and (3) of the local search. In spite of this, simulated annealing has been used for auto-tuning of programs. For example, in [15], simulated annealing is applied to determine the program parameter values of a sequence of transformations $T_1, \ldots, T_t$ which must be manually proposed by the user.

3.5. Nelder–Mead or simplex-based techniques

Nelder–Mead or simplex-based methods [28] are non-linear optimization strategies. They navigate a search space evaluating different alternatives within it, and assure convergence towards a local optimum (unless strong conditions are met) after evaluating a undetermined number of alternatives. A main disadvantage of these methods is that they can only be applied to optimize continuous functions which only depend on real variables, therefore, being only applicable when the search space is a subset of $\mathbb{R}^m$, where m is a number of real parameters whose value should be determined by the simplex. This is a hard restriction in the field of auto-tuning, meaning that simplex methods, for example, cannot be used to determine the sequence of transformations $T_1, \ldots, T_t$ that must be applied to the input program P. Instead, the sequence of transformations needs to be selected beforehand by an expert and the method can only be used to tune the parameters exposed by that set of prefixed transformations. Simplex should not be directly applied to optimize parameters whose values belong to a discrete or non-sorted set. For example, simplex cannot be applied to determine the mapping of threads onto CPU cores, since the space of possible solutions consists of permutations of thread identifiers (which is not a continuous space). In spite of these deficiencies, simplex-based methods have been considered by numerous works [15,33] to determine tile sizes values and loop unrolling factors.

3.6. *Evolutionary computation*

Evolutionary computation [3] encompasses different search techniques which have their inspiration in the evolution of species in the nature. Examples of evolutionary computation techniques are Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Strategy (ES), or some other methods which mimic the behaviour of some species in nature, such as Ant Colony Optimization (ACO – [24]). All these techniques share common behaviour. They all work with a set of different alternatives, which in most cases are randomly generated i.e., random values of the parameters governing the computing system, random sequences of transformations and random values of their associated parameters. This set is used to iteratively generate new alternatives by reusing the transformations and parameter values of the best configurations found so far. In summary, an evolutionary computation method navigates the search space following a stochastic path which explores with a higher probability the areas in the search space where the best sequences of transformations and parameter values are located. Evolutionary methods have been applied for auto-tuning of programs in several works. In [16], an evolutionary method auto-tunes a program by determining the compilation flag setting which yields the best execution time. In [22], a genetic algorithm is used to find the optimal tile sizes and unroll factors of a program to which tiling and unrolling transformations are manually applied. Also a genetic algorithm is applied in [2] to compute the optimal algorithm to auto-tune an application for optimal execution time and accuracy.

The popularity of Evolutionary Computation methods is based on their ability to potentially handle an optimization problem and their robustness to deal with complex search spaces such as those associated with auto-tuning of programs.

3.7. *Random search*

Random search implies to evaluate a set of random alternatives and to derive the alternative with the best outcome regarding the optimization criterion. The main disadvantage of this approach is the lack of any guarantee to find any optimal solution (global or local). Random search techniques may however be suitable in those cases where the difference in the value of a specific criterion of the best and worst alternatives is rather small, or when no other technique is available. The results produced by a random search are usually not reproducible in different applications of the method.

3.8. *Machine learning methods*

Machine learning methods are based on the concept of learning some behaviour from data. Examples of machine learning methods are Neural Networks [30], Support Vector Machines (SVN) [8], Gaussian Processes [5], etc. The learnt behaviour can be of different nature, for example, classification, model a function, etc. In the context of auto-tuning, the desired behaviour to be learnt can be the optimal value of the hardware parameters of the computer system, the sequence of transformations to be applied and the optimal parameter values exposed by these transformations. A machine learning method first evaluates several alternatives within the search space for k different input programs $P_1, \ldots, P_k$. The set of evaluated alternatives is referred to as *training data*. The size of this set cannot be predicted beforehand and it depends on the size of the whole search space: the larger the search space, the higher the size. The number of required input applications is also not easy to be determined. Ideally, a set of applications covering all possible program features is desired. Some features of the input programs with the training data results are correlated once the training data has been evaluated. The process of generating and evaluating the training data and learning some behaviour from this data is referred to as *training*. Once the training is finished, and given a new program P_{new} to be optimized, then P_{new} is compared with the programs in the training data to predict the optimal values of the system parameters, the sequence of transformations and their optimal program parameter values. The main advantage of machine learning methods is that the training has to be done only once and tuning a new application is instantaneous which only requires querying the selected machine learning method with the new program features. In general machine learning methods are considered to be good candidates for online tuning. However, in order to efficiently perform online auto-tuning, the training data must include information about the load of a computer system to generate optimal configurations for each different system load.

Machine learning methods mainly reproduce behaviour that has been observed in the training data. Thus, the generation of the training data is crucial when applying machine learning to auto-tune programs. As an example, a machine learning method will not be able to generate the optimal parameters of a transformation T if the training data does not cover these optimal values. More specifically, if T consists of tiling a loop, the training data must comprise the alternative with the optimal or least well-performing tile sizes. As predicting the optimal tile sizes for a tiled loop is not an easy task and must require the application of search techniques (as the ones introduced in this section), machine learning methods are only applied to reduce the search space in the literature. For example, in [1] machine learning is applied to determine the optimal sequence of transformations for a set of benchmark problems. For these transformations no parameters are considered or their value is fixed prior to the application of machine learning. Notice that this approach is only valid when the range of possible values of the parameters is small, and different values of the parameters do not have a significant impact on the code behaviour. In [14], machine learning is used to learn the optimal setting of compilation flags for a program when using GCC. Furthermore, neural networks [30] can be used to learn the behaviour of a given transformation $T(x)$ for different values of the parameter x. Later on, the model is used by another optimization technique in order to search for optimal values of x. Other works use machine learning to determine the hardware configuration for which an application performs best. For example, in [7,25] machine learning is used to determine an optimized kernel distribution strategy for OpenCL over a set of CPUs and GPUs.

4. Multi-objective software auto-tuning

This section is devoted to describe the main differences between single-objective and multi-objective software auto-tuning and to review existing approaches for multi-objective auto-tuning.

As commented before, software auto-tuning is applied many times to simultaneously optimize several non-functional behaviours of a program which may be in conflict. In this case, improving a program P regarding to a given non-functional behaviour a implies to worsen it regarding to another non-functional behaviour b. This has been already observed by some authors in the field of software auto-tuning. For instance, [31] demonstrates an auto-tuning case study to optimize both execution time and power consumption simultaneously. Code versions that are optimal for one objective are not optimal for others.

In such a situation, software auto-tuning is a multi-objective optimization problem to optimize a vector $\vec{f} = \langle f_1, \ldots, f_r \rangle$ of r criteria. The main characteristic of this type of problems is that there does not exist a single alternative which is optimal for all the criteria.

Solving a multi-objective optimization problem consists in computing a set of alternatives each of them representing a trade-off among the optimization criteria [9]. The alternatives in this set fulfill two conditions. Firstly, they cannot be further improved without worsening at least one of the optimization functions. Secondly, none of them is better than the others for all the objective functions. In the field of multi-objective optimization, solutions within this set are referred to as non-dominated solutions, and the set itself is defined as *Pareto set*. The same set, when only the value of the optimization functions are considered, is referred to as *Pareto front*. This set is highly valuable to decide about which solution to select. The Pareto front shows all the relations among the optimization functions and it reveals trade-offs which are very difficult to detect without it.

Although the field of multi-objective optimization has been very active in the last decade and several methods have been proposed for computing the Pareto set/front of a problem, much related auto-tuning work still transforms the multi-objective auto-tuning problem into a single-objective one (which are solved by any of the solution methods described in the previous section) which can be classified into two approaches. The first approach optimizes only one of the objectives as much as possible and consider the remaining objectives as constraints that cannot be violated. The problem with this approach is how to set realistic values for the constraints. On the one hand constraint values must be realistically achievable. On the other hand, it may be possible by marginally relaxing one of the constraints may enable an optimization method to dramatically improve the selected objective to be optimized.

The second approach, which is most widely used for auto-tuning, reduces the problem to a single objective by means of aggregating objective functions. This aggregation reflects the user preferences defined over the optimization criteria. For example, given a problem that minimizes the r objective functions $\langle f_1, \ldots, f_r \rangle$ and a vector of preferences $\langle \lambda_1, \ldots, \lambda_r \rangle$ over the objective functions, then we can transform this problem to optimize the following function with $\sum_{i=1}^{r} \lambda_i = 1$:

$$\sum_{i=1}^{r} \lambda_i f_i. \tag{1}$$

In the following, we evaluate the advantages of computing the Pareto front versus computing a single solution with an aggregation method.

4.1. Aggregation of the objective functions

Aggregation approaches provide a single solution without the need to select a specific solution from the Pareto front. However, this approach should be carefully used since there is no guarantee that the computed solution fulfils user preferences. Figure 3 and Table 1 demonstrate an example and problems associated with aggregation. For an application for which execution time (f_1) and energy consumption (f_2) must be minimized, we have plotted the same Pareto front twice. The leftmost plot shows the real values of the objective functions for each alternative within the Pareto set; the rightmost plot visualizes the same Pareto front with normalized values (between 0 and 1) for the objective functions. These two Pareto fronts apparently have the same geometric shape and in fact they are the same.

The numerical values of the alternatives composing the Pareto fronts are tabulated in Table 1. This table is divided in two parts: the left side part contains the information of the Pareto front without normalization; and, the right side part contains the information of the Pareto front after normalization. In both cases, the table contains for each solution in the Pareto front the weight vectors for which that solution is the optimal one in an aggregation approach like the one depicted by Eq. (1). This way, the first row on the left part indicates that the solution $f_1 = 3.7 \times 10^7$, $f_2 = 6.12$ is the optimal one (smallest value) of this equation for any $\lambda_1 \in [0.00001, 1]$ and $\lambda_2 \in [0.0, 0.99999]$. This table exposes that for an aggregation approach with preference vectors for which a solution within the Pareto front is optimal depends on the range of values of the optimization functions. For example, for the Pareto front without normalization, the solution (tabulated in the first row) minimizing f_1 optimizes Eq. (1) for practically all weight vectors except for $\lambda_1 = 0$, and $\lambda_2 = 1$. This means that an aggregation approach will almost always provide the same solution even though the user may have selected a different solution if the entire Pareto front would have been available. This problem is due to the different value ranges for the objective functions f_1 and f_2. On the other hand, for the Pareto front with normalization, there exists a larger variety of optimal solutions depending on the outcome of the weight vectors. It should be noted that the example shown in Fig. 3 and Table 1 corresponds to a real application code for which execution time (measured in milliseconds) and energy consumption (measured in Watt/hour) have been optimized.

In order to avoid such a misinterpretation of results, the objectives should be normalized prior to the application of an aggregation approach. Unfortunately, the

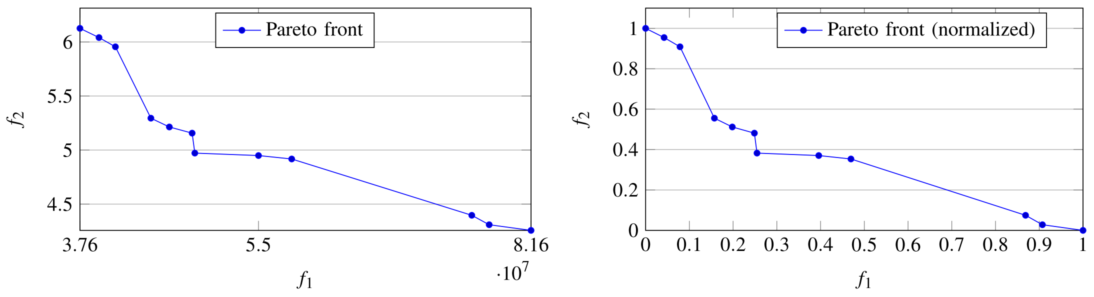


Fig. 3. The same Pareto front plotted twice. The left diagram illustrates the actual objective values. The right diagram shows the objectives values after being normalized to the [0, 1] interval. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140394.)

Table 1

Preferences (λ_1, λ_2) and resulting objective values (f_1,f_2) that minimize Eq. (1)

Pareto front without normalization				Pareto front with normalization			
Weight vector		Solution		Weight vector		Solution	
λ_1	λ_2	f_1	f_2	λ_1	λ_2	f_1	f_2
[0.00001, 1.0]	[0.0, 0.99999]	3.7×10^7	6.12	[0.74, 1.0]	[0, 0.26]	0	1
[0.0, 0.00001]	[0.99999, 1.0]	3.9×10^7	6.04	[0.73, 0.74]	[0.26, 0.27]	0.04	0.95
		4.1×10^7	5.95			0.07	0.90
		4.4×10^7	5.29	[0.63, 0.74]	[0.26, 0.37]	0.15	0.55
		4.6×10^7	5.21	[0.62, 0.63]	[0.37, 0.38]	0.19	0.51
		4.8×10^7	5.15			0.24	0.48
		4.9×10^7	4.97	[0.35, 0.62]	[0.38, 0.65]	0.25	0.38
		5.5×10^7	4.94	[0.34, 0.35]	[0.65, 0.66]	0.39	0.37
		5.8×10^7	4.91			0.46	0.35
		7.5×10^7	4.39			0.86	0.07
		7.7×10^7	4.30	[0.24, 0.34]	[0.66, 0.76]	0.90	0.02
		8.1×10^7	4.25	[0.0, 0.24]	[0.74, 1.0]	1	0

minimum and maximum values for every optimization function are required ahead of the normalization process. These values are usually not known before solving the optimization problem. Although different aggregation methods have been applied in the literature, all of these methods suffer by this drawback when objectives values are not normalized and the minimum and maximum value for each objective is unknown.

Despite these drawbacks, aggregation methods have been used in several works. In [26], dynamic programming, is used to reduce the energy consumption while maintaining or improving the performance of message-passing and shared memory applications. The search strategy relies on models for predicting execution time and energy consumption, thus avoiding the evaluation of alternatives to derive their execution time and energy consumption. [32] introduces how to optimize several combinations of power and execution time based on preferences by means of a hierarchical search procedure. In [34] real power measurements are obtained from a dedicated power device to guide the search for optimal solutions.

4.2. *Computing the whole Pareto front of solutions*

Computing the whole Pareto front have several advantages over the aforementioned method. Firstly, it can be shown to the user who can visually explore it and select the solution which best fits his/her interests. Secondly, solutions within the Pareto front can be automatically selected afterwards, using an aggregation approach (or any other decision making procedure as showed in [12]), therefore releasing the user of having to manually pick a solution out of the Pareto front. In this latter case, the aggregation approach does not suffer by the previously mentioned drawbacks: the Pareto front can be normalized because it contains already the optimal solutions for every optimization function.

Computing the Pareto front does not necessarily require additional computational effort compared to a single-objective approach. Furthermore, in some cases, computing the Pareto front may result in better results for each single objective than optimizing these objectives separately. An example of this was shown in [23], where using a multi-objective approach avoid getting trapped in local optima, and therefore, computing better results.

Some of the techniques applied for single-objective optimization do not have an equivalent multi-objective counterpart. For example, despite some attempts to apply machine learning to optimize multiple criteria [14], machine learning has never been used to solve a truly multi-objective optimization problem by computing the entire set of Pareto efficient solutions. The same applies to simplex-based techniques, local search methods, or simulated annealing.

Other popular optimization techniques use exhaustive search, random search, or evolutionary algorithms. Exhaustive and random search techniques can be extended for multi-objective optimization by simply including an external archive which contains all the non-dominated solutions found so far. In order to properly capture those solutions, the insertion mechanism of solutions in this archive should work as follows. Before inserting a solution s, it is compared with the content of the archive. If s is dominated by a solution in the archive, then s is discarded, otherwise, s should be inserted in the archive and all solutions dominated by s should be removed. Exhaustive search, random search and evolutionary algorithms also suffer from the same drawbacks explained for the multi-objective case: prohibitively high computational effort, and no guarantee of finding solutions close to the optimal. Random search has been explored for auto-tuning of programs in [19]. Exhaustive search techniques remain also unpractical for multi-objective auto-tuning.

Multi-objective evolutionary algorithms are popular methods for solving multi-objective optimization problems. They guarantee an approximation of the Pareto front with a reasonable computational effort. Only little work is known in the literature that explored these algorithms for auto-tuning of programs. For example, the trade-off between execution time and resource usage has been investigated in [19]. Reference [17] explored execution time and efficiency or execution time or efficiency and compilation time. The trade-off between time and energy consumption/power has been analyzed in [13] and [4]. However, multi-objective evolutionary algorithms suffer by the same drawbacks as mentioned for single-objective auto-tuning approaches. In particular, they are difficult to employ for online auto-tuning due to the number of different alternatives that should be tested.

5. Empirical comparison

This section presents our results for an empirical evaluation of single versus multi-objective auto-tuning considering some of the techniques described in previous sections. For this evaluation we use a parallel implementation of an *n-body* simulation which contains two-nested loops to which loop tiling has been applied. For each loop the tile size is exposed as a tunable parameter. The different auto-tuning techniques will explore these tile size parameters, the number of threads for running this code, and the processor clock frequency (by using DVFS – dynamic voltage and frequency setting). Our target machine is a quad-socket shared-memory system equipped with Intel Xeon E5-4650 processors, each offering 8 cores clocked at 1.2–2.6 GHz. Each core features private L1 and L2 caches of 64 and 256 KB each in addition to the CPU-wide shared L3 cache of 20 MB. The system provides 128 GB of main memory, uses a Linux operating system with a 3.5.0 kernel and our backend compiler is GCC 4.6.3. Hyper-Threading was not used for any of our experiments.

The evaluated techniques for single-objective auto-tuning comprise local search (LS), simulated annealing (SA), and a genetic algorithm (GA) as a representative for evolutionary computation. The goal of these approaches is to tune the code for minimizing execution time. We have not considered any other technique described in Section 2. We had no mathematical models for describing the behaviour of different configurations of the code. Exhaustive search cannot be applied due to the size of the search space. Simplex cannot be applied since the search space is discrete. Finally, for using machine learning, we would have needed more codes with similar features compared to the ones exposed by the *n_body* applications in order to generate enough data for the training phase.

In order to examine multi-objective auto-tuners we have considered a genetic algorithm known as NSGA-II [10], and RS-GDE3 a differential evolution method endowed with a mechanism to effectively prune the search space. The former is a very popular algorithm in the field of multi-objective optimization, and the latter has been applied in [19] to optimize execution time and resource-usage of five different parallel programs. In our experiment, the goal of these two methods is to optimize the code minimizing its execution time and the energy consumption entailed by its execution.

For the sake of a fair comparison, all the techniques evaluate the same number of code alternatives, which we set to a total of 500, therefore implying a similar computational effort. In order to achieve statistical significance for our observations, we have repeated every experiment 30 times.

Figure 4 shows the fastest configuration evaluated by each technique during the different iterations. The LS and SA have taken more time than the evolutionary techniques to converge towards a solution. Among all the evaluated techniques, RS-GDE3 has been the technique computing the fastest solution at much earlier iterations than all other techniques. The numerical results averaged after the 30 runs are summarized in Table 2.

Figure 5 displays the Pareto fronts derived with RS-GDE3 and NSGA-II versus the solutions computed by the single-objective auto-tuners. We can observe that the Pareto front computed by RS-GDE3 contains the best solution for each objective. When comparing RS-GDE3 with NSGA-II, we can find out that RS-GDE3

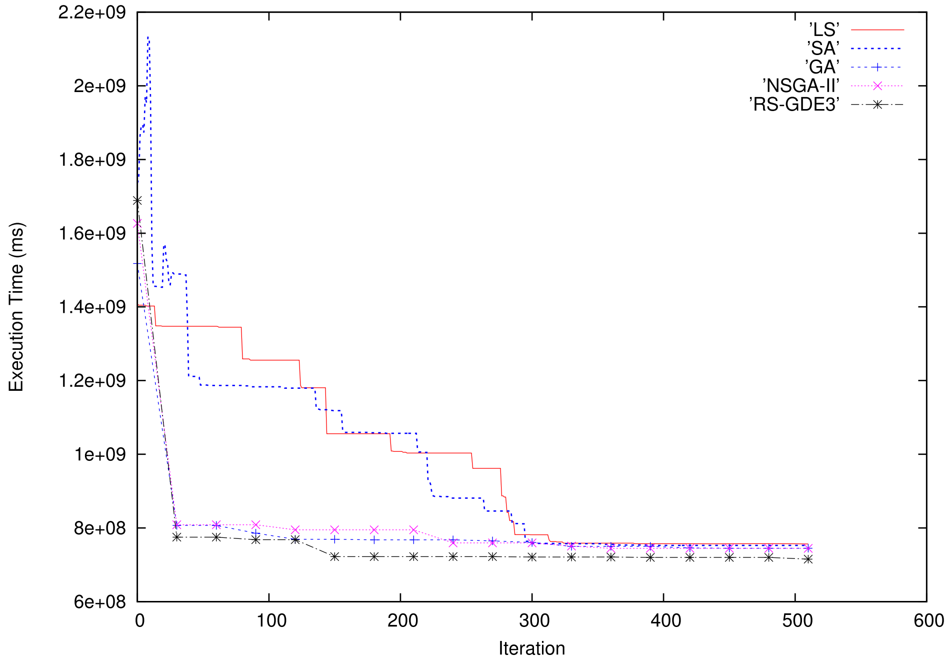


Fig. 4. Improvement in execution time behavior of the configuration computed by different iterations of various auto-tuning techniques for an n-body program. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140394.)

Table 2

Runtime of the fastest configuration of the *n_body* program found by the different auto-tuning algorithms

Method	Time (milliseconds)
Local search	7.56e + 08
Simulated annealing	7.52e + 08
Genetic algorithm	7.45e + 08
NSGA-II	7.45e + 08
RS-GDE3	**7.15e + 08**

in many cases obtains solutions with shorter execution time and lower energy consumption than the ones computed by NSGA-II.

Regarding the experiments conducted with single-objective auto-tuners, we found out that the local search and simulated annealing approach derived solutions with larger execution time than the ones computed by the multi-objective approaches. The genetic algorithm found a solution that is faster than the one determined by NSGA-II; however, the advantage of NSGA-II is that its solution consumes less energy. The comparison included in this section demonstrates the potential of multi-objective auto-tuners compared to single-objective approaches: With the same computational effort, the multi-objective auto-tuners not only provide a set of trade-off solutions between execution time and energy consumption, but they also derive better configurations with lower execution time compared to single-objective methods.

6. Conclusions and future work

In this paper we described and compared different approaches to auto-tuning of programs for single- and multi-objective optimization. Although many approaches exist, to the best of our knowledge none of them deal with the general auto-tuning problem stated in Section 2. Instead, different methods are applied to solve different relaxed versions of the problem. For example, search approaches like hill climbing, simplex or evolutionary computation methods have been extensively explored to determine the optimal parameter values of a computing system, or the parameter values of a set of fixed transformations; some of these approaches guarantee to find close to optimal solutions but they can be computationally quite expensive. Machine learning methods have been successfully applied to determine optimized sequences of program transformations or compiler flag settings. The computational effort is included in the generation and evaluation of the training data. However, once this phase is completed, the application of the method is fast. The results derived by machine learning based auto-tuning depend

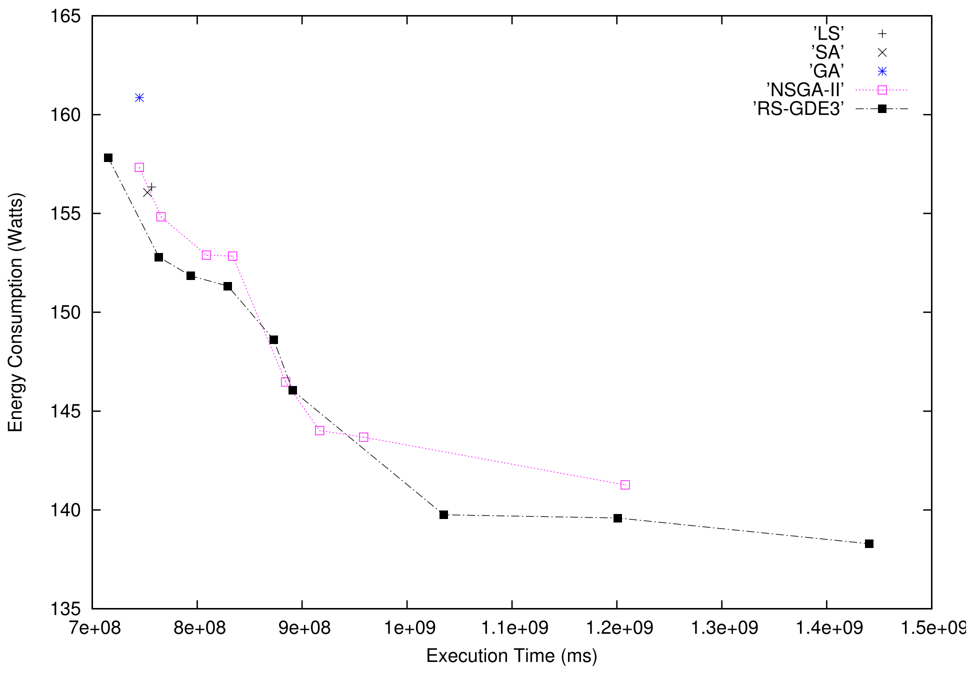


Fig. 5. Single vs multi-objective auto-tuning results for the n-body program. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140394.)

on the similarity of a program to be optimized compared to those codes used during the training phase. Furthermore, if we want to obtain results close to the optimal solutions, then these solutions should be included in the training data. In order to achieve this, search methods must be applied ahead of the machine learning phase.

For multi-objective optimization many works in the literature reduce the problem to a single-objective problem by using preferences. We argue in this paper that these preferences should be carefully applied, in particular if the objective values are not normalized. Multi-objective approaches that determine the entire Pareto front have not extensively researched in the past. We empirically demonstrated in this paper the potential of multi-objective compared to single-objective auto-tuning both in terms of optimization results and effort to derive these results.

Important future research directions in the area of auto-tuning of programs for multiple objectives comprise, firstly, the design of methods combining the advantages of search based approaches and machine learning suitable for solving the general auto-tuning problem. Secondly, more aggressive techniques to reduce the huge auto-tuning search space must be developed. Thirdly, more sophisticated techniques for deriving the entire Pareto front for multi-objective problems and the design of advanced methods for selecting a solution out of the Pareto front, are required.

Acknowledgements

This project was partially funded by the Austrian Science Foundation (FWF) under contract I1523 (EASE) and the Austrian Research Promotion Agency under contract 834307 (AutoCore).

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint and C. Williams, Using machine learning to focus iterative optimization, in: *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2006, pp. 295–305.

[2] J. Ansel, Y.L. Wong, C.P. Chan, M. Olszewski, A. Edelman, S.P. Amarasinghe and S.P. Amarasinghe, Language and compiler support for auto-tuning variable-accuracy algorithms, in: *CGO*, 2011, pp. 85–96.

[3] T. Back, D.B. Fogel and Z. Michalewicz, *Handbook of Evolutionary Computation*, IOP Publishing, 1997.

[4] P. Balaprakash, A. Tiwari and S.M. Wild, *Multi-Objective Optimization of HPC Kernels for Performance, Power, and Energy*, 2013.

[5] C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford Univ. Press, 1995.

[6] C. Blum and A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, *ACM Computing Surveys (CSUR)* **35**(3) (2003), 268–308.

[7] X. Chen and S. Long, Adaptive multi-versioning for OpenMP parallelization via machine learning, in: *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, ICPADS'09*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 907–912, available at: http://dx.doi.org/10.1109/ICPADS.2009.77.

[8] C. Cortes and V. Vapnik, Support-vector networks, *Machine Learning* **20**(3) (1995), 273–297.

[9] K. Deb and D. Kalyanmoy, *Multi-Objective Optimization Using Evolutionary Algorithms*, Wiley, New York, NY, USA, 2001.

[10] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Transactions on Evolutionary Computation* **6**(2) (2002), 182–197.

[11] M. Duranton, D. Black-Schaffer, K. De Bosschere and J. Maebe, *The HIPEAC Vision for Advanced Computing in Horizon 2020*, 2013, pp. 1–48.

[12] M. Ehrgott, *Multicriteria Optimization*, Springer, New York, NY, USA, 2005.

[13] V.W. Freeh and D.K. Lowenthal, Using multiple energy gears in MPI programs on a power-scalable cluster, in: *Proceedings of the Tenth ACM SIGPLAN PPoPP*, ACM, 2005, pp. 164–173.

[14] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Asthon, E. Bonilla, J. Thomson, C. Williams and M. O'Boyle, Milepost GCC: machine learning enabled self-tuning compiler, *International Journal of Parallel Programming* **39**(3) (2011), 296–327.

[15] A. Hartono, B. Norris, P. Sadayappan and P. Sadayappan, Annotation-based empirical performance tuning using Orio, in: *IPDPS*, 2009, pp. 1–11.

[16] K. Hoste and L. Eeckhout, Cole: Compiler optimization level exploration, in: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, 2008, pp. 165–174.

[17] K. Hoste and L. Eeckhout, Cole: compiler optimization level exploration, in: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, 2008, pp. 165–174.

[18] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 3B, Part 2, Intel, June 2013.

[19] H. Jordan, P. Thoman, J.J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer and M. Moritsch, A multi-objective auto-tuning framework for parallel codes, in: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–12.

[20] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, Optimization by simulated annealing, *Science* **220**(4598) (1983), 671–680.

[21] T. Kisuki, P.M.W. Knijnenburg and M.F.P. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, in: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, Philadelphia, October 15–19, 2000, IEEE Computer Society Press, pp. 237–246.

[22] P.M.W. Knijnenburg, T. Kisuki and M.F.P. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, *The Journal of Supercomputing* **24**(1) (2003), 43–67.

[23] J. Knowles, R. Watson and D. Corne, Reducing local optima in single-objective problems by multi-objectivization, in: *Evolutionary Multi-Criterion Optimization*, E. Zitzler, L. Thiele, K. Deb, C. Coello Coello and D. Corne, eds, Lecture Notes in Computer Science, Vol. 1993, Springer, Berlin/Heidelberg, 2001, pp. 269–283.

[24] G.A. Kochenberger et al., *Handbook in Metaheuristics*, Springer, 2003.

[25] K. Kofler, I. Grasso, B. Cosenza and T. Fahringer, An automatic input-sensitive approach for heterogeneous task partitioning, in: *Proceedings of the 27th ACM International Conference on Supercomputing, ICS'13*, ACM, New York, NY, USA, 2013, pp. 149–160.

[26] D. Li, B. de Supinski, M. Schulz, D. Nikolopoulos and K. Cameron, Strategies for energy efficient resource management of hybrid programming models, *IEEE Trans. Parallel Distrib. Syst.* **24**(1) (2013), 144–157.

[27] K. Naono, K. Teranishi, J. Cavazos and R. Suda, *Software Automatic Tuning (From Concepts to State-of-the-Art Results)*, Secaucus, NJ, Springer-Verlag, New York, USA, 2010.

[28] J.A. Nelder and R. Mead, A simplex method for function minimization, *The Computer Journal* **7**(4) (1965), 308–313.

[29] Z. Pan and R. Eigenmann, Fast and effective orchestration of compiler optimizations for automatic performance tuning, in: *International Symposium on Code Generation and Optimization, 2006. CGO 2006*, IEEE, 2006, pp. 319–332.

[30] M. Rahman, L.-N. Pouchet and P. Sadayappan, Neural network assisted tile size selection, in: *International Workshop on Automatic Performance Tuning (IWAPT'2010)*, Berkeley, CA, Springer-Verlag, 2010.

[31] S. Rahman, J. Guo, A. Bhat, C. Garcia, M. Sujon, Q. Yi, C. Liao and D. Quilan, Studying the impact of application-level optimizations on the power consumption of multi-core architectures, in: *Proceedings of the 9th Conference on Computing Frontiers*, ACM, 2012, pp. 123–132.

[32] S.F. Rahman, J. Guo and Q. Yi, Automated empirical tuning of scientific codes for performance and power consumption, in: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ACM, 2011, pp. 107–116.

[33] A. Tiwari, C. Chen, J. Chame, M. Hall and J.K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, in: *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, IPDPS'09*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12.

[34] A. Tiwari, M. Laurenzano, L. Carrington and A. Snavely, Auto-tuning for energy usage in scientific applications, in: *Euro-Par 2011: Parallel Processing Workshops*, Springer, 2012, pp. 178–187.