Multi-Objective Region-Aware Optimization of Parallel Programs

Juan J. Durillo[a], Philipp Gschwandtner[b], Klaus Kofler[b], Thomas Fahringer[b]

[a]*Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Boltzmannstrae 1, 85748 Garching, Germany*
[b]*University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria*

Abstract

Auto-tuning has become increasingly popular for optimizing non-functional parameters of parallel programs. The typically large search space requires sophisticated techniques to find well-performing parameter values in a reasonable amount of time. Different parts of a program often perform best with different parameter values. We therefore subdivide programs into several regions, and try to optimize the parameter values for each of those regions separately as opposed to setting the parameter values globally for the entire program. In order to manage this enlarged search space, we have to extend existing auto-tuning techniques to ensure high quality solutions to this optimization problem. In this paper we introduce a novel enhancement to the RS-GDE3 algorithm which is used to explore the search space for auto-tuning programs with multiple regions regarding several objectives. We have implemented our auto-tuner using the Insieme compiler and runtime system and provide a detailed analysis of the obtained results with the aim of gaining a better understanding of non-functional inter-region behavior in the context of auto-tuning. In comparison to a non-optimized parallel version of the tested programs, our novel approach achieves up to 7.6X, 10.5X, and 61.6X improvements for three tuned objectives wall time, energy consumption, and resource usage, respectively.

Keywords: auto-tuning, code region, multi-objective, OpenMP, parallel program, energy

1. Introduction

Optimizing parallel programs becomes increasingly difficult with the rising complexity of modern parallel architectures which includes, for example, a dramatic increase of the number of cores per chip or the availability of multi-level partially shared cache hierarchies. Automatic software tuning, or simply *auto-tuning* [1], arose as an attempt to better exploit hardware features by automatically tuning applications. An auto-tuner tries to find promising *configurations* for a given program executing on a given target platform. A configuration consists of a set of non-functional parameters with a corresponding value range which can influence a program's performance by transforming the source code of the application, or by finding the right parameter values that govern the execution of an application on a given architecture (e.g. number of threads, frequency per core, etc.).

This paper describes a novel auto-tuning approach for programs with multiple single-entry single-exit code regions whose non-functional behavior depends on at least one tunable parameter. We assume that we can measure the non-functional behavior of these regions for the optimization objectives (e.g. wall time, energy consumption, etc.). Tuning multi-region applications exposes additional challenges for auto-tuning techniques. Firstly, there usually does not exist a single set of parameter values which is optimal for all the regions of a program. However, setting the parameter values individually for every region leads to a huge search space as it grows exponentially with the number of tuning opportunities, i.e. the number of regions. Secondly, the execution of a region may be influenced by the parameter values applied to neighboring regions. Previous work [2] observed that the optimal parameter values for individual regions of hybrid MPI/OpenMP applications led to sub-optimal overall performance.

Auto-tuning techniques are widely used [3, 4, 5, 6, 7, 8, 9, 10, 11] but are often limited to using the same parameter values for every region, i.e. globally for the entire program, ignoring the fact that different parts of the code may benefit from specific parameter values.

Email addresses: juan.durillobarrionuevo@lrz.de (Juan J. Durillo), philipp@dps.uibk.ac.at (Philipp Gschwandtner), klaus@dps.uibk.ac.at (Klaus Kofler), tf@dps.uibk.ac.at (Thomas Fahringer)

Our auto-tuner can optimize a generic set of objectives which do not necessarily correlate with each other. This lack of correlation makes it impossible to find a single solution which is best in every objective. For example, if some regions are optimized for one objective while others are optimized for different objectives, it is very likely that the overall performance of the program will suffer. It can happen that a configuration exhibits small execution time at the cost of high resource usage for one region, but the contrary for another region, resulting in sub-optimal overall performance.

The approach proposed in this paper extends the method presented in [12], which is limited to optimizing each region in isolation, by adding region-aware auto-tuning support for the entire program. The contributions of this paper are the following:

- A region-aware multi-objective auto-tuner.
- A compiler-runtime system that automatically identifies regions and enables automatic tuning of their parameters.
- Evaluation of several global and region-aware auto-tuning strategies for several codes on different target architectures which demonstrates the importance of region-aware auto-tuning compared against global optimization.
- A thorough analysis of how the regions composing a program behave within different (quasi-)Pareto optimal program configurations.
- A study of the computed trade-off in terms of the three considered criteria.

The rest of this paper is organized as follows: Section 2 exemplary shows the inherit advantages of region-aware auto-tuning while Section 3 gives detailed insights on the auto-tuning approach presented in this paper. The implementation details of the presented work can be found in Section 4 while Section 5 details the metrics that we use to compare the quality of the different auto-tuning approaches. results of the experiments performed in this paper are shown in Section 6 includes the results obtained in our experiments as well as a comparison among different multi-objective region-aware tuners. A further analysis of the solutions computed by the best approach in the comparison before is included in Section 7. Section 8 gives an overview of the related work and the conclusion of our findings are summarized in Section 9.

2. Motivation

In this section, we motivate the need for region-aware auto-tuners by using a simple example program consisting of two regions. Both regions perform a parallel matrix multiplication. In the first region, only the outermost loop is parallelized; in the second, we parallelize only the innermost loop. As a consequence we have two regions with different execution behavior: the first one scales well with the number of threads whereas the second one does not. In order to have similar execution times for both regions, the matrix size in the second region is only one quarter of the matrix size in the first region.

The experiments in this section are performed on the Ivy Bridge-EP architecture described in Section 6. Our goal is to find the optimal configuration for executing this program. For the sake of performing an exhaustive search of all possible program configurations, we assume that these regions only expose the number of threads as a tunable parameter.

We select configurations for the previously described program using three different approaches:

- Isolated: This approach optimizes both regions in isolation.
- Global: This approach is constrained to find a single set of parameter values to be used in both regions.
- Region-Aware: This approach optimizes both regions using individual parameter values for each of them to maximize the program's overall performance.

Table 1: Theoretical potential of different auto-tuning approaches, determined by exhaustive search for an example program with two regions.

	Isolated	Global	Region-Aware
#Threads Region 1	20	10	10
#Threads Region 2	2	10	7
Region 1[1]	546	1075	1075
Region 2[1]	5798	2652	2366
Total[1]	6344	3727	3442
Relative time difference	1.84	1.08	1.00

[1] Execution time in milliseconds.

The best configurations found by these approaches are summarized in Table 1. The first two rows of the table show the number of threads for each region chosen by the corresponding approach — which can be found in the different columns. The following two rows include the execution time for each of these regions with the aforementioned indicated number of threads. The fifth row shows the program's execution time when the regions are configured as indicated in the first two rows. Finally, the last row shows the relative difference regarding the best found execution time across all three approaches.

The worst execution time corresponds to the Isolated approach. The reason is that it will run the first region with a large number of threads, as it does scale well. However, this has a negative effect on the second region, which does not scale well. The change from 20 to only a few threads between the regions introduces a significant overhead by the underlying runtime system, as it also implies a change of the number of sockets where computations are performed on. The fastest option for the second region is to use two threads, taking 5798ms. Using for example 7 threads, which is the fastest option after the first region has been executed with 10 threads, results in an execution time of 6387ms in this scenario.

The Global approach yields better results since the code regions are not optimized in isolation but regarding the overall program performance. However, as this approach is limited to the same parameter values for both regions, it is unable to exploit the full potential of the hardware. This drawback is overcome by the Region-Aware approach. Besides taking into account the overhead for a change in using different socket numbers, it can customize the regions to their needs for hardware resources.

3. Multi-objective Tuning of Multi-Region Programs

In this section we firstly introduce some background related to multi-objective auto-tuning of programs. Afterwards, we state the main challenges when tuning multi-region programs.

3.1. Background on Multi-Objective Auto-Tuning

Auto-tuners may optimize several objectives which sometimes conflict with each other. This means that optimizing one of them is only possible by worsening the value of at least one of the other objectives. The mathematical solution to such problems is not defined by a single point, but by a set of points representing a trade-off between these objectives. The set of solutions representing the optimal trade-off between the considered objectives is known as *Pareto set*.

Our approach to apply multi-objective software auto-tuning consists in computing the Pareto set or an approximation of it [13]. Often, related work reduces multi-objective optimization problems to a single objective one by using fixed weights for the individual objectives. Therefore, they try to find a single solution which is near optimal in a pre-defined set of preferences for the objectives. Computing the Pareto set instead of a single solution is often incorrectly cited by related work with the belief that it implies a manual selection of a solution by the user which is just one possibility. However, different alternatives comprise a selection based on preferences specified a posteriori — i.e. once the Pareto set is computed. This latter approach does not require manual interaction of the user, and has the advantage that the computed solution belongs to the optimal trade-off. While computing the Pareto set may seem to require a higher computational effort than computing a single solution, literature in multi-objective optimization shows that this is not necessarily the case [14]. Indeed, the opposite is often true: computing the whole Pareto set may

be easier than computing some individual solutions within it, as finding solutions which show a good compromise between two or more objectives implies a different way of navigating the search space.

3.2. Challenges in Tuning Multi-Region Programs

We focus on programs composed of multiple regions. Tuning such programs requires to tune each of these regions. A region's performance may depend on the way other regions are executed, what data they access, and other side effects. Neighboring regions on a control flow are not independent of each other. This issue implies that regions should not be tuned in isolation, which has been observed in [2].

Tuning multi-region applications introduces additional challenges. Firstly, the search space of possible configurations of a program grows exponentially with the number of regions. For example, the *matrix multiplication* kernel considered in [12] consists of a single region. That region requires to tune three tiling dimensions and the number of threads. For a problem size of $14,000$ —i.e. matrices of $14,000 \times 14,000$ elements— and a machine with 32 cores, the search space of possible program executions is $700^3 \times 32 \simeq 10^{10}$. If a program consists of two regions similar to that one, the search space would be $(700^{\times}32)^2 \simeq 10^{20}$. Larger search spaces often reduce effectiveness of search methods [15]. Secondly, in a multi-objective multi-region scenario, it is crucial that the parameter values for the different regions within a program aim the optimization of the same objective. Otherwise, if two regions are assigned parameter values optimizing different objectives, most likely the execution of both regions together will not be optimal for any of these objectives. For example, this is the case when half of the regions within a program would be executed with optimal parameter values for a given objective and the other half of the regions with parameter values optimal for a conflicting objective.

Our goal in this paper is to design an auto-tuner that can find a single Pareto set of configurations for a given program with multiple regions. While the parameter values of every region are tuned separately, we measure the effect of changing the parameter values of a region regarding the entire program instead of considering the effect only for individual region executions. In this way, we optimize the whole program execution instead of focusing on specific regions. After the Pareto set for the whole program is computed, a single configuration for the entire program can be selected from the Pareto set, either manually or automatically. This approach differs from the one proposed in [12], which is based on computing an individual Pareto set for every single region in isolation, making this approach prone to the performance penalties described in Section 8. Furthermore, computing a Pareto set independently for every region requires a decision making process for every single region. Therefore, the approach presented in [12] is unfeasible for tuning programs with a large number of regions. A feasible way to compare the approach in Jordan et al. [12] to the one presented in this paper is using the same set of parameters for every region of the program, thereby reducing the search space and producing only a single Pareto set for the entire program. Additionally, the auto-tuner can evaluate the performance of a configuration for all regions at once which makes it aware of eventual performance penalties caused by region interferences. In Section 6, we compare this version of the auto-tuner presented in [12], which we call RS-GDE3 Global, against the new version of this paper.

3.3. Method

Our approach extends the RS-GDE3 algorithm presented in [12], which is based on iterative compilation. It uses a fixed size set of different program configurations to be executed on the target architecture in order to determine their performance. RS-GDE3 refers to this set as *population*. Iterative compilation methods update this set across different iterations by generating possibly better performing configurations for the program being tuned. In the case of RS-GDE3, this is done by generating a new population called *offspring population* from the current population as explained later in this section. At the end of every iteration, the current population is updated by considering its content and the content of the offspring population. Details about how configurations are chosen to be part of the population for the next iteration can be found [12].

In order to tune multi-region programs regarding multiple objectives, we need to overcome the following problems:

1. All the parameter values within a configuration should aim for a common goal. If the tuner generates a program where a region *a* uses parameter values optimized for a given objective, and for a subsequent region *b* it uses the best parameter values regarding another objective, then the execution of both regions will unlikely be optimal for any of these objectives nor will it represent an optimal trade-off.

1: **for all** configurations *parent* in population P **do**
2: Randomly pick configurations c_1, c_2 and c_3 from P
3: $diff \leftarrow c_3 + (c_1 - c_2)/2$
4: Create empty configuration c_{new}
5: **for all** parameter x **do**
6: Randomly choose x value from either $diff$ or *parent*
7: Add x value to the newly generated configuration c_{new}.
8: **end for**
9: Add c_{new} to newly generated Population P_{new}fas
10: **end forreturn** P_{new}

Figure 1: General Differential Evolution.

2. An intractable large search space, which may reduce the effectiveness of the search performed by RS-GDE3.
3. Existing or changing parameter settings of one region that may negatively impact other regions, introducing additional overheads.
4. Well performing sets of parameter values for individual regions may be discarded by the tuner if they are considered in combination with poorly performing parameter values for other regions.

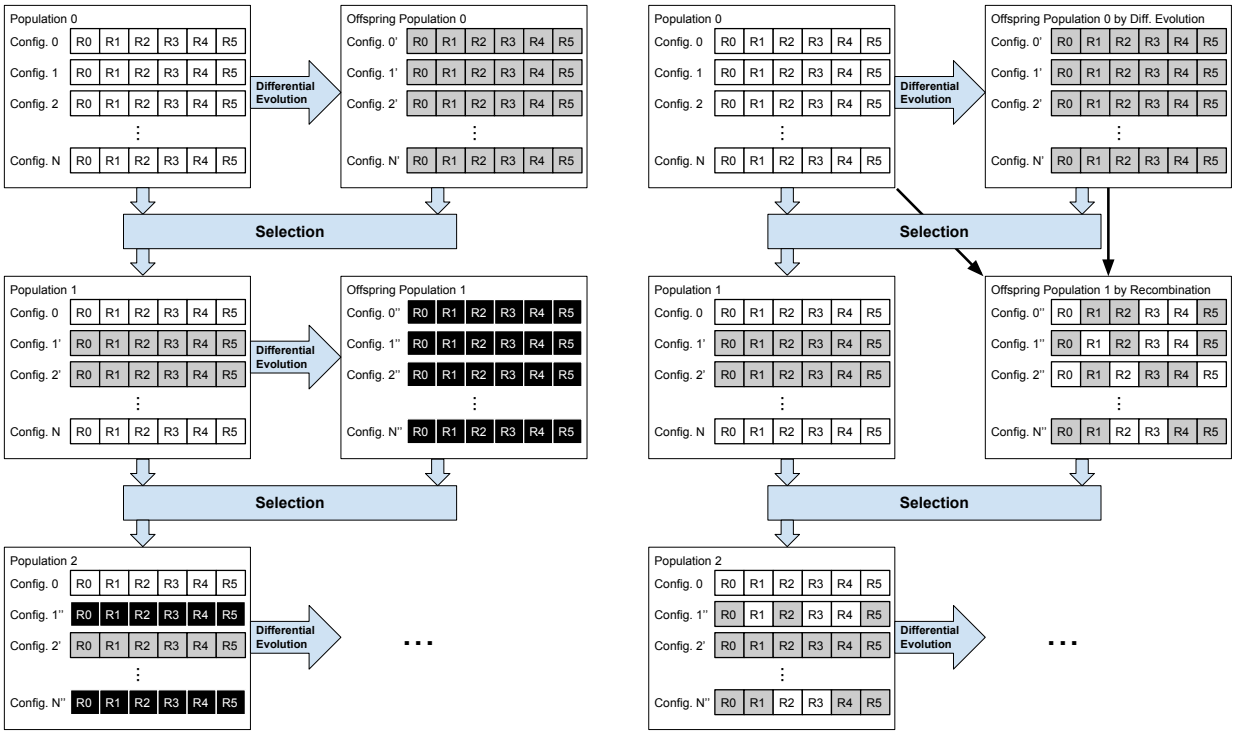
To solve the first problem, our approach does not consider regions in isolation. Instead, our configurations are comprised of the parameter values of all regions and will be kept as part of the population only if they contribute to optimize the whole program.

To overcome the second problem, finding a good starting point in this huge search space is crucial to improve the effectiveness of the tuner. For this reason, we perform a *global pre-tuning* phase. During the pre-tuning phase, the auto-tuner uses the same set of parameter values for every region within the program. The idea is to reduce the size of the search space, making it easier for tuner to find the best configurations within the limited search space. We use these configurations as the starting point of a second tuning phase where every region can have a different set of parameter values. The global pre-tuning takes place during a few iterations at the beginning of our method. Besides reducing the dimensionality of the search space, the pre-tuning phase also helps to overcome the third problem, since the found configurations avoid overheads caused by changing hardware settings between regions. After the global pre-tuning phase, overheads that arise from changing hardware settings may occur. However the auto-tuner will discard these configurations, unless the benefit of the different parameter values for each region outweighs the overheads caused by using different parameter values for individual regions.

To deal with the problem related to the fourth issue, we developed a novel approach which we call *recombination*. RS-GDE3 is a population based algorithm that generates new configurations by applying an operator called *differential evolution* [16]. Differential evolution is an approximation technique which also accounts for genetic algorithms or simulated annealing. We chose to use differential evolution because it has been empirically shown that it generally needs lower computational effort to produce higher quality results than other approximation algorithms [17]. A differential evolution algorithm generates new configurations by combining some existing ones as shown in Figure 1.

While generating new configurations by combining some existing ones may be beneficial, it also represents a drawback. For example, if the parent configuration c_1 contains the best possible parameter values p_1 for region a and the newly generated configuration c_2 contains the best possible parameter values p_2 for region b, it would make sense to compose a third configuration c_3 that uses p_1 in region a and p_2 in region b. However this is not possible in RS-GDE3 due to the way the differential evolution operator works. The recombination strategy solves this problem by generating new configurations using a different method every second iteration. This new generation method takes the parent configuration and the newly generated one and swaps several of their parameter values. In particular, the best set of parameter values for individual regions out of the parent and the newly generated configuration are preserved. This approach maximizes the chances of combining parameter values which perform well.

A comparison of a traditional, region-aware auto-tuner to the novel approach using recombination is shown in Figure 2. The former, depicted in Figure 2a, generates a new offspring population out of the current population using differential evolution in every iteration and selects the best performing configurations from the current population and the offspring population to form the population for the next step, as it is described in [12]. The latter, depicted



(a) Region-aware auto-tuner using GDE3 only. In each iteration an offspring population of N elements is generated out of the current population.

(b) Region-aware auto-tuner using GDE3 and recombination. In every second iteration, an offspring population is created by recombining the best performing settings for each individual region from the previous population and offspring population.

Figure 2: Examples for the evolution of two region-aware auto-tuners over two iterations, using a population of N configurations, tuning a program with six regions (R0 to R5). Both auto-tuners perform 2×N evaluations in this example. In each iteration, a new population is created by selecting the best configurations from the population of the previous iteration and the corresponding offspring population.

in Figure 2b, uses the differential evolution only in every second iteration. In the other iterations, a new offspring population is generated by recombining the parameter values of the configurations in the current population and the offspring population. The selection of configurations which will form the population of the next step is unaltered compared to the traditional approach. In order to maximize the benefit of the recombination steps, we are using two different types of recombination, which are applied in an alternating fashion:

1. The first type compares each configuration which has been generated during the last iteration of the optimizer with its parent's configuration. Out of those two configurations (offspring and parent), for each region it selects the set of parameter values which results in better values, averaged over all objectives, and combines them to a new configuration for the entire program.
2. The second type selects one configuration for each objective from the offspring population as well as their corresponding parents' configurations. From the parameter values of those configurations, it constructs a new configuration for each objective, combining the sets of parameter values of the regions which deliver the best performance for the corresponding objective.

Finally, the pseudo-code of the whole method is given in Figure 3.

4. Implementation and Experimental Setup

The presented framework is implemented within the Insieme source-to-source compiler and runtime system presented in [12], based on branch *inspire_1.3* which is freely available at [18]. The Insieme compiler performs, among

1: Generate a population P_{pre} of configurations, where regions within the same configuration have the same parameter values
2: Set the iteration counter to zero
3: **while** iteration counter ≤ threshold **do**
4: Generate new offspring population OP_{pre} using the differential evolution method
5: Update P_{pre} with the best trade-off solutions from $P_{pre} \cup OP_{pre}$
6: Increase the iteration counter
7: **end while**
8: Generate a new population P of configurations using solutions in P_{pre}, replicating the global set of parameter values for each region.
9: **while** iteration counter ≤ maximum number of iterations **do**
10: **if** iteration counter is even **then**
11: Generate an offspring population OP using the differential evolution method
12: **else**
13: Generate an offspring OP population using the recombination method
14: **end if**
15: Increase the iteration counter
16: Update P with the best trade-off solutions from $P \cup OP$
17: **end while**
18: **return** the non dominated solutions (i.e. the Pareto set) of P

Figure 3: Multi-region auto-tuner using a combination of general evolution and recombination.

other tasks, code analysis and code transformations. Insieme uses INSPIRE [19] as an intermediate representation for extracting individual regions and applying the required code transformations. The transformed INSPIRE code is then converted back to C. The resulting source code is compiled to binary using the Gnu GCC Compiler. For all measurements presented in this paper, we set the `-O3` flag to obtain the highest performance. In the case of transformations such as tiling, the Insieme compiler generates a different version for each tile size to be evaluated in order to obtain the best performance. This means, that a separate version of the code is generated and compiled for each configuration that needs to be evaluated.

The Insieme runtime system executes the transformed input code and measures its performance. The measurements are reported to the Insieme compiler which in turn provides them to the auto-tuner. The Insieme runtime system allows to set the number of threads individually for each region and is also responsible for mapping the executed program to OS-level threads. For each OS-level thread used, one *worker* is created in the runtime system [20]. The number of workers created for a specific execution is equal to the maximum number of threads used by any region of the given program. When a region is executed with fewer threads than the number of workers started, the additional workers are sent to sleep by the runtime system. While incurring some minor performance overhead upon wakeup, this ensures a minimum of energy consumption for runtime system workers that are not active. Furthermore, each worker is bound to a specific core in ascending order, with a "dense" mapping (one socket is filled first before employing cores of another socket).

Measurements are obtained via x86's *rdtsc* instruction for execution time and Intel's *RAPL* interface for energy consumption. The latter offers a data resolution of 15.3 microjoules and time resolution of 1 millisecond, and related work has shown it to be accurate enough for our purpose [21]. The resource usage metric is calculated by multiplying the number of threads used with the execution time of each individual region.

The experiments are executed on two different machines, the names and characteristics of which are listed in Table 2. The CPU clock frequency is fixed as listed in Table 2, and HyperThreading is disabled on all machines.

4.1. Regions

Our auto-tuner targets parallel programs implemented in C using OpenMP [22] for parallelization. As mentioned in Section 1, the programs are subdivided into several regions. We define each parallel OpenMP for-loop to be a separate region for several reasons: Besides being parallel, these loops usually contain most of a program's computational work. Furthermore, the implicit synchronization following each parallel OpenMP for-loop is well suited as a

Table 2: Machine characteristics.

name	CPUs	total cores	cache sizes	RAM	OS	compiler
Sandy Bridge-EP	2x E5-2690 v2 @ 3.0 GHz	20	priv.: 32 KB, 256 KB, shared: 25 MB	128 GB	CentOS 6.5, 2.6.32-431	GCC 5.1 -O3
Ivy Bridge-EP	4x E5-4650 @ 2.7 GHz	32	priv.: 32 KB, 256 KB, shared: 20 MB	256 GB	CentOS 6.7, 2.6.32-573	

point to vary the number of threads, while the number of threads cannot be changed within the body of an OpenMP for-loop. Additionally, the restrictions enforced by OpenMP on parallel for-loops, such as no continue, break or return statements as well as no modification of the iterator variable inside the loop's body, increase the probability that a loop nest starting with a parallel OpenMP for-loop is suitable for tiling. If an OpenMP loop is called several times from different contexts, each context creates a different region that can be tuned individually.

Loop nests which can be tiled are of special interest to us, as they typically have high optimization potential and consume most resources. Our auto-tuner examines whether a loop nest is suitable for tiling by using the Polyhedral Model [23] which is integrated in the Insieme Compiler. This analysis determines whether a loop nest is tilable and also provides information about the minimum and maximum tile size for this transformation.

For every region the auto-tuner can tune the number of threads that are used to execute it. Furthermore, for regions which are tilable, the tile size in each dimension is tuned.

5. Testing Methodology

To compare the different proposed approaches, we use the same objectives as in [12]. For the result S of every auto-tuning run we calculate $|S|$ and $V(S)$. S corresponds to the resulting Pareto set of the auto-tuner while $|S|$ is the number of elements in the Pareto set S. A larger number of elements of the Pareto set is considered superior, as it offers more flexibility to choose a desired solution. $V(S)$ defines the normalized size of the hypervolume covered by the performance measurements of the elements in the Pareto set S [24], i.e. the relative size of a hypervolume formed by all points dominated by the points in S in a normalized hyperrectangle defined by the highest and lowest measurement in each objective. When the function $V(S)$ is used to compare several solutions S_i where $i \in [0, I]$ and $I \in \mathbb{N}^+$, this hyperrectangle is defined by the highest respectively lowest measurement in each objective of the combined Pareto set of all solutions S_i. This means, if all configurations in S_i are dominated by configurations found in S_j with $j \in [0, I]$ and $j \neq i$, the coverage $V(S_i)$ is 0. A configuration c_0 *dominates* another configuration c_1 if c_0 delivers better performance than c_1 in every objective. As the coverage is calculated on a normalized hypervolume, the result of $V(S)$ ranges from 0 to 1 where 1 corresponds to an ideal solution covering the entire hypervolume, i.e. dominating all other solutions. For each approach we report the average population size $\overline{|S|}$ and hypervolume $\overline{V(S)}$ over 16 runs.

The huge search space of the tested programs prevents a comparison of the results of an auto-tuner to the theoretical optimum, as finding the theoretical optimum implies an exhaustive search over the entire search space.

6. Comparison Among Different Auto-Tuners

This section evaluates our approach on some exemplary test cases. We present the results obtained by different auto-tuners including random, global tuners and region-aware ones. The complete list is:

- Random: It randomly generates 3000 configurations with individual settings for each region.
- RS-GDE3 Global: It uses the RS-GDE3 tuner introduced in [12] to determine values for all tunable parameters. This version resembles is a version of the auto-tuner presented in [12] as described in Section 3.2. Every region within the entire program uses the same set of parameter values, thereby reducing the search space. Solutions are generated with the differential evolution operator described in [12].

Table 3: Search space description for the evaluated programs.

	mg	heated-plate	bt
#Regions	94	10	122
#Tilable Regions	82	4	114
#Tunable Parameters	268	18	453
Search Space Size[1]	10^{590}	10^{33}	10^{897}
Search Space Size[2]	10^{609}	10^{35}	10^{922}

[1] On Ivy Bridge-EP
[2] On Sandy Bridge-EP

- RS-GDE3 Region: Region-aware version of RS-GDE3 Global, which sets the parameter values for every region individually.
- RS-GDE3 Region GPT: Extends the RS-GDE3 Region using a global pre-tuning phase as described in Section 3. The first ten iterations are devoted to this phase.
- RS-GDE3 Recombination: Based on RS-GDE3 Region, but new configurations are generated using the recombination method described in Section 3 in every second iteration.
- RS-GDE3 Recombination GPT: RS-GDE3 Recombination with a global pre-tuning phase that uses ten iterations.

All compared RS-GDE3 auto-tuners variations use a population size of 30 and perform 100 iterations, leading to a total of about 3000 executions of the program (as performed also by the Random tuner). Our experiments did not show any significant performance improvements by enlarging the population any further with that budget of iterations.

The effectiveness of these approaches are evaluated on three benchmarks. Two of them, mg and bt, are taken from the NAS parallel benchmarks [25] C/OpenMP implementation by the Omni group [26]. bt is a block tri-diagonal solver for nonlinear partial differential equations while mg approximates the solution to a three-dimensional discrete Poisson equation using a multi-grid method. For bt we choose problem size *w*, for mg the problem size *b*, in order to get reasonable execution times for auto-tuning. The heated-plate benchmark [27] is a stencil-code solving the steady heat equation on a two dimensional, rectangular plate. The matrix size used for this benchmark was set to 384×384 elements. The total number of regions, the number of regions to which tiling can be applied as well as the total number of tunable parameters for each of those programs are listed in Table 3. This Table also indicates the search space size when tuning those programs. The number of tunable parameters is the sum of the number of tiling dimensions of all regions plus the number of regions, as we can set the number of threads separately for each region. The search space is the product of the ranges for each of those parameters. Therefore, the size of the search space depends on the target architecture, as a higher number of cores also provides more tuning possibilities. For each tuned program, we include the results of comparing the six previously described auto-tuners in Table 4. The computed Pareto set of most auto-tuners contains 30 elements, i.e. the entire population. The RS-GDE3 Recombination GPT auto-tuner delivers the best result in terms of hypervolume in all cases. The global pre-tuning phase yields a higher improvement on the Sandy Bridge-EP architecture than on the Ivy Bridge-EP which can be explained by their differing socket numbers: Whereas the Ivy Bridge-EP system has only two sockets, the Sandy Bridge-EP system has four sockets. Changing the number of threads between regions can imply an additional cache coherency overhead when the regions are executed in different number of sockets. This overhead is a consequence of not having shared cache between sockets. Therefore, it is beneficial to start with a configuration that does not change the number of threads between regions, which is achieved by the global pre-tuning phase.

The time required for the auto-tuning is dominated by the time needed to compile and execute the program. Therefore, shorter and faster programs can be tuned in less time. The tuning time for our test cases is shown in Table 5. The Random auto-tuner exhibits the longest tuning time, as it typically evaluates the configurations with the lowest performance. The fastest auto-tuner over all test cases is the RS-GDE3 Global, because it never experiences any slowdowns from performance penalties caused by region interferences. From the region-aware auto-tuners, those without a global pre-tuning phase are slower than the auto-tuners with a global pre-tuning phase in most cases. The

Table 4: Results of several auto-tuner variants for different benchmarks on two different architectures.

	Ivy Bridge-EP					
	mg		heated-plate		bt	
	$\overline{\lvert S \rvert}$	$\overline{V(S)}$	$\overline{\lvert S \rvert}$	$\overline{V(S)}$	$\overline{\lvert S \rvert}$	$\overline{V(S)}$
Random	14.9	0.0721	12.0	0.0284	26.9	0
RS-GDE3 Global	29.9	0.9058	29.1	0.6475	23.8	0.6399
RS-GDE3 Region	29.9	0.3352	29.4	0.6441	29.8	0
RS-GDE3 Region GPT	30.0	0.9013	30.0	0.6699	30.0	0.6122
RS-GDE3 Recombination	30.0	0.8465	30.0	0.6602	30.0	0.6568
RS-GDE3 Recombination GPT	30.0	0.9168	30.0	0.6857	30.0	0.7058
	Sandy Bridge-EP					
	mg		heated-plate		bt	
	$\overline{\lvert S \rvert}$	$\overline{V(S)}$	$\overline{\lvert S \rvert}$	$\overline{V(S)}$	$\overline{\lvert S \rvert}$	$\overline{V(S)}$
Random	26.2	0	4.3	0	12.4	0
RS-GDE3 Global	27.6	0.8572	29.9	0.7802	23.6	0.6148
RS-GDE3 Region	30.0	0	29.9	0.7543	23.1	0
RS-GDE3 Region GPT	30.0	0.8570	29.9	0.7975	30.0	0.6016
RS-GDE3 Recombination	30.0	0.0560	29.2	0.8065	30.0	0.5767
RS-GDE3 Recombination GPT	30.0	0.8836	29.6	0.8199	30.0	0.7101

Table 5: Tuning time in seconds of several auto-tuner variants for different benchmarks on two different architectures.

	Ivy Bridge-EP			Sandy Bridge-EP		
	mg	heated-plate	bt	mg	heated-plate	bt
Random	21945	26638	24242	31882	34774	38621
RS-GDE3 Global	16262	9605	15218	19480	15125	27274
RS-GDE3 Region	16903	12829	19187	33273	20596	42808
RS-GDE3 Region GPT	19318	10784	16639	21371	17771	28754
RS-GDE3 Recombination	19868	10839	18758	31164	15006	34038
RS-GDE3 Recombination GPT	19732	10257	18116	22686	16217	32209

latter converge faster to a population with reasonably fast configurations, which leads to significantly lower execution times of the tuned program. Similarly, the region-aware auto-tuners using the Recombination step are faster than their counterparts using only the traditional differential evolution in most cases, as the average execution time of the resulting program versions is shorter.

In addition to the Pareto set size $|S|$ and hypervolume $V(S)$ we also report the objective values of the best configuration found by the RS-GDE3 Recombination GPT auto-tuner for the three real world codes compared to the non-optimized (without auto-tuning) versions. To calculate the speedup we use the best configuration from the auto-tuner's Pareto set for each individual objective. Typically, this is a different configuration for every objective. We compare these configurations to two non-optimized configurations that do not apply any tiling: the sequential version, using only one thread and the parallel version using all threads available on the corresponding machine. The results shown in Table 6 demonstrate the superior performance compared to the non-optimized versions, both sequential and parallel, in every objective. As expected, the largest improvement over the sequential version can be achieved in wall time (up to 13.8X) while the parallel version is primarily outplayed in resource usage. Especially on the Sandy Bridge-EP architecture with 32 cores, the non-optimized version suffers from the moderate scalability of heated-plate and bt, allowing our auto-tuner to achieve an improvement factor of up to 63.7 in resources usage. These results clearly demonstrate the benefit of our region-aware multi-objective auto-tuner, even if only a single objective is of interest. This is underlined by the achieved speedup of the auto-tuner over the parallel version, which ranges from 1.3 to 4.0 on the tested architectures and programs. If a well-balanced trade-off solution across several objectives is

Table 6: Improvement over non-optimized versions i.e. not tiled with a constant number of threads, in each individual metric achieved by the RS-GDE3 Recombination GPT auto-tuner.

	Ivy Bridge-EP					
	mg		heated-plate		bt	
	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]
wall time	7.7	1.3	13.7	4.0	4.4	3.2
energy	3.2	2.4	8.1	4.5	2.1	3.6
resource usage	1.4	4.8	3.4	20.2	1.5	21.9
	Sandy Bridge-EP					
	mg		heated-plate		bt	
	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]
wall time	3.7	2.2	7.6	4.0	2.4	4.0
energy	2.2	6.0	4.5	7.5	1.6	10.6
resource usage	1.3	24.4	2.7	46.4	1.2	63.7

[1] Improvement over non-optimized sequential version.
[2] Improvement over non-optimized parallel version.

required, the benefit may be even higher, depending on the user's preferences.

7. Results Analysis

The results presented in the previous section demonstrate that the best version of our region-aware auto-tuner, the RS-GDE3 Recombination GPT, outperforms a global auto-tuner based on RS-GDE3 (RS-GDE3 Global in this paper).

When comparing the Pareto sets generated by these two tuners, we observe that some configurations in the Pareto set of the RS-GDE3 Global auto-tuner are dominated by others in the Pareto set of the RS-GDE3 Recombination GPT. This means that for each of these configurations computed by RS-GDE3 Global, the latter algorithm has found a configuration providing a shorter wall time while consuming less energy and with lower resource usage. The opposite, however, happens in very few cases. Most of the configurations found by the region-aware tuner dominate the solutions computed by the global approach, or are simply non-dominated (i.e. no solution computed by the global approach is better in all the considered objective functions).

The goal of this section is to analyze the obtained results when considering a region based approach. To this end, we perform a thorough analysis of the solutions computed by RS-GDE3 GPT. First, we analyze the configurations computed by this algorithm and the global approach (RS-GDE3 Global) in order understand how region-aware tuners exploit different parameter values in different regions of the same program. Second, we analyze how RS-GDE3 GPT performs in each region for each of the trade-off configurations it computes. Finally, we focus on analyzing the trade-off among the three objectives considered in this work.

7.1. Exploitation of Regions by Different Tuners

In this analysis, we focus on comparing the obtained results on two of the considered applications, the bt and the heated-plate. For these comparisons, we pick a program configuration computed by each of these two tuners and observe the parameter values within each region.

For the first comparison, the two program configurations are taken from the Pareto set generated by the two auto-tuners on the Sandy Bridge-EP architecture. We refer to the configuration computed by the region-aware tuner as C_r, and to the configuration computed by the global tuner as C_g. The first thing to note is that the tiling values used for different regions in C_r show a high variation. For example, while region 40 is tiled using the values {368,13,1}, the region 88 uses the tiling parameters {1,1,2}. The tile sizes used by C_g are {1,8,2} for all the regions within the program. All regions in C_g are executed with eight threads, which corresponds to the maximum number of cores on one socket on our Sandy Bridge-EP architecture. Also in C_r, the maximum number of threads used is eight, meaning that some of the regions are also executed using eight cores. However, many regions in C_r are executed with fewer

Table 7: Performance of two individual configurations in bt on the Sandy Bridge-EP architecture.

	C_g [1]	C_r [2]	Improvement
wall time (ms)	2076	2074	1.00
energy consumption (J)	148	147	1.00
resource usage (ms)	16611	13838	1.20

[1] Taken out of the Pareto set generated with the RS-GDE3 Global auto-tuner.
[2] Taken out of the Pareto set generated with the RS-GDE3 Recombination GPT auto-tuner.

Table 8: Performance of two individual configurations in heated-plate on the Ivy Bridge-EP architecture.

	C_g [1]	C_r [2]	Improvement
wall time (ms)	2323	1749	1.33
energy consumption (J)	65	63	1.04
resource usage (ms)	2323	2289	1.02

[1] Taken out of the Pareto set generated with the RS-GDE3 Global auto-tuner.
[2] Taken out of the Pareto set generated with the RS-GDE3 Recombination GPT auto-tuner.

threads, where any number between one and eight is used at least once. This results in the objective values presented Table 7. While in terms of wall time and energy consumption both configurations are similar, the resource usage of C_g is 20% higher than for C_r. This means, C_r is as fast and consumes as little energy as C_g using less resources. This is possible because the region-aware tuner found a configuration that executes regions which do scale well up to eight threads with such number of threads; at the same time, regions that do not benefit from being executed on eight cores are executed with fewer threads. Such a degree of adaption is not possible with any auto-tuner that uses the same parameter values for every region in the entire program.

For the second comparison, we present the performance figures for two program configurations for the heated-plate benchmark on the Ivy Bridge-EP architecture. Again we label C_r the configuration found by the region-aware tuner and C_g the configuration found by the global tuner. In this case, for C_g we choose the configuration with the least resources usage from the Pareto set. While C_r is not the configuration leading to the lowest resource usage in its Pareto set, it still dominates C_g. Obviously, C_g uses only one thread for every region in order to minimize the resource usage and the tile sizes used by this configuration are {1,214}. In contrast to that, C_r uses two threads to execute the biggest region of heated-plate, i.e. region 8. The increased number of threads also requires a different tile size in order to perform well; in this case it uses {31,251}. All other regions, which account for more than 1% of the total wall time, use very similar parameter values as the one used in C_g: they are executed using only one thread, and the tile size in the first dimension is equal to 1, while the tile size in the second dimension varies from 233 to 251. This indicates, that a tile size of 1 in the first dimension is beneficial when regions are executed sequentially, while a higher number of threads benefits from larger tile sizes. The combination of a custom tile size and higher number of threads for region 8 of heated-plate results in a significantly lower wall time as shown in Table 8. As that region does scale well, C_r has also a slight advantage over C_g in both, resource usage and energy consumption, despite the increased number of cores used. Additionally, as indicated in the table, the rather different parameter values for region 8 cause a 25% drop in wall time, compared to C_g. As in the comparison before, these performance figures can only be achieved using different parameter values for the individual regions of the program.

7.2. Performance Analysis of Regions across Different Program Configurations

In [12], the idea of computing a Pareto set for each of the individual regions that form part of the program to be tuned was introduced. The proposed approach implies (manually or automatically) selecting among the computed trade-off parameter values for each of these regions in order to build a valid program configuration. As commented in related work ([2]), and empirically shown in Section 2, dependencies among regions may imply, however, that parameter values that perform well for a region when executed in isolation do not perform well when the region is executed as part of a whole program (with possibly other regions executed before and after it). This problem is not specific to the multi-objective case, as shown in Section 2, it also takes place when a single objective is considered. The

goal of this section is to analyze how regions behave within different (quasi-)Pareto optimal program configurations (i.e. the ones computed by the RS-GDE3 Recombination GPT).

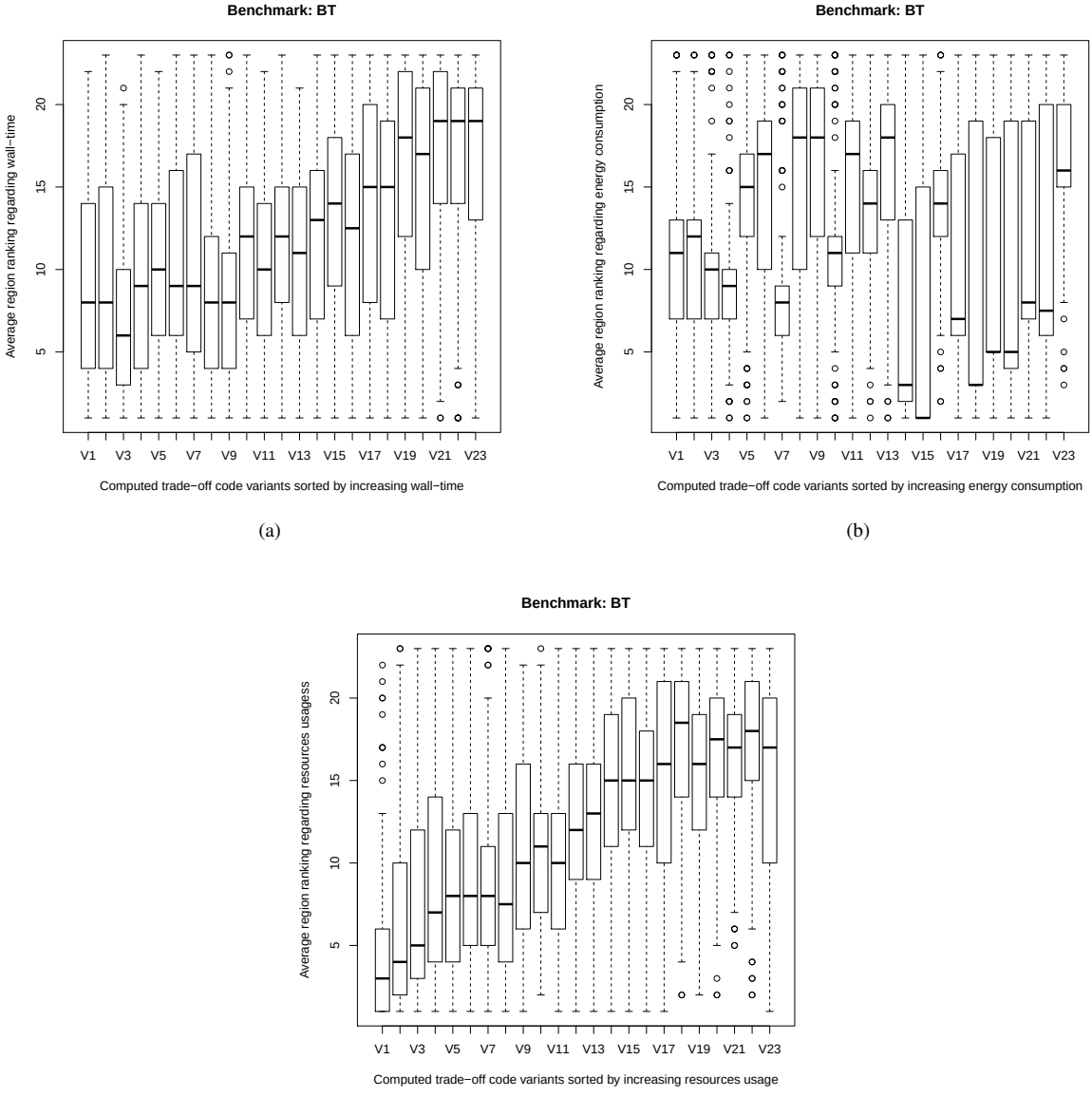
For the analysis in this section, we made use of boxplot representations (see figures 4, 5, 6, 7). Each single plot corresponds to one of the evaluated benchmark problems when executed on one of the two considered architectures. The horizontal axis represents one of the computed program configurations in one of the performed runs. These program configurations are sorted regarding one out of the three considered objectives. The vertical axis represents how these program configurations perform in the different individual regions regarding the rest of program configurations. As an example, the leftmost box in Figure 4a, which corresponds to the solution with lowest wall time, shows that for only 25% of the regions (first quartile), that program configuration has been among the four fastest ones; that program configuration has also been among the eight fastest ones in 50% of the regions (median) composing it; that configuration has been slower that the 14 fastest ones for 25% of the regions (see third quartile). This plot shows, therefore, that the fastest computed program configuration is not the fastest on every individual region; instead, there are regions within this program configuration with a poor performance regarding other program configurations.

We start the analysis with the bt application depicted in the figures 4 and 5 for the Ivy Bridge-EP and Sandy Bridge-EP architectures, respectively. The issue commented before is clearly visible in both architectures. Each computed program configuration, independently of the considered objective function, performs very differently on the different regions composing it. If we look at the 50% of the regions for which each program configuration perform better regarding other program configurations, we observe a clear trend for wall time and resource usage: the better a program configuration performs regarding these objectives, the better that configuration also performs in these regions. This trend is observable in the two considered architectures. The way to interpret this result is: if we want to optimize this benchmark for wall time or resource usage, it is more likely that most of the regions composing it should be individually optimized for these objectives as well. If we focus on energy consumption, this trend is not observable. Indeed, it is possible to see that several program configurations which overall perform poorly in terms of energy consumption, provides an excellent performance on individual regions, for the sake of making other regions consuming extra energy—therefore deteriorating the overall energy consumption of the program. For example, let us compare the configurations labeled as $V1$ and $V20$ from the plot 4b. The former configuration provides the lowest energy consumption for the bt application when executed on the Ivy Bridge-EP architecture, while the latter is the solution providing the 20th highest energy consumption for the same problem in the same machine. $V1$ consumes more energy in 50% of the regions than at least 10 other program configurations, despite providing the best overall energy consumption. On the contrary, $V20$ is the fifth best or better solution in terms of energy consumption in 50% of the regions and, at least the 4th best in 25% of of them.

The results obtained for the mg problem are very similar to the bt case if we look at wall time and resource-usage: the same trend appears in these cases. No trend or conclusion can be obtained if we focus on energy consumption. Due to these similarities, we omit the figures for this application.

Finally, we pay attention to the benchmark problem consisting of the smallest search space (and, hence, theoretically simpler to solve). In this case, no clear trend can be observed for any of the objective functions (see figures 6, and 7). It is remarkable, in the case of the Ivy Bridge-EP architecture, how the different program configurations perform similarly in the regions composing them when looking at energy consumption and resources usage for this architecture.

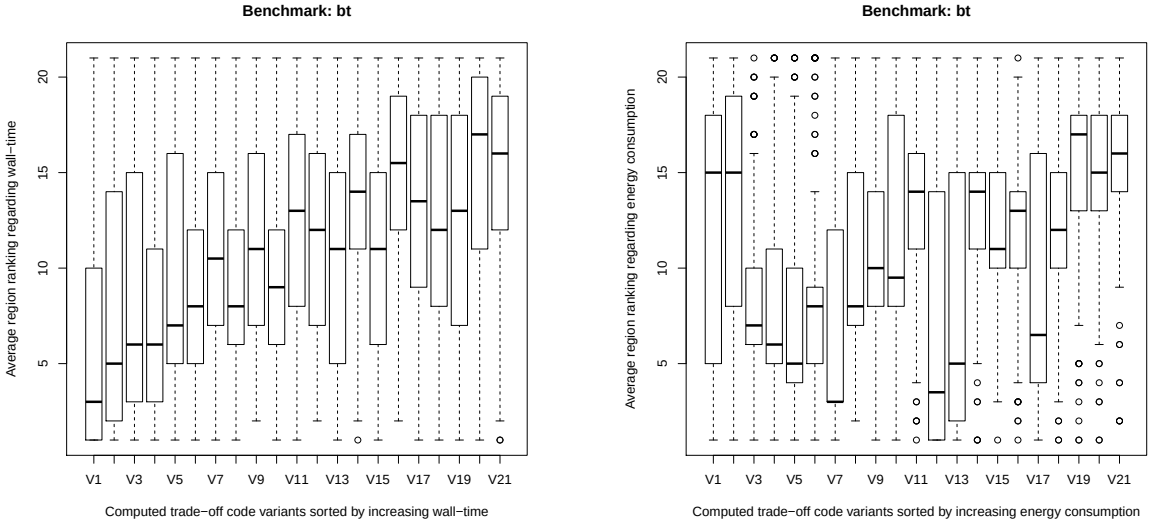
Overall, this section analyzes how program configurations perform in the code regions. The obtained results showed that it is not possible to establish any kind of pattern among the regions within the same configurations: a configuration may perform exceptionally good in some regions and exceptionally poor in others. In this context, it is hard to establish any kind of correlation among the performance of a program configuration and individual regions. In some cases, like the bt or mg benchmark problems, such a correlation indeed exist if we focus on a subset of code regions in the case of wall time and resources usage. In the case of energy consumption, none of the examined problems have shown any correlation among the behavior of individual regions and the whole program configuration. These results demonstrate once again the challenges of optimizing problems composed of several regions and the need for auto-tuners able of deal with the dependences between them. They also demonstrate that the initial idea proposed in [12] cannot be realized, because there is no similarities between the performance of the whole program and code regions. In other words, a trade-off program configuration is composed of regions being optimal regarding very different preference vectors.



(a) (b)

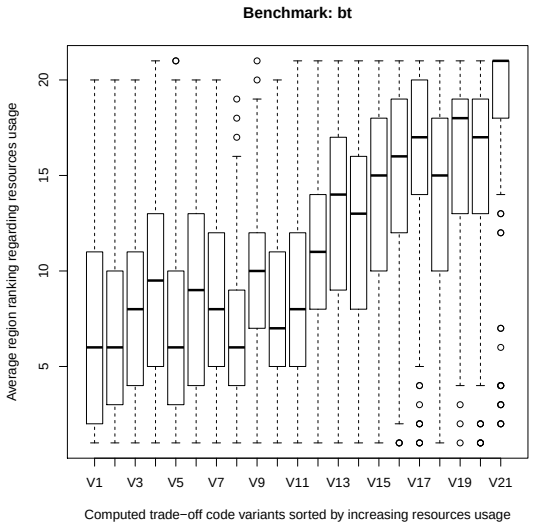
(c)

Figure 4: Behavior of the Computed Trade-off Configurations on Each Program Region (benchmark: bt; architecture: Ivy Bridge-EP)



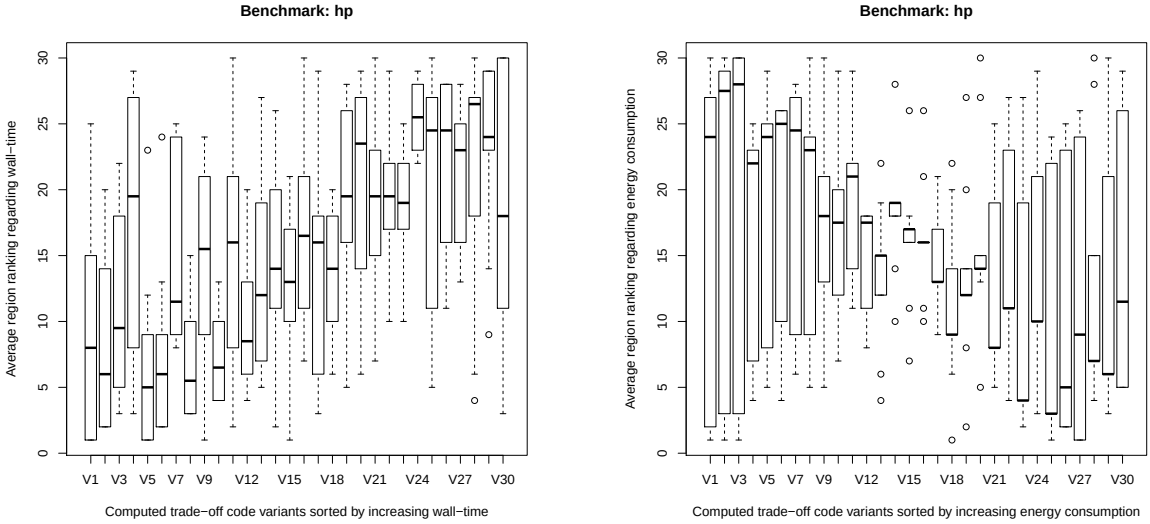
(a)

(b)



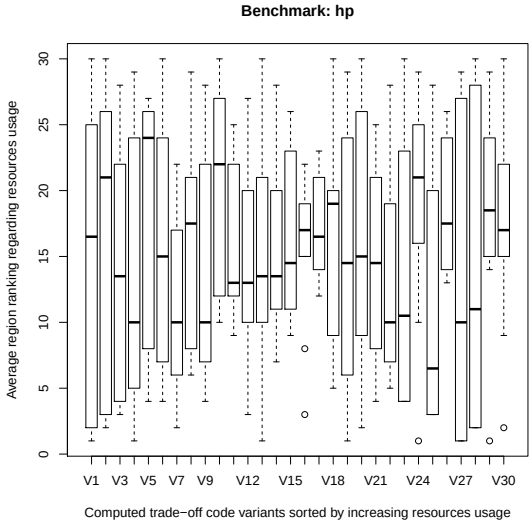
(c)

Figure 5: Behavior of the Computed Trade-off Configurations on Each Program Region (benchmark: bt; architecture: Sandy Bridge-EP)



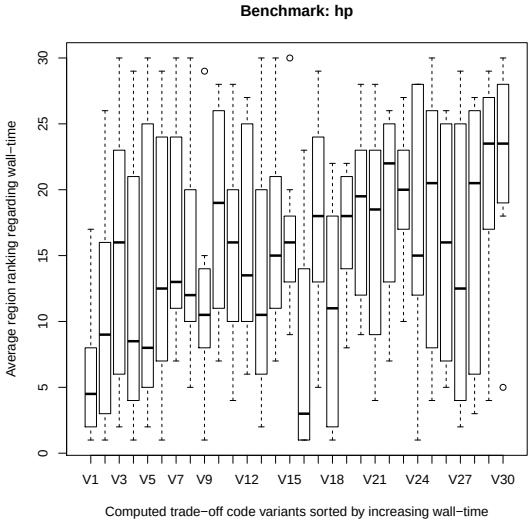
(a)

(b)

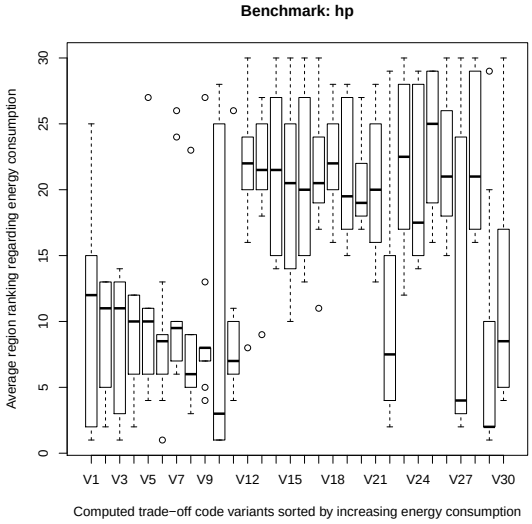


(c)

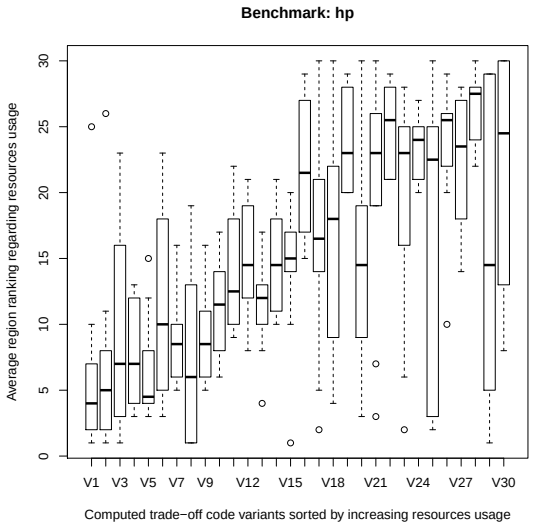
Figure 6: Behavior of the Computed Trade-off Configurations on Each Program Region (benchmark: hp; architecture: Ivy Bridge-EP)



(a)



(b)



(c)

Figure 7: Behavior of the Computed Trade-off Configurations on Each Program Region (benchmark: hp; architecture: Sandy Bridge-EP)

7.3. Analysis of the computed Trade-off solutions

The goal of this section is to analyze the computed trade-off program configurations in the two considered architectures. These configurations are depicted in the form of *star-diagrams* in the figures 8 and 9 for the analyzed Ivy Bridge-EP and Sandy Bridge-EP machines respectively. One star-diagram is used for each benchmark problem. In these diagrams, each of the equiangular spokes represents one configuration. For each of these configurations, the three analyzed criteria wall time, energy consumption and resource usage are depicted in blue, red, and yellow, respectively, after normalization.

We start the discussion with the machine featuring the Ivy Bridge-EP architecture. The first noticeable fact across the three evaluated benchmarks in this machine is the high correlation between resource usage and energy consumption (yellow and red colors, respectively) of the different program configurations. In other words, solutions which are characterized by high usage of resources also entail a high energy consumption. In the case of wall time, its correlation regarding the aforementioned two objectives is not that obvious. In most of the cases, however, low wall time values correspond with high values of energy consumption and resource usage. Therefore, in this machine our multi-objective auto-tuning problem could be simplified to simply optimize two objectives: wall time and one of either energy-consumption or resource usage.

In the case of the Sandy Bridge-EP architecture, the obtained trade-off among the three considered objective functions is more complex to analyze. On the bt and mg benchmarks, there are many program configurations for which optimizing wall time also means optimizing for energy consumption. These two objectives seem to conflict with optimizing resource usage in most of these program configurations. In the case of the hp problem, almost half of the computed configurations exhibit a highly negative correlation between wall time and the other two objectives (energy consumption and resource usage). The rest of the computed configurations provide the lowest values of resource usage at the cost of requiring more energy and wall time.

8. Related Work

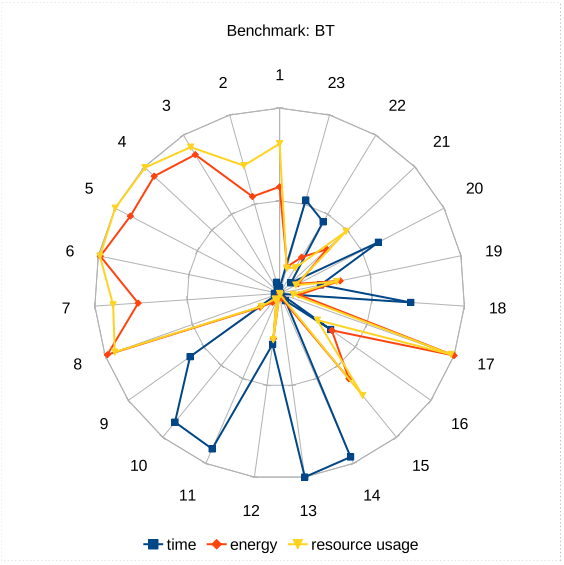
In the literature we find several frameworks for software auto-tuning, for example self-tuning libraries like ATLAS [3], OSKI [4], SPIRAL [5] or FFTW [6], or other auto-tuning frameworks including Active Harmony [7], Sequoia [28], PetaBricks [8, 9], Patus [10], and OpenTuner [11].

In the past, most auto-tuners focused on improving the wall time of programs. However, recent work shows an inarguable attention to tune applications regarding several objectives. Besides wall time, energy consumption entailed by a program's execution is becoming a popular objective [29, 30, 31, 32, 33, 34, 35, 2, 36]. Resource usage [12, 36], compilation time, or the size of the executable binary [37, 38, 39] also received attention in related work. Most of these works fail to capture the trade-off between these objectives and reduce them to a single one. Only a few works focus on computing and analyzing the trade-off between several conflicting objectives [40, 12, 41].

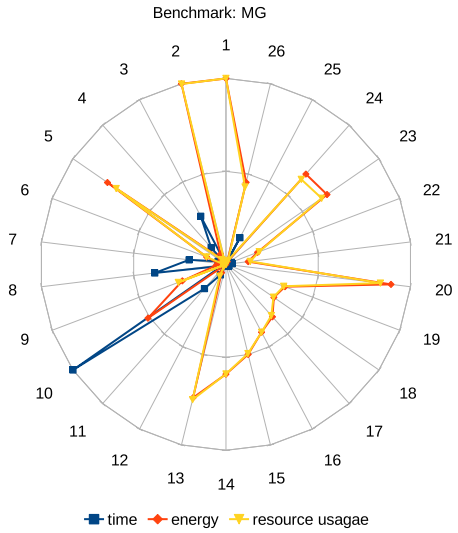
All the aforementioned works applied the same tuning options to the whole program. Approaches that tune code regions individually and examine their inter-relationships with respect to single or multiple objectives are rare. A major issue is the definition of code regions for programs. In [42] program functions are considered to be the regions to tune. In MPI programs regions are often defined as the code between pairs of communication directives in [2]. In [43] regions are obtained from applications Regions within the same cluster are tuned using the same parameters or code transformations. The Periscope tool of the AutoTune project[1] tunes regions regarding any function measuring properties of that function (run-time, energy consumed, etc.). Although different objectives can be tuned, they are not considered simultaneously. Furthermore, Periscope tunes regions individually without considering side effects among regions. In contrast to the framework presented in this work, Periscope does not describe a methodology to identify regions within a program.

In [42, 43], programs are split into several regions which are tuned in isolation. However, the authors of [2] show that regions within a program impact each other's execution time behavior. They demonstrate that when regions are executed with the best set of parameter values known so far, a non-negligible penalty may be paid as a result of changing hardware settings across adjacent regions. The same work also discusses the benefits of using the same set of parameter values for every region in the entire program versus a per-region tuning approach, and the need for tuning mechanisms that can find configurations aware of interferences between regions.

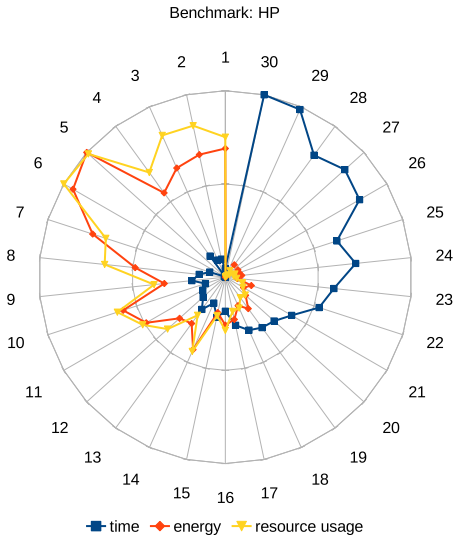
[1] `http://www.autotune-project.eu/`



(a)

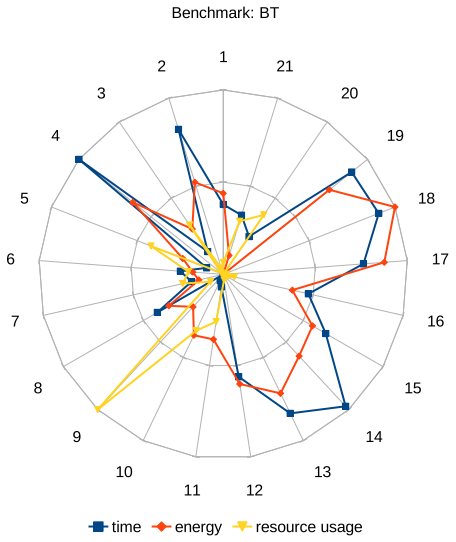


(b)

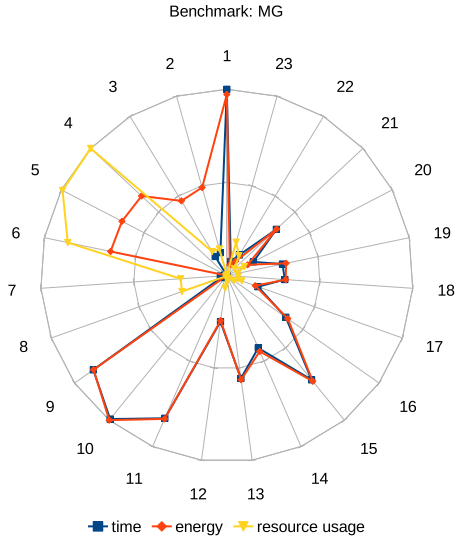


(c)

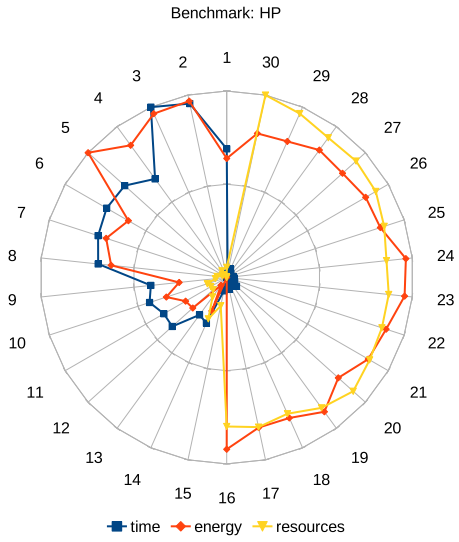
Figure 8: Wall time vs Energy Consumption vs Resource Usage of the Computed Trade-off Program Configurations (Behavior of the Computed Trade-off Configurations on Each Program Region (architecture: Ivy Bridge-EP)



(a)



(b)



(c)

Figure 9: Wall time vs Energy Consumption vs Resource Usage of the Computed Trade-off Program Configurations (Behavior of the Computed Trade-off Configurations on Each Program Region (architecture: Sandy Bridge-EP)

9. Conclusion

Most existing work on auto-tuning focuses on a global setting of parameter values which are fixed for the entire program, ignoring the optimization potential by customizing parameter values to individual region's peculiarities. In this paper, we introduced a novel auto-tuning framework that is based on a source-to-source compiler, a runtime system and a new RS-GDE3 auto-tuner variation to provide a solution for multi-region, multi-objective auto-tuning. The challenges introduced by the huge search space, region dependencies and conflicting objectives are tackled by adding a pre-tuning phase to the region-aware auto-tuner which tunes the program using the same parameter values for all regions, as well as an intermediate evolutionary step for the RS-GDE3 auto-tuner, that generates new configurations by recombining the parameter values generated in previous steps.

Experiments have shown that our new approach is more effective in tuning three different programs on two different parallel computers than non-region-aware global auto-tuning. We outperform a non-region-aware RS-GDE3 auto-tuner in hypervolume $V(S)$ by up to 15%. We demonstrated that our approach reaches up to 7.6, 10.5 and 61.6X improvements in wall time, energy consumption and resource usage respectively, over the non-optimized parallel version.

Furthermore, we provide a thorough analysis of how the regions composing a program behave within different (quasi-)optimal program configurations. Our study reveals strong dependencies between regions. In particular, we show how program configurations which are optimal for a given criterion may perform worse than other program configurations for almost each individual region regarding that criterion.

Future work will devote attention to the design of tuners (based either in search or machine learning) able to identify and better exploit region dependencies. In addition, the identification of regions for programs not written in the OpenMP language extension needs to be explored.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603.

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

[2] D. Li, B. R. de Supinski, M. Schulz, D. S. Nikolopoulos, K. W. Cameron, Strategies for Energy-Efficient Resource Management of Hybrid Programming Models., IEEE Trans. Parallel Distrib. Syst. 24 (1) (2013) 144–157.
URL http://dblp.uni-trier.de/db/journals/tpds/tpds24.html#LiSSNC13

[3] R. C. Whaley, J. J. Dongarra, Automatically Tuned Linear Algebra Software, in: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 1–27.
URL http://dl.acm.org/citation.cfm?id=509058.509096

[4] R. Vuduc, J. W. Demmel, K. A. Yelick, OSKI: A Library of Automatically Tuned Sparse Matrix Kernels, Journal of Physics: Conference Series 16 (1) (2005) 521.
URL http://stacks.iop.org/1742-6596/16/i=1/a=071

[5] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code Generation for DSP Transforms, Proceedings of the IEEE 93 (2) (2005) 232–275. doi:10.1109/JPROC.2004.840306.

[6] M. Frigo, A Fast Fourier Transform Compiler, SIGPLAN Not. 34 (5) (1999) 169–180. doi:10.1145/301631.301661.
URL http://doi.acm.org/10.1145/301631.301661

[7] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth, Active Harmony: Towards Automated Performance Tuning, in: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC '02, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–11.
URL http://dl.acm.org/citation.cfm?id=762761.762771

[8] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe, PetaBricks: A Language and Compiler for Algorithmic Choice, SIGPLAN Not. 44 (6) (2009) 38–49. doi:10.1145/1543135.1542481.
URL http://doi.acm.org/10.1145/1543135.1542481

[9] S. Amarasinghe, PetaBricks: A Language and Compiler Based on Autotuning, in: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, ACM, New York, NY, USA, 2011, pp. 3–3. doi:10.1145/1944862.1944865.
URL http://doi.acm.org/10.1145/1944862.1944865

[10] M. Christen, O. Schenk, H. Burkhart, PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures, in: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 676–687. doi:10.1109/IPDPS.2011.70.
URL http://dx.doi.org/10.1109/IPDPS.2011.70

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. M. O'Reilly, S. Amarasinghe, OpenTuner: An Extensible Framework for Program Autotuning, in: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), 2014, pp. 303–315. doi:10.1145/2628071.2628092.

[12] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, H. Moritsch, A Multi-Objective Auto-Tuning Framework for Parallel Codes, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 10:1–10:12.
URL http://dl.acm.org/citation.cfm?id=2388996.2389010

[13] C. Coello, D. Van Veldhuizen, G. Lamont, Evolutionary Algorithms for Solving Multi-Objective Problems, Genetic Algorithms and Evolutionary Computation, Springer US, 2013.
URL https://books.google.de/books?id=VmnTBwAAQBAJ

[14] J. Handl, S. C. Lovell, J. Knowles, Multiobjectivization by Decomposition of Scalar Cost Functions, in: G. Rudolph, T. Jansen, N. Beume, S. Lucas, C. Poloni (Eds.), Parallel Problem Solving from Nature – PPSN X, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 31–40.

[15] J. J. Durillo, A. J. Nebro, C. A. C. Coello, J. Garcia-Nieto, F. Luna, E. Alba, A Study of Multiobjective Metaheuristics When Solving Parameter Scalable Problems, IEEE Transactions on Evolutionary Computation 14 (4) (2010) 618–635. doi:10.1109/TEVC.2009.2034647.

[16] R. Storn, K. Price, Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, J. of Global Optimization 11 (4) (1997) 341–359. doi:10.1023/A:1008202821328.
URL http://dx.doi.org/10.1023/A:1008202821328

[17] J. Durillo, A. Nebro, F. Luna, C. Coello Coello, E. Alba, Convergence Speed in Multi-objective Metaheuristics: Efficiency Criteria and Empirical Study, International Journal for Numerical Methods in Engineering 84 (11) (2010) 1344–1375.

[18] Insieme Source Code Repository, https://github.com/insieme/insieme/tree/inspire_1.3 (2017).

[19] H. Jordan, Insieme: A Compiler Infrastructure for Parallel Programs, Ph.D. thesis.

[20] P. Thoman, Insieme-RS: A Compiler-Supported Parallel Runtime System, Ph.D. thesis, University of Innsbruck (7 2013).

[21] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring Energy Consumption for Short Code Paths Using RAPL, SIGMETRICS Perform. Eval. Rev. 40 (3) (2012) 13–17. doi:10.1145/2425248.2425252.
URL http://doi.acm.org/10.1145/2425248.2425252

[22] OpenMP Architecture Review Board, OpenMP Application Program Interface, http://www.openmp.org/mp-documents/OpenMP3.1.pdf (2011).

[23] P. Feautrier, C. Lengauer, Encyclopedia of Parallel Computing, Springer, 2011, Ch. Polyhedron Model, pp. 1581–1592. doi:http://dx.doi.org/10.1007/978-0-387-09766-4.

[24] E. Zitzler, L. Thiele, Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach, Trans. Evol. Comp 3 (4) (1999) 257–271. doi:10.1109/4235.797969.
URL http://dx.doi.org/10.1109/4235.797969

[25] NASA, The NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/ (2016).

[26] Omni OpenMP Compiler, http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html (2016).

[27] J. Burkardt, HEATED_PLATE_OPENMP, http://people.sc.fsu.edu/~jburkardt/c_src/heated_plate_openmp/heated_plate_openmp.html (Mar. 2016).

[28] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, P. Hanrahan, Sequoia: Programming the Memory Hierarchy, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, ACM, New York, NY, USA, 2006. doi:10.1145/1188455.1188543.
URL http://doi.acm.org/10.1145/1188455.1188543

[29] J. H. Laros, III, K. T. Pedretti, S. M. Kelly, W. Shu, C. T. Vaughan, Energy Based Performance Tuning for Large Scale High Performance Computing Systems, in: Proceedings of the 2012 Symposium on High Performance Computing, HPC '12, Society for Computer Simulation International, San Diego, CA, USA, 2012, pp. 6:1–6:10.
URL http://dl.acm.org/citation.cfm?id=2338816.2338822

[30] M. Rahman, L.-N. Pouchet, P. Sadayappan, Neural Network Assisted Tile Size Selection, in: International Workshop on Automatic Performance Tuning (IWAPT'2010), Springer Verlag, Berkeley, CA, 2010.

[31] A. Tiwari, M. A. Laurenzano, L. Carrington, A. Snavely, Auto-tuning for Energy Usage in Scientific Applications, in: M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, J. Weidendorfer (Eds.), Euro-Par 2011: Parallel Processing Workshops, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 178–187.

[32] S. M. F. Rahman, J. Guo, A. Bhat, C. Garcia, M. H. Sujon, Q. Yi, C. Liao, D. Quinlan, Studying the Impact of Application-level Optimizations on the Power Consumption of Multi-core Architectures, in: Proceedings of the 9th Conference on Computing Frontiers, CF '12, ACM, New York, NY, USA, 2012, pp. 123–132. doi:10.1145/2212908.2212927.
URL http://doi.acm.org/10.1145/2212908.2212927

[33] R. Springer, D. K. Lowenthal, B. Rountree, V. W. Freeh, Minimizing Execution Time in MPI Programs on an Energy-constrained, Power-scalable Cluster, in: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, ACM, New York, NY, USA, 2006, pp. 230–238. doi:10.1145/1122971.1123006.
URL http://doi.acm.org/10.1145/1122971.1123006

[34] J. Peraza, A. Tiwari, M. Laurenzano, L. Carrington, A. Snavely, PMaC's Green Queue: A Framework for Selecting Energy Optimal DVFS Configurations in Large Scale MPI Applications, Concurrency and Computation: Practice and Experience 28 (2) (2016) 211–231. doi:10.1002/cpe.3184.
URL http://dx.doi.org/10.1002/cpe.3184

[35] Y. Dong, J. Chen, X. Yang, L. Deng, X. Zhang, Energy-Oriented OpenMP Parallel Loop Scheduling, in: 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, 2008, pp. 162–169. doi:10.1109/ISPA.2008.68.

[36] P. Gschwandtner, J. J. Durillo, T. Fahringer, Multi-Objective Auto-Tuning With Insieme: Optimization and Trade-Off Analysis for Time,

Energy and Resource Usage, Springer International Publishing, Cham, 2014, pp. 87–98. doi:10.1007/978-3-319-09873-9_8.
URL http://dx.doi.org/10.1007/978-3-319-09873-9_8

[37] K. Hoste, L. Eeckhout, Cole: Compiler Optimization Level Exploration, in: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08, ACM, New York, NY, USA, 2008, pp. 165–174. doi:10.1145/1356058.1356080.
URL http://doi.acm.org/10.1145/1356058.1356080

[38] L. Luo, Y. Chen, C. Wu, S. Long, G. Fursin, Finding Representative Sets of Optimizations for Adaptive Multiversioning Applications, arXiv preprint arXiv:1407.4075.

[39] G. Fursin, A. Memon, C. Gillon, A. Lokhmotov, Collective mind, part ii: Towards performance- and cost-aware software engineering as a natural science, in: 18th International Workshop on Compilers for Parallel Computing (CPC15), 2015.

[40] V. W. Freeh, D. K. Lowenthal, Using Multiple Energy Gears in MPI Programs on a Power-scalable Cluster, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, ACM, New York, NY, USA, 2005, pp. 164–173. doi:10.1145/1065944.1065967.
URL http://doi.acm.org/10.1145/1065944.1065967

[41] P. Balaprakash, A. Tiwari, S. M. Wild, Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy, in: S. A. Jarvis, S. A. Wright, S. D. Hammond (Eds.), High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation, Springer International Publishing, Cham, 2014, pp. 239–260.

[42] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, J. W. Davidson, Practical Exhaustive Optimization Phase Order Exploration and Evaluation, ACM Trans. Archit. Code Optim. 6 (1) (2009) 1:1–1:36. doi:10.1145/1509864.1509865.
URL http://doi.acm.org/10.1145/1509864.1509865

[43] P. de Oliveira Castro, Y. Kashnikov, C. Akel, M. Popov, W. Jalby, Fine-grained Benchmark Subsetting for System Selection, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, ACM, New York, NY, USA, 2014, pp. 132:132–132:142. doi:10.1145/2544137.2544144.
URL http://doi.acm.org/10.1145/2544137.2544144