Optimizing Performance of Stencil Code with SPL Conqueror

Alexander Grebhahn,
Norbert Siegmund, Sven Apel
University of Passau
Passau, Germany

Sebastian Kuckuk, Christian Schmitt,
Harald Köstler
University of Erlangen-Nuremberg
Erlangen, Germany

ABSTRACT

A standard technique to numerically solve elliptic partial differential equations on structured grids is to discretize them via finite differences and then to apply an efficient geometric multi-grid solver. Unfortunately, finding the optimal choice of multi-grid components and parameters is challenging and platform dependent, especially, in cases where domain knowledge is incomplete. Auto-tuning is a viable alternative, but faces the problem of large configuration spaces and feature interactions. To improve the state of the art, we explore whether recent work on configuration optimization in product lines can be applied to the stencil-code domain. In particular, we extend and use the domain-independent tool SPL Conqueror in a series of experiments to predict the performance-optimal configurations of two geometric multi-grid codes: a program using the HIPA^{cc} framework and an evaluation prototype called HSMGP. For HIPA^{cc}, we can predict the performance of all configurations with an accuracy of 98 %, on average, when measuring 57.5 % of the configurations, and we are able to predict a configuration that is close to the optimal one after measuring only less than 4 % of all configurations. For HSMGP, we can predict the performance with an accuracy of 88 % when measuring 11 % of all configurations.

Keywords

Stencil computations, parameter optimization, auto-tuning, product lines, SPL Conqueror

1. INTRODUCTION

In many areas of computation, large linear or nonlinear systems have to be solved. Geometric multi-grid is one method to solve such systems that have a certain structure, e. g., that arise from the discretization of partial differential equations (PDEs) on structured grids and lead to sparse and symmetric positive definite system matrices. For good introductions and a comprehensive overview on multi-grid methods, we refer to [6, 20]. Algorithmically, most of the multi-grid components are functions that sweep over a computational grid and perform nearest neighbor computations. Mathematically, these computations are linear-algebra operations, such as matrix-vector products. Since the matrices are sparse and contain often similar entries in each row, they can be described by a *stencil*, where one stencils represents one row in the matrix.

A multi-grid algorithm consists of several components and traverses a hierarchy of grids several times until the linear system is solved up to a certain accuracy. As already mentioned, the components and also the parameters, such as how many times a certain component is applied on each level, are highly problem and platform dependent. That is, depending on the hardware and the application scenario, some settings perform faster than others. This gives rise to a considerable number of *configuration options* to customize multi-grid algorithms.

Selecting configuration options (i. e., specifying a *configuration*), to maximize performance is an essential activity in exascale computing [7]. If we use a non-optimal configuration, we may not exploit the full computational power, leading to increased cost and time. However, identifying the performance-optimal configuration is a complex task. Measuring performance of all configurations to determine the fastest one does not scale, because the number of configurations may grow exponentially with the number of configurations options. Alternatively, we can use domain knowledge to identify the fastest configuration. However, domain knowledge may not always be available, and domain experts are rare and expensive.

In product-line engineering, different approaches have been developed to tackle the problem of finding optimal configurations [9, 16, 17]. The idea is to measure some configurations of a program and *predict* the performance of all other configurations (e. g., by using machine-learning techniques).

Our goal is to find out whether existing product-line techniques can be applied for automatic stencil-code optimization. In particular, we use the tool SPL Conqueror by Siegmund et al. [17] to determine the performance influence of configuration options of stencil code implementations. The general idea is to determine the performance influence of individual configuration options first, and to consider interactions between them subsequently. To this end, we use a heuristic with which we learn the performance contribution of numerical parameters with the help of functions learning.

Overall, we make the following contribution: We demonstrate that SPL Conqueror can be applied to the stencil domain to identify an optimal configuration after performing

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

a small number of measurements. After performing more measurements, we can predict the performance of all configurations with an accuracy of about 88 % in average.

We demonstrate applicability of our approach with two case studies from the stencil domain. One of the systems investigated is a geometric multi-grid implementation using HIPA^{cc} [14], a framework for generating GPU code. The other system is a prototype of a highly scalable geometric multi-grid solver, developed for testing various algorithms and data structures on high-performance computing systems.

Our experiments show that there are a huge number of interactions between configuration options having considerable influence on performance. However, we can predict a configuration with a good performance even with a small number of measurements.

2. PRELIMINARIES

In this section, we present preliminaries of our approach of finding performance-optimal stencil-code configurations. Since our approach stems from the product-line domain, we introduce respective terminology and provide background information on how to model variability.

2.1 Feature Models

Customization options of variable software systems are often called *features* [11]. Since features may depend on or exclude each other, we use *feature models* to describe a set of valid combinations [2]. In detail, a feature model describes relationships among the features of a configurable system. As examples, we present the feature models of our two subject systems in Figure 1 and Figure 2.

A feature can either be *mandatory* (i. e., required in all configurations where its parent feature is selected), *optional*, or be part of an *Or-group* or an *Xor-group*. If the parent feature of a group is selected, exactly one option of an Xor-group and at least one feature of an Or-group has to be selected. Furthermore, we can define arbitrary propositional formulas to further constrain the variability. For example, one feature may *imply* or *exclude* another one.

Extended Feature Models. Standard feature models can express only the presence or absence of features in a configuration; we refer to these feature as *Boolean features*. Unfortunately, this is insufficient when modeling variability of arbitrary options exposed by stencil code. To overcome this limitation, *extended* or *attributed feature models* have been proposed [3, 4]. These models are extended with possible non-Boolean attributes (*Parameters*), describing properties or special characteristics. To model stencil-code parameters, we use a notation in which attributes have a domain type (i. e., the definition range of the parameter) and a default value. An example of a feature attribute is *Padding* (see Figure 1a). The type of this parameter is "Integer, between 0 and 512" with a step size of 32, and a default value of 0.

2.2 Predicting Performance of Customizable Programs

To predict the performance of configurations of a customizable program, Siegmund et al. propose the SPL Conqueror approach that quantifies the influence of individual features on performance [17, 18]. To this end, they propose several heuristics that assess the individual performance contributions to predict the total performance of a given configuration.

Heuristics. The first heuristic, *feature-wise (FW)*, measures the performance influence of each individual feature. For each feature, two configurations – one with and one without the regarded feature – are measured. The performance difference is interpreted as the performance contribution of the feature in question. With this heuristic, the number of required measurements grows linearly with the number of configuration options. As a consequence, this heuristic can also be applied to huge configuration spaces. A drawback, however, is that the FW-based prediction does not always correspond to the actual performance, because features may influence each other. This impact on performance is called *feature interaction* [17]. For example, effects from changing the used *Texture Memory* can vary based on the current *API* (see Figure 1). There can be even interactions between more than two features. To account for these different orders of interactions (i. e., the number of interacting features), Siegmund et al. propose three further heuristics [17].

The first heuristic considers interactions between all pairs of features (i. e., order of one), called *pair-wise heuristic (PW)*. To measure interactions of a higher order, the *higher-order heuristic (HO)* is used. Lastly, in some customizable programs there are features interacting with many other. To consider the contribution of such *hot-spot features*, we can apply the *hot-spot heuristic (HS)*. The three heuristics considering interactions build on each other and on the FW heuristic. As a consequence, the HS heuristic uses all measurements performed by the FW, PW and HO heuristic.

The heuristics of Siegmund et al. can predict performance contributions of Boolean features only. As a consequence, it is not possible to incorporate parameters. To overcome this limitation, we use a *function-learning heuristic (FL)*. The basic idea is to learn performance-contribution functions for each parameter (non-Boolean attribute). Since each parameter can have a performance influence on different features, we learn multiple functions per parameter. The polynomial of the performance function has to be given by a domain expert or learned by a machine-learning approach. Consequently, a feature model with n features and m parameters requires $n \cdot m$ functions. To learn these *performance-contribution functions*, we sample the type of the parameter (i. e., we select values from the domain) and measure performance of the corresponding configurations. Using a least-squares approach, we determine the contribution of the parameter. When sampling a single parameter, we keep the remaining parameters constant and use their default values.

3. VARIABILITY IN THE MULTI-GRID DOMAIN

In general, a (standard) multi-grid cycle can be defined as follows: The algorithm starts at the finest level. Firstly, high-frequency errors are smoothed with a fixed number of pre-smoothing steps. Afterwards, to get rid of the low-frequency errors, the residual is calculated, restricted to the next coarser level, and then solved for recursively. Next, the solution from this process is propagated onto the current level and used to correct the solution. Lastly, the error is smoothed again by applying a fixed number of post-smoothing steps.

Obviously, the recursion has to be stopped at some point, at the latest when there is only one unknown left. In this case, direct and exact solving is possible and feasible. In the context of large-scale applications, however, this is not practicable and thus the cycle is stopped before, usually when only a few unknowns are left on each compute unit, and a specialized coarse grid solver is employed. This solver is usually chosen according to the requirements arising from the domain structure, the problem description, and performance considerations.

In multi-grid computations, different types of variabilities arise, which can be grouped according to six criteria.

1. A suitable *hardware platform* and concomitant *external software components* should be chosen. Here, hardware choices include the number of compute units and their type (i. e., CPUs, GPUs, other accelerators or even a combination of them). Concerning software, different compilers and abstraction layers for hardware access, parallelization, and inter-process communication are possible. They include, CUDA, OpenCL, OpenMP, as well as a number of MPI implementations.
2. The algorithmic and numeric components can be adapted. To this end, the *cycle type*, basically, a description of how and when the recursion is performed, can be chosen. The most prominent choices are *V-cycle* and *W-cycle*. Furthermore, different components of the multi-grid cycle can be exchanged, usually only targeting the *smoother* and the *coarse grid solver*. Yet, in general, altering the *restriction* and *propagation operators* is possible as well.
3. Different parameters can be tuned, where these are usually described through numerical values. Examples are the *number of pre-* or *post-smoothing steps* and the *smoothing parameter* ω. Usually, these parameters are limited in the range of values they can take, and further constraints, such as a restriction to integer values, may apply.
4. Optional optimizations can be added on demand. These include basic optimization strategies, such as *padding*, *vectorization*, *(software) prefetching*, *tiling*, and many more. As a detailed overview of possible techniques is beyond the scope of this work, please be referred to [10, 12] for further reading.
5. There is also variability in the problem to be solved. This, however, is usually not controllable from the optimization process, as it is fixed by the applications. Nevertheless, impacts on the (performance) characteristics of the components can be quite prominent and highly diverse. The choices of the PDE to be solved and the applied *boundary conditions* fall into this category, both of which mostly influence the computational complexity. Additionally, different *geometric properties* (i. e., a uniform domain in contrast to a general block-structured domain) are common and influence mostly the communication behavior.
6. Lastly, there are various other changes that could be of interest including different *discretization* schemes (e. g., finite elements instead of the presented finite differences).

Although the range of adjustments is quite broad, their impacts can basically be divided into two groups, namely *convergence* and *performance* impacts. Roughly said, convergence describes how many iterations are required to achieve a satisfyingly accurate solution to the given problem, and performance describes how much time one iteration takes. In our context, most high-level decisions influence both, however often in opposing matter (i. e., the number of iterations decreases while the time per iteration increases or vice versa). In contrast, low-level optimizations typically only influence performance behavior, since the algorithmic layout, and thus convergence, remain unchanged.

4. EXPERIMENTS

The goal of our experiments is to evaluate whether the SPL Conqueror approach is feasible for predicting performance of multi-grid solver configurations. To this end, we define the following research question:

- Q1: What is the prediction accuracy and measurement effort of the heuristics (FW, PW, HO, HS, FL)?
- Q2: What is the performance difference between the optimal configuration and the configuration predicted to perform best.

4.1 Experimental Setup and Procedure

To answer the research questions, we selected two multigrid solver implementations for different application domains, which will be described in the following sections. Although different evaluation criteria are of interest, we decided for the time to compute the solution. Note that this does not include the time required for compilation, which can easily be in the ranges of minutes for a single configuration and thus may need more time than the computation itself.

Each experiment consists of two phases. In the first phase, we measure a subset of configurations of the subject systems and determine the contribution of features, feature interactions, and parameters; the measured configurations are selected by the heuristic applied (FW, PW, HO, HS, FL). In the second phase, the performance of all possible configurations is predicted, based on the contributions measured before. To determine the prediction accuracy, we measured all configurations of the current system, an approach we call *Brute Force* (BF), as a reference. Then, the average error rate μ of each prediction can be calculated for each heuristic, as follows:

$$\mu = \frac{1}{n} \sum_{0 \leq i < n} \frac{\left| t_{i,\text{measured}} - t_{i,\text{predicted}} \right|}{t_{i,\text{measured}}},$$

where n is the number of configurations of the subject system.

Additionally, we determine the performance difference between the performance-optimal configuration and the configuration predicted to perform best.

4.2 HIPA^{cc}

HIPA^{cc} – the Heterogeneous Image Processing Acceleration Framework[1] – generates efficient low-level code from a high-level abstract domain-specific language (DSL) [14]. Coming from the domain of medical image processing, HIPA^{cc} supports several boundary conditions, such as mirroring and clamping. Image filters are described as kernels applied to the target image in a stencil notation and may have read access to multiple source images. Automatic parallelization and generation of CUDA, OpenCL, or Android

[1] `http://hipacc-lang.org`

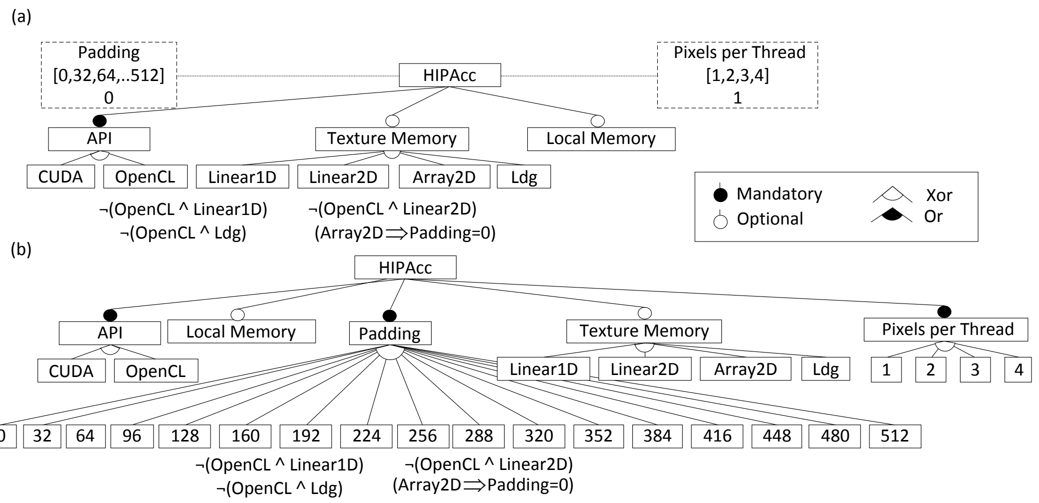


Figure 1: Feature model for HIPA^{cc} experiments, as initially (a) modeled and as modeled (b) with Xor-groups for *Padding* and *Pixels per Thread*.

FilterScript/RenderScript code is performed by a source-to-source-compiler based on Clang. During compilation, HIPA^{cc} optimizes the computational kernels by exploiting domain and hardware knowledge.

The geometric multi-grid code that we use in our experiments solves a finite difference discretization of the Poisson equation. As HIPA^{cc} is a framework for image processing, only a regular, rectangular grid can be employed. The test case uses the Jacobi method for pre- and post-smoothing steps and the standard restriction and prolongation operators. It uses a fixed number of V-cycles and smoothing steps, independent of problem size or convergence rates. For coarse grid solving, the smoother is applied again with a sufficient number of iterations. While there is no variability in the program itself, we consider various optimization switches of HIPA^{cc}. HIPA^{cc} has a built-in list of supported hardware platforms, allowing the target architecture to be selected via a switch at compile time. This switch, however, is only used for the built-in auto-tuning process and to prevent the generation of code unsuitable for the targeted device. Therefore, it has not been varied in our experiments. Since the *API* used for interaction with the device has to be specified, it was modeled as a *Xor-group*. Valid options for the targeted hardware platform are *CUDA* and *OpenCL*. Another *Xor-group* is the memory layout to be used, determined via the parameter *Texture Memory*. Additionally, *Local Memory* is an optional feature toggling the use of another memory type. The integer value *Padding*, modeled as a parameter, may be specified to optimize memory layout. For technical reasons, this parameter can only be increased in steps of 32. Furthermore, the integer value *Pixels per Thread*, denoting how many pixels are calculated per thread, may be varied. We illustrate this variability with the model in Figure 1a. Please note that this feature model is not complete and only resembles the parameters varied for this paper. A more complete description of the HIPA^{cc} parameter space can be found in [15].

Although this model can be used as base for the FL heuristic presented in Section 2.2, the model has to be adjusted for the other heuristics (FW, PW, HO, HS), as they can only hardly handle non-Boolean parameters. To circumvent this problem, we create an *Xor-group* for each of these parameters and introduce a feature for every possible value, resulting in an adapted model, as depicted in Figure 1b.

All measurements of the HIPA^{cc} system are performed on an nVidia K20 card.

4.3 Highly Scalable MG Prototype (HSMGP)

HSMGP is a prototype code for benchmarking HHG (Hierarchical Hybrid Grids) [5, 8] data structures, algorithms, and concepts [13]. It was designed to run on large scale systems such as JuQueen, a Blue Gene/Q system, located at the Jülich Supercomputing Centre, Germany. In its current form, it solves a finite differences discretization of Poisson's equation on a general block-structured domain. Concerning variability, the solver provides different smoothers, where the most relevant ones to us are *Jacobi (Jac)*, *Gauss-Seidel (GS)*, *Red-Black Gauss-Seidel (RBGS)*, and a *Block-Smoother (BS)*. For two of the smoothers, GS and RBGS, additional communication (AC) steps can be performed within the smoother iterations, resulting in a performance overhead but, in turn, also in a possibly improved convergence. Since we want to avoid modeling this as optional feature, we decided for introducing additional modified versions of the original components. Consequently, two new smoothers, *GS with additional communication (GSAC)* and *RBGS with additional communication (RBGSAC)*, are added.

The second customizable algorithmic component is the coarse grid solver. Here, a parallel Conjugate Gradient (CG) and an implementation of an Algebraic Multi-Grid (AMG) provided by the software package HYPRE [2] are available. If AMG is used, HSMGP can perform a redistribution of the coarse grid data onto a smaller number of compute nodes, before starting the solver. Again, we follow the idea presented before and end up with three choices, an *in-place CG (IP_CG)*, an *in-place AMG (IP_AMG)*, and an *AMG with data reduction (RED_AMG)*. As using multiple *smoother* types or *coarse grid solvers* is something usually not occuring in our context, we model these choices as two Xor-

[2]`http://www.llnl.gov/CASC/hypre/`

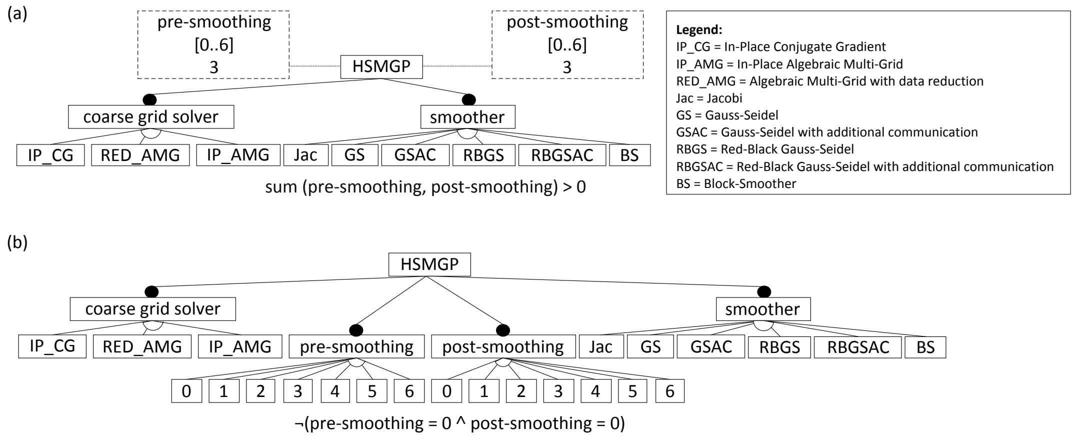

(a)
pre-smoothing
[0..6]
3
post-smoothing
[0..6]
3
HSMGP
coarse grid solver
smoother
IP_CG
RED_AMG
IP_AMG
Jac
GS
GSAC
RBGS
RBGSAC
BS
sum (pre-smoothing, post-smoothing) > 0
Legend:
IP_CG = In-Place Conjugate Gradient
IP_AMG = In-Place Algebraic Multi-Grid
RED_AMG = Algebraic Multi-Grid with data reduction
Jac = Jacobi
GS = Gauss-Seidel
GSAC = Gauss-Seidel with additional communication
RBGS = Red-Black Gauss-Seidel
RBGSAC = Red-Black Gauss-Seidel with additional communication
BS = Block-Smoother
(b)
HSMGP
coarse grid solver
pre-smoothing
post-smoothing
smoother
0 1 2 3 4 5 6
¬(pre-smoothing = 0 ^ post-smoothing = 0)

Figure 2: Feature model for HSMGP, as (a) initially modeled and as (b) modeled with Xor-groups for the number of *pre-* and *post-smoothing* steps.

groups.

Furthermore, various parameters can be tuned. In our experiments, we chose the number of *pre-* and *post-smoothing* steps, both of which we limit to integer values between 0 and 6, the default value is set to 3. Additionally, we introduce a constraint that the sum of the two parameters can not be 0, since this would disable smoothing and thus prevent solving. Overall, we end up with the model given in Figure 2a.

For the same reasons given previously, we create an Xor-group for each of the parameters and introduce a feature for every possible value. This results in the feature model depicted in Figure 2b.

Since, the configuration space is quite small, compared to a full model of HSMGP, we are able to measure all configurations (BF), which is necessary to determine accuracy of the SPL Conqueror approach.

For performing our measurements, we chose the JuQueen system at the Jülich Supercomputing Centre, Germany. Although the application scales up to the whole machine (458752 cores) [13], we decided to use only a smaller test case to ensure reproducibility and cost effectiveness. Thus, we setup HSMGP to run with 16384 threads on 4096 cores, solving for roughly $4 \cdot 10^9$ unknowns.

5. RESULTS

The Measurement results for the two subject systems are given in Table 1, we describe them in detail in the remaining section.

5.1 HIPA^{cc}

With respect to the results from our BF measurements, the performance-optimal configuration of HIPA^{cc} use *OpenCL*, a *Padding* of *256*, *4 Pixels per Thread* with *Local Memory* and *Texture Memory* options *disabled*. The time-to-solution of the configuration is 21.25 ms.

With the FW heuristic, we can predict performance for all configurations of HIPA^{cc} with a average error rate of 8.6 % (see Table 1). To achieve this prediction accuracy we perform 26 measurements (3.7 % of all configurations). The absolute time-to-solution difference between the performance-optimal configuration and the configuration predicted to perform best is 0.45 ms, resulting in an error of 2.1 %.

Using the PW heuristic the average error rate decreases to 5.7 % and requires 126 additional measurements, resulting in 152 measurements in total. The absolute performance difference between the optimal configuration and the configuration predicted to perform best decreases to 0.03 ms (0.1 % of the time needed to perform the optimal configuration).

The average error rate decreases to 1.5 % if we apply the HO heuristic. For this heuristic, we need to perform 277 measurements. The performance difference between the optimal configuration and the configuration predicted to perform best is the same as for PW.

If we consider hot-spot features by applying the HS heuristic, we have to perform 407 measurements and reach an average error rate of 1.2 %. We found that the different *Texture Memory* features, the *API* features, *Local Memory*, and different *Pixels per Thread* are hot-spot features. The absolute performance difference between the configuration predicted to perform best and the performance-optimal configuration is only 0.03 ms.

For the FL heuristic, we perform only 52 measurements. The average error rate of this heuristic is about 9.9 %. The absolute difference between the optimal configuration and the configuration predicted to be optimal is 0.02 ms. This is less than 0.1 % of the time to perform the optimal configuration.

5.2 HSMGP

The performance-optimal configuration of HSMGP uses the *IP_AMG* coarse grid solver, the *GS* smoother, one *pre-* and five *post-smoothing* steps with a time-to-solution of 1137.41 ms.

The average error rate of predictions of the FW heuristic is 51.9 % with measuring 22 configurations. The difference between the configuration predicted to perform best and the optimal configuration is 93.8 ms (8.3 % of the time to perform the optimal configuration).

With the PW heuristic, the average error rate decreases to 12.2 %, but requires to measure 192 configurations, 22 % of all valid configurations. The run-time difference between the optimal configuration and the configuration predicted to be optimal is 184 ms.

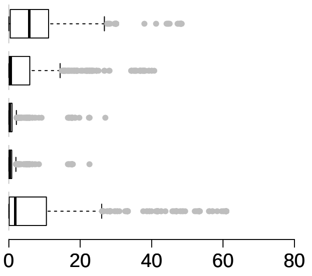
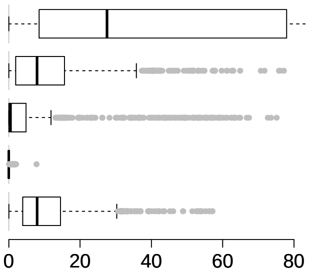
Program	Heuristic	# M	Time [ms]	Faultrate distribution	$\mu \pm \sigma$	Δ [ms]	δ [%]
HIPA^{cc}	FW	26	698.30		8.6 ± 10.2	0.45	2.1
	PW	152	4 155.11		5.7 ± 9.9	0.03	0.1
	HO	277	9 384.90		1.5 ± 3.7	0.03	0.1
	HS	407	15 491.94		1.2 ± 3.1	0.03	0.1
	FL	52	1 513.92		9.9 ± 15.7	0.02	<0.1
	BF	696	24 580.92		—	—	—
HSMGP	FW	22	51 773.21		51.9 ± 59.3	93.81	8.3
	PW	192	518 721.48		12.2 ± 14.7	184.80	16.3
	HO	636	2 037 253.25		8.5 ± 17	2474.11	217.5
	HS	864	2 230 326.94		0.2 ± 0.6	5.06	0.4
	FL	88	209 415.50		11.2 ± 10.9	695.31	61.1
	BF	864	2 230 326.94		—	—	—

Table 1: Experimental results for the two subject systems; FW: feature-wise, PW: pair-wise, HO: higher-order, HS: hot-spot, FL: function learning, BF: brute force, # M: number of measurements required for the heuristic, Time: runtime for all measurements neglecting compilation times, μ: average error rate, σ: standard deviation, Δ: absolute difference between the measured performance of the optimal configuration and the measured performance of the configuration predicted to perform best, δ: percentage share of Δ on measured performance of the optimal configuration.

For the HO heuristic, the number of required measurements increases to 636, which is 73 % of all valid configurations. With this heuristic, the average error rate drops to 8.5 %. Although the average error rate decreases, a non-performance-optimal configuration is predicted to perform best. The absolute difference between this configuration and the optimal configuration is 2474.1 ms (about 217.5 % of the time to perform the optimal configuration).

When applying the HS heuristic, we detect that all available *coarse grid solvers* are hot-spot features. When measuring the interactions of all hot-spot features, we reach an error rate of 0.2 %, but we have to measure *all* configurations for that. The performance difference between the optimal configuration and the configuration predicted to perform best is about 5.1 ms (0.4 % of performance of the optimal configuration).

For the FL heuristic, we observe an average error rate of 11.2 %. We have to measure 88 configurations, representing only 10.2 % of all configurations. With the performance predictions of this heuristic, we predict a configuration to be optimal that has a performance difference of 695.3 ms to the optimal configuration.

6. DISCUSSION

Next, we discuss results for the two subject systems with respect to the research questions Q1 and Q2.

6.1 HIPA^{cc}

Using the FW heuristic, we have to measure only 26 of 696 configurations. With this small set, we are able to predict the performance of a configuration with an accuracy of about 91 % on average. We are also able to predict a configuration to be performance optimal that is only slightly worse than the optimal configuration. But the average error rate suggests the existence of feature interactions. When applying the PW heuristic and considering first-order interactions, we identified several feature interactions, while performing 126 additional measurements. With these measurements, the average error rate decreases to 5.7 %. Additionally, the absolute performance difference between the configuration predicted to be optimal and the performance-optimal configuration is less than 1 % of performance of the optimal configuration. With the HO heuristic, the average error rate decreases to 1.5 % in average. However, the difference between the optimal configuration and the configuration predicted to perform best remains constant.

When additionally considering hot-spot features by using the HS heuristic, we are able to cover all existing interactions. However, the performance-optimal configuration was not predicted to perform best. Yet, the absolute performance difference between the optimal configuration and the configuration predicted to perform best is only 0.02 ms and thus negligible because it is less than 1 % of the performance. Although, we identify more hot-spot interactions than second-order iteration the accuracy improvement is minimal when applying the HS heuristic in contrast to HO heuristic. As a consequence, we state that there are many hot-spot interaction having only a small impact on performance.

Although the FL heuristic performs more measurements than FW heuristic, and also considers interactions between features and parameters, it has the worst prediction accuracy for HIPA^{cc}. This is because interactions between two or more features and between two or more parameters are not considered.

Overall, the FW heuristic is able to give an impression over the general performance distribution of the configurations. Moreover, an almost optimal configuration was predict to perform best. On the top, even when considering all interactions, we can not identify the optimal configuration but only a configuration that is almost optimal. Additionally, it is possible to predict performance of a random selected configuration with a high accuracy when applying the HO heuristic.

6.2 HSMGP

When considering contributions of individual features only, we have to measure 22 of 864 configurations. The average prediction error is about 51.9 %. As a consequence, there are many feature interactions having an impact on performance. However, the performance loss when using the configuration predicted to perform best is with 8.3 % relatively small.

When using the PW heuristic on HSMGP, 170 additional measurements are needed, compared to the simpler FW heuristic, and the error rate drops substantially, so we conclude that the vast majority of interactions are pair-wise interactions. Measuring 22 % of all configurations of the system to reach an accuracy of 88 %, on average, may be sufficient for some application scenarios. Yet, the accuracy for predicting the optimal configuration decreases.

With the HO heuristic, we considered also interactions between three features, which however required to measure 73.6 % of all configurations. The small improvement of 4 % compared to the PW heuristic does not justify this large number of additional measurements. Worse, a configuration with a bad performance is predicted to perform best. In examining the predictions of all configurations, we find that the configuration predicted to perform second optimal has only an performance penalty of 2.5 % compared to the real optimal configuration.

Similar to the results of HIPA^{cc}, the HS heuristic reaches the best overall prediction accuracy, but for this system, we need to measure all configurations. This strongly indicates that all features interact with each other. In general, this is the worst case for this heuristic. However, even with measuring all configuration the performance-optimal configuration was not predicted to perform best. This is because we use the measured feature contributions to predict performance of a configuration and do not simply return the measured value if the configuration was already measured by the approach.

The FL heuristic requires only 88 measurements (10.2 % of all configurations) to reach a prediction accuracy of 88.8 %, on average. Thus, it produces more accurate predictions with less measurements than the PW heuristic (which was not the case for HIPA^{cc}). However, the performance difference between the optimal configuration and the configuration predicted to perform best is much larger.

As a result, because of the high number of interactions in this system, using the HO and HS heuristics is not beneficial. Again, the FW heuristic performs best to predict the performance optimal configuration when considering the number of performed measurements. However, the PW and FL heuristic can be used to minimize the configuration space.

6.3 Threats to Validity

Internal Validity. Performance measurements are often biased by environmental factors or concurrent system load. Since we use measurements for training and evaluating the predictions' accuracies, these factors threaten internal validity. To reduce this threat, we repeated each measurement multiple times and computed the average (arithmetic mean), which we then use for our evaluation. Although this approach increases the time needed to perform a prediction, it minimizes the measurement noise.

External Validity. As we performed experiments with only two configurable stencil codes, we cannot safely transfer the results to all other stencil programs. However, we examine two systems from different domains that increases external validity slightly. Moreover, we repeated our experiments on JuQueen using a larger test case (16384 instead of 4096 cores), and observed almost identical performance-prediction accuracies.

However, our experiments are just a proof of concept, testing whether applying existing approaches to performance-optimal configuration is feasible in the stencil domain. We are aware, that further experiments and a more refined approach are needed to draw more robust conclusions.

7. RELATED WORK

Optimization and auto-tuning of configurable programs is a widely researched area. There is a number of approaches for performing configuration optimization without relying on domain knowledge. These approaches can be divided in white-box and black-box approaches.

Siegmund et al. propose an white-box approach [19], extending the approach we use for our experiments. They create a simulator of the configurable program [1] to predict the performance contributions of the configuration parameters. Since the simulator contains all variability and all executable code, they are able to predict contributions of multiple parameters within one run.

Gou et al. propose an black-box approach [9] similar to the one of Siegmund et al. They use statistical learning to predicting performance of program configurations. However, in contrast to Siegmund's approach, in which configurations are selected based on heuristics, they perform a random selection of configurations. After measuring an initial set of configurations, they determine which features are responsible for the largest performance deviation. Then, they divide the set of configurations according to the selection of these performance-critical features. When predicting a configuration, they follow the path according to the selected features. Eventually, they use the measured performance of a configuration that has a similar feature selection than the configuration to be predicted. However, this approach is not applicable to non-Boolean parameter values and, since it does not determine feature interactions, it is unclear how this approach handles the huge number of interactions that exist in stencil codes, as in our subject systems.

Datta performed auto-tuning of stencil-code computations for several multicore systems such as IBM Blue Gene/P [7]. With the help of the roofline model [21], Datta predicts performance bounds of stencil codes and the related quality of the optimizations. The optimization consists of two steps: First, he performs an optimization of the parameter ranges to minimize the configuration space. Second, he performs an iterative greedy search. As a consequence, he optimizes one parameter while the values of other parameters are fixed. The order of parameters are given as domain knowledge.

However, since it is necessary rewriting the code of the target program for some parameters, they need to re-adjust some already optimized parameters. Moreover, he does not consider the number of measurements required for an optimization.

8. CONCLUSION

Stencil codes expose a number of customization options to tune their performance. Without any domain knowledge, it is hard to determine which configuration (i. e., selection of customization options) leads to the best performance. To tackle this problem, we transfer an approach from product-line engineering by Siegmund et al. [17], which predicts performance of software configurations, to the stencil code domain. In a series of experiments, we demonstrated that the approach of Siegmund et al. can be used to predict performance of configurations of stencil codes. For the HIPA^{cc} system, we can predict the performance of all valid configurations with an accuracy of 98 %, on average, when measuring 277 out of 696 configurations. For the HSMGP system, we can predict the performance of all configurations with an accuracy of 88.8 %, when measuring 88 out of 864 configurations. However, our predictions for the optimal configuration have an inaccuracy of, at least, 8.3 %. Yet, it is possible to use our predictions to minimize the configuration space without losing configurations with a good performance.

As future work, we will examine the prediction accuracy of SPL Conqueror for larger configuration spaces. Moreover, we will extend the approach with heuristics considering feature interactions and parameter interactions.

9. ACKNOWLEDGMENTS

We thank Richard Membarth for providing the tested multi-grid implementation in HIPA^{cc} as well as advice on suitable evaluation parameters. This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing", under the contracts AP 206/7-1, TE 163/17-1 and RU 422/15-1. Sven Apel's work is also supported by the DFG under the contracts AP 206/4-1 and AP 206/6-1.

10. REFERENCES

[1] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. ASE*, pages 372–375. IEEE, 2011.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proc. SPLC*, pages 7–20. Springer, 2005.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proc. CAISE*, pages 491–503. Springer, 2005.

[5] B. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A massively parallel multigrid method for finite elements. *Computing in Science and Engineering*, 8(6):56–62, 2006.

[6] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial*. 2nd edition, 2000.

[7] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[8] B. Gmeiner, H. Köstler, M. Stürmer, and U. Rüde. Parallel multigrid on hierarchical hybrid grids: A performance study on current high performance computing clusters. *Concurrency and Computation: Practice and Experience*, 26(1):217–240, 2014.

[9] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. ASE*, pages 301–311. IEEE, 2013.

[10] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., 1st edition, 2010.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-2, CMU, SEI, 1990.

[12] H. Köstler, M. Stürmer, and T. Pohl. Performance engineering to achieve real-time high dynamic range imaging. *Journal of Real-Time Image Processing*, pages 1–13, 2013.

[13] S. Kuckuk, B. Gmeiner, H. Köstler, and U. Rüde. A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids. 2013. accepted at ParCo2013.

[14] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proc. IPDPS*, pages 569–581. IEEE, 2012.

[15] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Mastering Software Variant Explosion for GPU Accelerators. In *Proc. HeteroPar*, pages 123–132. Springer, 2012.

[16] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proc. ICSE*, pages 492–501. IEEE, 2013.

[17] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

[18] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3):491–507, 2013.

[19] N. Siegmund, A. von Rhein, and S. Apel. Family-based performance measurement. In *Proc. GPCE*, pages 95–104. ACM, 2013.

[20] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[21] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. In *HOT Chips, A Symposium on High Performance Chips*. IEEE, 2008.