A Multi-Objective Auto-Tuning Framework for Parallel Codes

Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini
Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch
Institute of Informatics, University of Innsbruck
Technikerstrasse 21a, 6020 Innsbruck, Austria
Email: {herbert,petert,juan,spellegrini,philipp,tf,hans}@dps.uibk.ac.at

***Abstract*—In this paper we introduce a multi-objective auto-tuning framework comprising compiler and runtime components. Focusing on individual code regions, our compiler uses a novel search technique to compute a set of optimal solutions, which are encoded into a multi-versioned executable. This enables the runtime system to choose specifically tuned code versions when dynamically adjusting to changing circumstances.**

We demonstrate our method by tuning loop tiling in cache-sensitive parallel programs, optimizing for both runtime and efficiency. Our static optimizer finds solutions matching or surpassing those determined by exhaustively sampling the search space on a regular grid, while using less than 4% of the computational effort on average. Additionally, we show that parallelism-aware multi-versioning approaches like our own gain a performance improvement of up to 70% over solutions tuned for only one specific number of threads.

I. Introduction

Efficient parallelization and optimization of programs for modern parallel architectures is a time-consuming and error-prone task that requires numerous iterations of code transformations, tuning and performance analysis which in many cases have to be redone for each different architecture.

Auto-tuning has been extensively studied in recent years to largely automate the process of tuning codes for (parallel) computers to realize portable performance [1]–[5]. The basic idea is to automatically find an effective set of transformations with proper parameter settings (e.g. tile sizes, loop ordering and unrolling factors) for individual code regions. However, while a carefully selected and tuned transformation sequence might be beneficial for one objective, the same may have adverse consequences on others [6]. Many successful practical auto-tuning solutions focus on specific applications [2], [3]. For more generic, compiler-based approaches, however, the prohibitively large and complex optimization problem of selecting, customizing and ordering transformations to obtain an optimal variant of a user's input code is still among the most fundamental open issues in compiler research [6]–[10].

Existing auto-tuning compilers often try to tackle this problem by (1) offline searching a subset of the parameter space [7] based on domain specific constraints and heuristics or analytical performance models [6], (2) online tuning of program parameters (e.g. tile sizes) [9], or (3) dynamic code generation and replacement [11]. One factor common to most of these methods and systems, particularly in the field of parallel computing, is that they focus exclusively on a single optimization objective such as execution time, memory behavior or resource consumption. Only little support exists for code optimizers that deal with trade-offs between multiple, conflicting goals.

In this paper we introduce a novel multi-objective auto-tuning framework for parallel codes. It consists of a compiler component featuring a multi-objective optimizer and a runtime system. The multi-objective optimizer derives a set of non-dominated solutions, each of them expressing a trade-off among the different conflicting objectives. This set is commonly known as *Pareto set* in the field of multi-objective optimization research [12]. Our search algorithm, which explores code transformations and their parameter settings, is based on Differential Evolution [13]. Additionally, Rough sets [14] are employed to reduce the search space, and thus the number of evaluations required during compilation.

To make effective use of the resulting Pareto set of optimal solutions, each of them has to be made available at runtime. This is achieved by having the compiler generate a set of code versions per region, each corresponding to one specific solution. The runtime system then exploits the trade-off among the different objectives by selecting a specific solution (code version) for each region, based on context-specific criteria.

We have implemented our techniques based on the *Insieme Compiler and Runtime* infrastructure [15]. Our approach is generic and can be applied to arbitrary transformations and parameter settings. We demonstrate our implementation by exploring the trade-off between execution time and efficiency when tuning tile-sizes and the numbers of involved threads in shared memory parallel applications.

The major contributions of this work include:

- the design of a novel auto-tuning architecture facilitating the consideration of multiple conflicting criteria simultaneously by interpreting the central task as a multi-objective optimization problem
- the combination of the search for optimal tile sizes and the ideal number of threads for a parallel code section to minimize execution time and maximize parallel efficiency into a single, multi-objective optimization problem

This research has been partially funded by the Austrian Research Promotion Agency under contract 834307 (AutoCore) and by the FWF Austrian Science Fund under contract I01079 (GEMSCLAIM).

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

- the development of a multi-objective optimization algorithm capable of solving the combined problem using a reasonable number of iterative compilation steps

In the following section initial experimental results motivate the research described in this paper. Section III describes the essential components of our framework and is followed by a brief overview of our implementation in Section IV. Section V presents experimental results that demonstrate the effectiveness of our method. In Section VI we compare our approach to related work before we conclude in Section VII.

II. Motivation

Two separate but related observations motivate the design of our multi-objective auto-tuner. Firstly, it is well-known that for many computational problems, strong parallel scaling can not be achieved. Beyond some threshold – the exact value of which is problem, architecture and implementation dependent – increasing the number of threads (cores) will no longer sufficiently decrease computation time, which results in an inefficient use of the available resources. The resulting trade-off between efficiency and speedup motivates the use of multi-objective optimization techniques in compiler research. Fig. 1 illustrates this trade-off on one of the parallel computers and benchmarks used in our evaluation (see Section V for details).

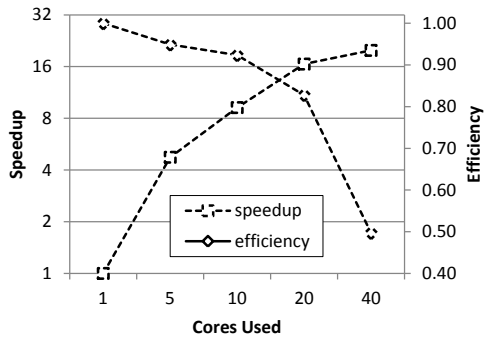


Fig. 1. Efficiency and speedup trade-off in a matrix multiplication kernel.

A second observation, which motivated the use of multi-versioning for our compilation approach, is that different numbers of threads may require specific transformation parameter values or even distinct transformation sequences for optimal performance. A potential reason for this is that the effective capacity of shared cache levels exploitable by individual threads depends on the number of concurrent threads running on the same chip. In Fig. 2 this behaviour is illustrated for different tile sizes in the case of three-dimensional tiling for matrix multiplication (IJK loop ordering). The images show the relative execution time of tile size combinations for the i and j loop, keeping the tile size for the k loop fixed. Darker areas represent faster tile size combinations. It can be seen that the selection of optimal tile sizes depends on the amount of threads used in the computation – a pattern that has also recently been observed by Shirako et al. [6].

These preliminary results demonstrate that, in order to offer the highest possible performance at any desired efficiency

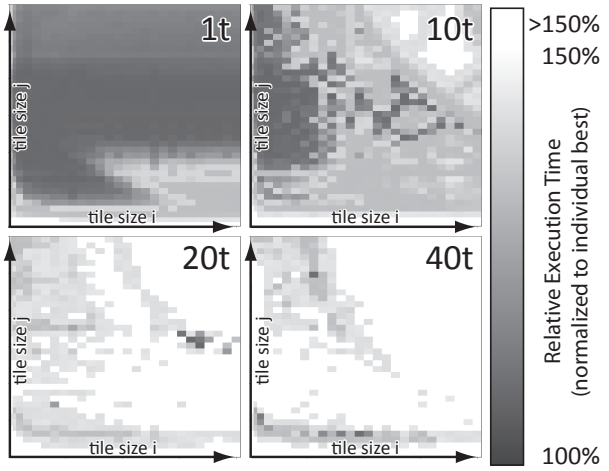


Fig. 2. Relative execution time of tiled matrix multiplication with different tile size selections, for 1, 10, 20 and 40 threads.

level – and thus thread count – an optimizing multi-objective compiler has to include some adaptive mechanism to specialize each tuned parallel code section for a given number of threads.

III. Method

Our method, which is based on a combination of compiler and runtime techniques, is described in this section. The first part gives an overview of the general architecture of our approach, whereas the second part provides details about our multi-objective optimization algorithm.

A. Architecture Overview

Fig. 3 illustrates the overall architecture of our solution, highlighting the four main components: the code analyzer, the multi-objective optimizer, the multi-versioning backend and the runtime system. Each of them can be individually customized or exchanged by alternative implementations.

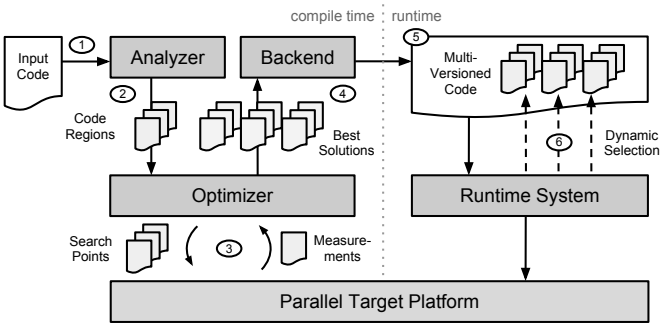


Fig. 3. Overview of our multi-objective optimization infrastructure

The labels (1-6) in Fig. 3 follow the processing of a program within our framework. An input code will be loaded by the compiler (1), analyzed and decomposed into regions to be tuned by the optimizer.

For each region, the analyzer determines a set of *transformation skeletons* which describe generic sequences of code

transformations using unbound *parameters* for tunable properties (e.g. tile sizes, unrolling factors or number of threads).

The regions, together with their associated transformation skeletons and some (optional) parameter constraints, are passed on to the optimizer (2). At this point, the optimizer conducts auto-tuning by iteratively selecting sets of *configurations* for each of the regions to be evaluated (executed) on the target system (3). Each configuration corresponds to an instantiation of a transformation skeleton's parameters. During the evaluation, a single execution of the resulting program is sufficient to obtain measurements for all simultaneously tuned regions. To further reduce the optimization time, multiple independent configurations are generated, compiled and if possible evaluated in parallel on distinct instances of the targeted platform.

In the end, the optimizer derives a Pareto set for each of the selected regions (4). The backend then generates, for each code region, a set of specialized *code versions* – one for each solution in the obtained Pareto set (5). In addition, each code version is annotated with meta-information to be used by the runtime system, including details regarding the represented trade-off.

Finally, during execution of the resulting code region, the runtime system dynamically selects among the available code versions (6). The actual strategy used to do so remains application specific. The decision might be forwarded to the user. Alternatively, system wide performance settings may be considered. In more sophisticated scenarios, dynamic or static task schedulers could be extended to exploit this additional flexibility to improve their own (potentially multi-objective) quality of service. However, the ultimate method to utilize the newly gained opportunity of dynamically customizing non-functional attributes is beyond the scope of this paper and left for future research.

B. The Static Optimizer Algorithm

The role of the static optimizer is to tune the transformation skeletons provided for the selected regions by computing a set of good configurations (solutions) for each of them.

1) Multi-Objective Optimization: A multi-objective optimization problem is characterized by an *objective function* $f : C \rightarrow \mathbb{R}^m$, where C is the set of all valid configurations and $m \geq 2$ the number of objectives. A configuration $c_1 \in C$ dominates a configuration $c_2 \in C$, if c_1 provides better results than c_2 for all objectives. On the contrary, two configurations are non-dominated if neither is dominating the other. A set of non-dominated configurations is called *Pareto set*. A Pareto set is said to be *optimal* when no configuration $c' \in C$ dominates any configuration in it. The goal of a multi-objective optimization algorithm is to find a Pareto set $S \subseteq C$ as close as possible to the optimal Pareto set.

Within the static optimizer, the problem of tuning a region is mapped to an instance of a multi-objective optimization problem and solved using a generic solver. Therefore, the search space C is derived from the parameters exhibited by the associated skeletons. Within each configuration all tuning options, including the skeleton to be selected, potential flags enabling optional parts of the transformation skeleton, unrolling factors, tile sizes and thread count specifications are modeled uniformly. Furthermore, each configuration comprises all the information required to convert a code region into a code variant which can be evaluated by executing it on the target system. This process is performed by the implementation of the objective function f, which executes the resulting version and collects measurements to quantify the individual objectives (e.g. execution time, resource usage, energy consumption, etc.).

2) Multi-Objective Optimization Techniques: Usually, the cardinality of the set C is prohibitively large, rendering it impossible to perform an exhaustive search evaluating all the configurations. In order to deal with this problem, approximation techniques are employed. The challenge is to compute solutions that are close to the optimal Pareto set by evaluating only a minimal number of configurations.

Traditionally, approximation techniques from the field of operational research like Nelder-Mead, Simplex or Genetic Algorithms (GA) [10], [11], [16] have been applied within optimizing compilers. Although these techniques only evaluate a small fraction of the overall search space, the number of steps is still too large to represent a viable option to be used within a compiler. In order to overcome this obstacle, several complementary methods for additionally reducing the search space have been proposed [6]. The effectiveness of most of these methods relies on analytical prediction models that are largely domain specific. Furthermore, these approaches only focus on a single objective. It has not yet been demonstrated whether these techniques can be extended to multi-objective optimization problems.

In this paper, we propose a novel multi-objective optimization algorithm to be used within our iterative compiler framework. We refer to this algorithm as *RS-GDE3*. It combines an approximation technique from the class of Differential Evolution (DE) algorithms [13] with a reduction mechanism based on Rough Set theory [14]. Unlike other reduction mechanisms, our selected concept does not depend on any domain-specific knowledge. To reduce the search space, the Rough Set based approach requires only a small number of evaluated configurations. Consequently, our approach does not depend on any analytical models or heuristics to reduce the search space – making it de facto independent of the actual interpretation of the tuned parameters.

Fig. 4 shows a graphical representation of our chosen approximation technique. It iteratively searches for better configurations within gradually updated search space boundaries. In each iteration, new configurations are generated using the Differential Evolution algorithm GDE3 [17]. Subsequently, the current search space boundaries are updated based on the results obtained from the newly generated configurations. The process terminates when results are no longer improving. A brief overview on GDE3 and the employed Rough Set concepts are provided within the following sections.

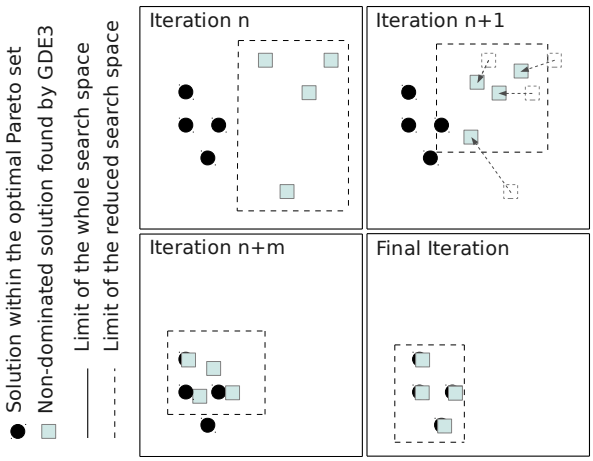


Fig. 4. Iterative progress of our combined RS-DGE3 algorithm.

Algorithm 1 Generating a new configuration in DE

```
1: Input: a, b, c, d four different configurations, B the current boundary
2: Output: r
3: index =floor(rand() · |a|) + 1
4: for i = 1 .. |a| do
5:     if (rand() < CR ∨ i = index) then
6:         r(i) = b(i) + F · (c(i) − d(i))
7:     else
8:         r(i) = a(i)
9:     end if
10: end for
11: return B.getClosestTo(r⃗)
```

3) Differential Evolution for Computing the Pareto Set: The most common approximation techniques in multi-objective optimization belong to the field of Evolutionary Computation [12]. We selected GDE3 (Generalized Differential Evolution [17]) due to its acceptable robustness and fast convergence rate towards the optimal Pareto set demonstrated for other application domains [18]. GDE3 is a search based method that optimizes a problem by iteratively trying to improve a set of candidate solutions (also called population).

Within our algorithm, the input population is the result of the previous iteration or an initial random sample of the search space C. For every configuration a in the population, three other configurations b, c, and d are selected to compute a new configuration r according to Algorithm 1. If r dominates a, then it replaces a in the population. If r is dominated by a, then it is discarded. Otherwise, r replaces a under circumstances exceeding this summary. Details can be found within [17].

The parameters CR and F within Algorithm 1 guide the generation of new configurations. We use $CR = 0.5$ and $F = 0.5$, which were used as default values in previous work [17], [18]. Further, [19] evaluated the impact of the population size. Based on this reference and some experiment runs we chose to use a small population size of 30 configurations for all our experiments.

In every iteration, GDE3 produces a set of new configurations to be evaluated on a target architecture – one for each element within the current population. We have configured GDE3 for our work to stop iterating when the solutions do not improve for three consecutive iterations. A particular advantage of GDE3 is that configurations can be evaluated simultaneously to reduce the search time, a property exploited by our auto-tuner which evaluates configurations in parallel.

4) Search Space Reduction Using Rough Sets: In every iteration of our algorithm, GDE3 searches for better configurations within a reduced search space (B within Algorithm 1). This search space is computed using Rough Sets. Rough Set theory is a mathematical approach to model imperfect knowledge. In our case, we want to model which area of the search space contains non-dominated solutions and which area can be ignored. We follow the approach described in [20] for building the reduced search space, which is illustrated in Fig. 5.

The illustrated example involves two parameters, p_1 and p_2, being optimized for two (conflicting) objectives. For building the reduced search space our algorithm explores the configurations contained in the most recently obtained population of the GDE3 phase. That population contains non-dominated and dominated solutions (squares and triangles in Fig. 5). The goal is to determine a region enclosing all non-dominated solutions (the squares). For computing the desired boundaries of that region, our Rough Set mechanism uses the coordinates of the dominated configurations surrounding the non-dominated solutions in the population. The larstest hyper-rectangle limited by those points which is enclosing all non-dominated solutions will be the new reduced search space for the next GDE3 iteration.

A drawback of this approach is that the reduced search space may not contain all the solutions within the desired optimal Pareto set. To mitigate this issue, we continuously update the reduced search space, as shown in Fig. 4, by considering new generations of non-dominated solutions to gradually steer the search towards the area where the optimal Pareto set is located.

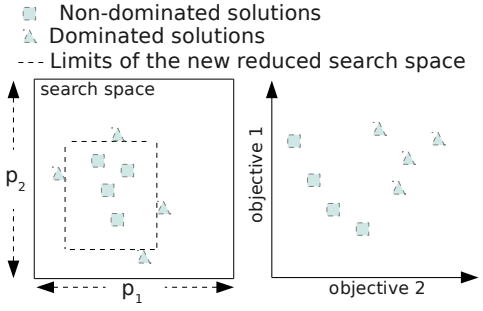


Fig. 5. Search space reduction based on Rough Set theory.

IV. Implementation

Our implementation of the multi-objective auto-tuning framework described in this paper is based on the *Insieme Compiler and Runtime* infrastructure [15] which will be briefly introduced in the following.

The *Insieme* project aims to provide an easy to use, powerful framework for source-to-source transformations and program analysis for heterogeneous multi-core parallel computers. It

consists primarily of two components: the *Insieme Compiler* and the *Insieme Runtime System*.

The *Insieme Compiler* provides a foundation for source-to-source program parallelization and optimization research. Its architecture is designed to support the processing of hybrid input codes that can include C, C++, OpenMP, MPI and OpenCL. Insieme is built on the uniform, high-level *Insieme Parallel Intermediate Representation* (INSPIRE), which aligns similar concepts of different input languages by employing identical constructs for their encoding. The design of INSPIRE focuses on a clean and compact representation, eliminating semantically overloaded or redundant constructs and providing a formal language specification. Furthermore, its implementation is designed to inherently support parallel analysis and program manipulation, enabling the parallel exploration of transformation and parameter spaces in this work.

Built around INSPIRE, Insieme offers a growing collection of analysis and transformation utilities including support for polyhedral loop transformations [21] and for automatic evaluation of static and dynamic program features to be used in program analysis and optimization. Insieme supports exchangeable backends generating C, OpenCL or MPI code. The native backend, however, produces code customized for the *Insieme Runtime System*.

The *Insieme Runtime System* complements the Insieme compiler by offering an extensible framework for parallel runtime research. Its fundamental application model enables low-overhead generic task processing with access to a rich set of metadata annotating the currently executing program. This data comprises fine-grained communication requirements, as well as extendable, compiler-deduced features characterizing the non-functional behavior of code regions. Customizable runtime system components such as schedulers, resource managers and performance tuners can rely on meta-information as well as real-time system monitoring results for their decision-making processes.

Insieme has been extended to implement the multi-objective auto-tuning techniques described in this paper. A compiler driver has been developed to implement the workflow shown in Fig. 3. The Analyzer searches for nested loops and performs a dependency test (based on the polyhedral model) to determine the largest subset of loops which can be tiled and optionally collapsed, without sacrificing the possiblity of parallelizing the resulting loop. The collapsing step is essential to mitigate load balancing issues potentially introduced by tiling with large tile sizes.

For the optimizer, a generic interface has been defined, including an abstract method to evaluate sets of configurations. Our implementation of the evaluation mechanism exploits the availability of multiple cores of the underlying computer to generate, compile and execute code versions in parallel. Since our optimization algorithm iteratively derives lists of configurations to be tested, parallel evaluation can be exploited effectively, thereby reducing the overall compilation time.

In the last step of the workflow, the backend needs to generate code including multiple versions of the tuned code regions. It does so by first outlining the selected regions into functions. Based on the configurations obtained by the optimizer, variations of the outlined functions are generated. Finally, function pointers enriched with meta-information comprising specific properties of the individual versions are aggregated within a table that is statically generated and embedded within the resulting multi-versioned code (label (5) in Fig. 3 and Fig. 6). This table includes information about the trade-off represented by each code version, which can be exploited by the runtime decision making process that selects a code version to be executed in order to achieve a specific optimization goal.

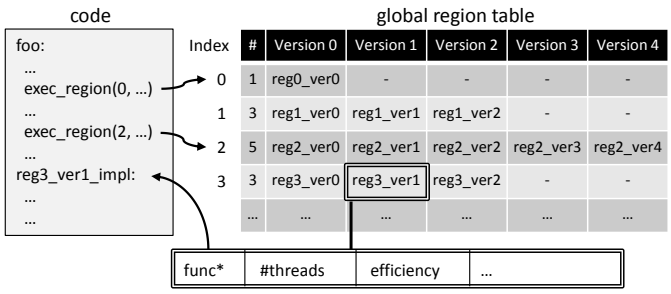


Fig. 6. Multi-versioning implementation in generated code.

It should be noted that for some transformations, it would also be possible to generate a single, parameterized version of the code instead of performing multi-versioning (see e.g. [9]). However, this approach is not general, as there are some transformations such as loop unrolling, fission and fusion which can not be realized using parameterized code. Moreover, multi-versioning with fixed parameters is likely to allow the binary compiler following our source-to-source conversion to generate more efficient code than for an equivalent, parametrized code version.

Finally, we added a module to the *Insieme Runtime System* for the selection of a particular code version when processing multi-versioned code regions. We delegate the invocation of each outlined region function to the runtime system. The runtime then selects an adequate version from the global table. The actual policy for selecting code versions is dynamically configurable. For instance, a user may supply weights w_c for each component c of the objective function f. The runtime system then uses the available meta-data to select the version v from the Pareto set S which minimizes $\sum_c w_c f_c(v)$.

V. Experimental Results

A. Experimental Setup

For our experiments we used two parallel computing systems. The first, an Intel-based system, incorporates 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. We refer to this system as *Westmere* throughout the paper. The second system is an AMD server, named *Barcelona*, which is based on 8 Opteron 8356 processors, each contributing 4 cores. Table I summarizes the configuration of these systems. Note that L1 and L2 are per-core private caches, whereas L3 is shared among the cores of each CPU.

TABLE I
HARDWARE PLATFORMS FOR EXPERIMENTAL EVALUATION.

System	Sockets/ Cores	Cache L1d/i	Cache L2	Cache L3	Setup Kernel	Setup GCC
Westmere	4/40	32K/32K	256K	30M	2.6.32	4.5.3
Barcelona	8/32	64K/64K	512K	2M	2.6.18	4.5.3

```
1: for i = 1 ... N do
2:     for j = 1 ... N do
3:         for k = 1 ... N do
4:             C[i][j] = C[i][j] + A[i][k] * B[k][j]
5:         end for
6:     end for
7: end for
```

Fig. 7. Simple MM kernel using IJK loop ordering.

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. For our experiments we observed that the hyper-threading support on the Intel architecture did not provide any extra benefit – neither in terms of speed nor efficiency. Therefore, we are skipping the corresponding results for brevity.

B. A Detailed Example

To provide a detailed analysis of our approach we have selected the widely known and investigated *Matrix Multiplication* (mm) kernel as shown within Fig. 7. We are applying loop tiling on all three loop levels, thereby creating a three-dimensional parameter space. This transformation is followed by collapsing the two outermost tiling loops (corresponding to the i and j loops) to increase the number of iterations before parallelizing the resulting outermost loop.

Unlike in the sequential or the non-tiled case, using the (initially) more cache friendly IKJ loop ordering as a starting point is not an option due to a inherent load balancing issue introduced by doing so. When tiling the IKJ variant of an mm-kernel using (necessarily) large tile sizes $t = (t_i, t_k, t_j)$ to obtain a good last level cache utilization, the number of loop iterations is limited by $\lceil N/t_i \rceil$ due to tiling. The larger t_i the less iterations can be distributed among the threads processing the loop. Additionally, each iteration computes larger sections of the resulting matrix. As a consequence load balancing can degrade, which may seriously limit the scalability for an increasing number of cores. To mitigate this effect, loop collapsing can be applied before parallelizing the outermost loop, thereby effectively reducing the workload associated with an individual iteration. However, in the IKJ case collapsing the tiling-loops of i and k prohibits the intended subsequent parallelization. Since this limitation does not occur when starting with the IJK loop ordering, we have selected the IJK loop ordering for our evaluation.

1) Tiling of Parallel Loops: We first want to investigate the influence of the number of threads used for processing a parallel loop nest on the optimal tile size for this loop. Therefore, we conducted an extensive search within a necessarily restricted search space on two different architectures. For a problem size of $N = 1400$, mm-kernel variations using more than 14.000 tiling configurations $t = (t_i, t_j, t_k) \in [1..700]^3$ have been generated, compiled and evaluated on our target platforms for different thread numbers. On Westmere we performed our evaluation using $1, 5, 10, 20$ and 40 cores while on Barcelona configurations involving $1, 2, 4, 8, 16$ and 32 cores have been investigated. In the remainder of this paper, we refer to this kind of extensive search as the *brute force* method. A subset of the results has been illustrated in Fig. 2.

Each of the resulting configurations has been evaluated multiple times and the median of the collected execution times was used for comparison. Table II lists the best tile sizes found for each specific number of employed threads on our target platforms. For instance, the best configuration for 10 cores found on Westmere is $(t_i, t_j, t_k) = (32, 288, 9)$. The table also includes the relative performance loss when running an optimal configuration obtained for one number of threads using a different number of cores. For instance, using 40 threads to run the mm kernel version tuned for 10 threads takes 11% longer than the variant tuned specifically for 40 threads.

As can be observed in Table II, on both architectures the performance impact of using tiling parameters tuned for a non-matching number of threads can be quite severe. For instance, if we apply the best tile size configuration found for a single thread to a matrix multiply execution with all available cores on our target architectures, then the resulting performance degradation amounts to 15.1% and 18%, respectively, compared to the best found tile size for the largest number of available threads. On our Barcelona system, this performance impact even reaches 30.1% when using the configuration tuned for 8 cores for a mm run involving 32 cores. Also, when using configurations obtained for a large number of cores in scenarios involving fewer threads, a slowdown can be observed. Using the best 32-core configuration on the Barcelona system with less threads results in performance loss between 5.3% and 22% compared to the individual best configurations.

Configurations representing the optimum for a single number of cores are on average between 1.8% and 8.3% (column Avg. in Table II) slower than individually tuned solutions on the Westmere architecture. On the Barcelona architecture this range varies between 5.4% and 13.7%. Finally, the comparison with the GCC -O3 baseline demonstrates the well known, enormous potential of tiling in general as well as the high quality of the individually tuned solutions.

2) Trade-off between Speedup and Efficiency: In addition to the impact on optimal tiling parameters, the collected data also enables us to investigate the trade-off between speedup and efficiency when tuning the mm-kernel for multiple objectives.

Let t_s be the execution time of the fastest (tiled) sequential code version and $t_p(x)$ its parallel counterpart using x threads. The speedup of a code being executed using x threads is usually defined by $s(x) = \frac{t_s}{t_p(x)}$ and its efficiency by $e(x) =$

TABLE II
OPTIMAL TILING PARAMETERS FOR DIFFERENT NUMBER OF THREADS AND ARCHITECTURES

Westmere Architecture

	opt. Tile Sizes			Perf. Loss over Best for # of Cores in %					
Nr. of Cores	t_i	t_j	t_k	1	5	10	20	40	Avg.
1 core	96	128	8	-	1.2	5.4	11.3	15.1	8.3
5 cores	32	304	9	1.0	-	1.5	4.5	0.3	1.8
10 cores	32	288	9	0.9	0.1	-	4.3	11.0	4.1
20 cores	32	192	12	2.3	2.2	2.5	-	10.4	4.4
40 cores	32	208	12	1.7	0.4	3.2	6.8	-	3.0
GCC -O3	-	-	-	605.3	593.3	559.7	502.6	422.8	536.8

Barcelona Architecture

	opt. Tile Sizes			Perf. Loss over Best for # of Cores in %						
Nr. of Cores	t_i	t_j	t_k	1	2	4	8	16	32	Avg.
1 core	96	480	5	-	7.3	7.3	10.7	6.0	18.0	9.9
2 cores	80	496	5	0.6	-	4.5	5.1	14.9	17.1	8.4
4 cores	128	352	7	1.4	4.3	-	12.1	4.2	19.3	8.3
8 cores	176	304	7	6.7	3.1	5.5	-	23.2	30.1	13.7
16 cores	176	352	8	7.4	6.6	8.0	2.1	-	2.9	5.4
32 cores	144	240	8	5.9	5.3	9.8	22.0	13.5	-	11.3
GCC -O3	-	-	-	3948.7	3732.9	3619.9	3886.5	3486.5	4628.0	3883.8

$\frac{s(x)}{x} = \frac{t_s}{x \cdot t_p(x)}$. Unfortunately, both definitions depend on the fastest sequential execution time, which may not always be known. However, t_s is a constant which can be omitted when comparing the quality of different versions of the same region. Hence, instead of comparing the speedup $s(x)$ we can use the execution time $t_p(x)$ when trying to derive the fastest solution - just as the total *resource usage* $r(x) = x \cdot t_p(x)$ can be used as an equivalent, yet obtainable substitute for $e(x)$ as part of our optimizer. However, in general we strive to maximize $e(x)$ whereas $r(x)$ needs to be minimized.

When plotting execution time vs. the resource usage of all the configurations evaluated using the brute force search mechanism, the pattern illustrated within Fig. 8 is produced. Due to the correlation between the execution time $t_p(x)$ and the resource usage $r(x) = x \cdot t_p(x)$, all points using the same number of threads are located on a line. Furthermore, due to the large number of points evaluated, the individual lines are densely populated within a certain range. In every line, there is a single point exhibiting the shortest execution time and thus also the least resource requirements. This point dominates all other solutions within the same line and corresponds to the best solutions listed in Table II.

The globally non-dominated tips of the lines are the desired configurations to be obtained using our static multi-objective optimizer. Together, they form the Pareto front of the multi-objective optimization problem for speedup and efficiency. Unfortunately, finding these optimal solutions requires searching the optimal tiling parameters for individual thread counts in multiple vast, three-dimensional search spaces. For our evaluation we only considered 5 respectively 6 out of 40 respectively 32 options for the number of cores – all of them contributing one point to the Pareto front, which might not always be the case. Each additional number of cores adds another line which may add one extra point on the Pareto front. This problem definition automatically eliminates configurations using too many cores for non-scaling codes.

TABLE III
IMPACT OF NUMBER OF THREADS ON SPEEDUP AND EFFICIENCY

Westmere Architecture

Cores	Speedup	Efficiency	Relative Time	Relative Resources
1	1.00000	1.00000	100%	100%
5	4.82873	0.96575	21%	104%
10	9.26091	0.92609	11%	108%
20	16.77778	0.83889	6%	119%
40	26.35799	0.65895	4%	152%

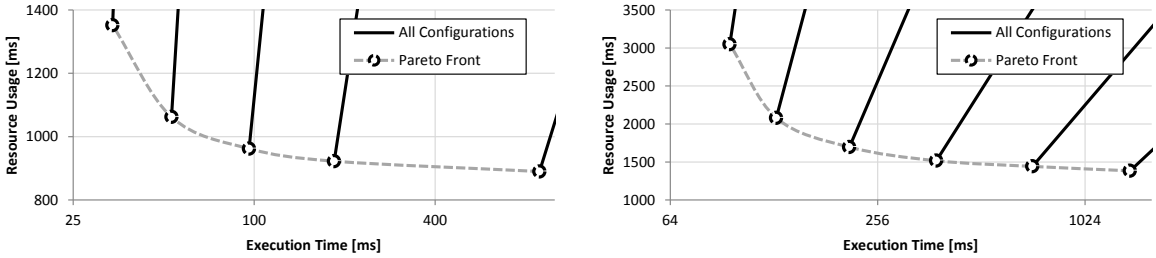
Barcelona Architecture

Cores	Speedup	Efficiency	Relative Time	Relative Resources
1	1.00000	1.00000	100%	100%
2	1.92067	0.96033	52%	104%
4	3.65286	0.91322	27%	110%
8	6.53208	0.81651	15%	123%
16	10.65231	0.66577	9%	150%
32	14.53095	0.45409	7%	220%

For these cases, the execution time will increase for larger number of cores and thus their corresponding configurations will not be part of the Pareto front. The same happens to solutions using an inadequate number of cores. For instance, if a region is only fully utilizing the available threads if their number is a power-of-two, all other thread numbers will result in solutions dominated by a configuration using fewer cores due to their bad resource utilization. Those will also be automatically discarded.

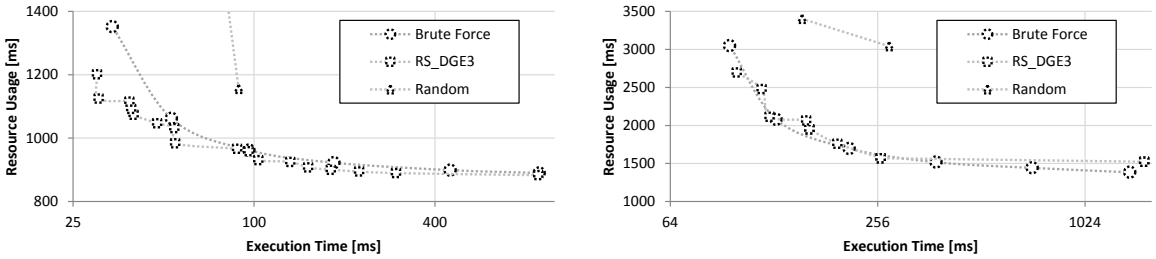
Table III lists the properties of the optimal points within the Pareto front and provides details about the trade-off between speedup and efficiency for each of these points.

3) The Optimizers' Solutions: In a final step, we processed the mm-kernel using our static optimizer as well as a random search strategy for comparison. Unlike within the brute force case, where artificial restrictions had to be added to facilitate its completion within a reasonable timeframe, only few search-space restrictions need to be defined for our optimizer. The upper boundary for tile sizes has been set to $N/2$, since larger



(a) Westmere Architecture (b) Barcelona Architecture

Fig. 8. Execution time and resource usage for different configurations evaluated based on brute force



(a) Westmere Architecture (b) Barcelona Architecture

Fig. 9. Pareto fronts computed using different optimization algorithms

tile sizes clearly have little potential to dominate smaller tile sizes. Further, the upper boundary for the number of threads was set according to the target machine. Both restrictions could easily be extracted statically from the targeted region and platform. No constrains regarding the granularity of the potential configurations have been defined.

Fig. 9 compares the Pareto front obtained by our optimizer with the ones obtained after exploring the search space with the brute force mechanism and a random search in both architectures. The implemented random search generates random configurations, evaluates them and returns those which are non-dominated. In the case of the Westmere architecture, we observe that the configurations generated by our optimizer are better (lower execution time and better utilization of resources) than the solutions generated by the brute force approach. For this specific architecture, our algorithm produces solutions that are up to 13% faster. In the case of the Barcelona architecture, our solutions are close to the brute force results. In both architectures, random search using an equal number of evaluations as our method is very far off the quality achieved by the other techniques.

To systematically compare different optimization strategies we are using three derived metrics. Let S be the set of solutions obtained by an algorithm. The first, simplest metric is $|S|$, the number of points within the set. Since a larger number of solutions offers a higher flexibility for the dynamic decision-making, this quantitative metric has been included. Additionally, to judge the quality of a solution set S, its *hypervolume* $V(S) \in [0 \dots 1]$ [22] is employed as a metric. It computes the normalized volume (in the bi-objective case the area) behind a front. The larger $V(S)$, the closer the front could be pushed toward the hypothetical ideal $(0, 0)$ point. Hence, the hypervolume provides a metric for the quality of the individually obtained solutions, ranging from 0 (worst solutions) to 1 (unattainable optimal solutions). Finally, we compare the number of points E evaluated for obtaining a solution set, representing a metric for the time required to apply the corresponding method. Unlike the other two metrics, E does not describe the quality of the obtained solution but provides an indicator of the efficiency of the algorithm itself.

Since two out of three of the covered search strategies are stochastic algorithms, they produce different results in different runs. Hence, the results of a single run are not sufficient for an objective comparison. Thus, results collected from repeated executions have to be aggregated and compared. In our evaluation we use the arithmetic means $\overline{E}$, $\overline{|S|}$ and $\overline{V(S)}$ derived by running the optimizer 5 times as a directly comparable substitute.

The results of this comparison for the mm-kernel are listed in Table VI together with the results for the kernels investigated in the following subsection. The metrics reflect the observation made in Fig. 9. For the Westmere architecture, our algorithm obtains 9.4 solutions on average, which are exceeding the quality of the solutions obtained using the brute force approach although evaluating fewer than 1.1% of the points within the search space. For the Barcelona system, 10.4 solutions with a slightly weaker performance are obtained on average by only evaluating less than 0.9% of the points the brute force approach touches. In both cases, our RS-DGE3 algorithm is vastly outperforming the much simpler random

TABLE IV
KERNEL CHARACTERISTIS

Kernel	Problem Size	Computation	Memory
mm	1400^2	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
dsyrk	1400^2	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
jacobi-2d	10000^2	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
3d-stencil	600^3	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$
n-body	500000	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$

search strategy covering a comparable number of points.

C. Additional Kernels

Finally, we would like to demonstrate the general applicability of our optimization scheme. To this end, four additional kernel codes, each exhibiting different computation and memory usage characteristics, have been selected and evaluated.

The set consists of one additional BLAS-3 linear algebra kernel (*dsyrk*, computing $B = A * A^T + B$), two stencil codes (*jacobi-2d* and a generic 3x3x3 *3d-stencil*) and a naive implementation of an *n-body* simulation. Except for the mm and $dsyrk$ kernels, all of them exhibit distinct computation / memory complexities as listed in Table IV and hence considerably different memory reuse and access patterns. Also, although identically categorized in terms of complexity, the memory access patterns of mm and $dsyrk$ are very different since the (on-the-fly) transposition of A eliminates the unaligned matrix access conducted within the mm kernel.

Table V summarizes the impact of thread-specific optimization when applying brute force on the selected kernels. Each row corresponds to the last column included in Table II, representing the average performance loss when applying the ideal tile size for some particular number of threads across all other thread counts. Additionally, the overall average loss (*avg*) and the maximum loss incurred when only optimizing for serial execution (*1tmax*) are included. Unfortunately, due to space limitations, a more detailed representation of all obtained results has to be omitted.

In the *jacobi-2d* kernel, the average performance loss across all thread numbers is 11.8% on Westmere, while it goes up to 28.7% on Barcelona. Conversely, *3d-stencil* averages 24.6% on Westmere and only 14.7% on Barcelona, demonstrating that the impact of choosing distinct per-thread-count tile parameters varies across both hardware platforms and software applications. This is even more pronounced in the *n-body* kernel, where our chosen problem size fits entirely in the cache on Westmere, resulting in almost no variation, while the performance loss is extremely significant on Barcelona, averaging 70.7% due to its limited 2 MB L3 cache. We included the *1tmax* column to point out the potentially large losses incurred when optimizing for serial performance and using the same results for parallel execution. Particularly on the Barcelona system, execution times can increase by up to a factor of 4 (293.0% loss) with this approach, and losses are above 89.2% for *jacobi-2d*, *3d-stencil* and *n-body*.

Finally, we applied our optimizer to the range of investigated kernels and compared the results with the ones obtained by the brute force evaluation and a random search. The results of this comparison are listed within Table VI.

TABLE V
BRUTE FORCE RESULT SUMMARY

Westmere Architecture

	% Avg. Perf. Loss of Best Params for						
Kernel	1t	5t	10t	20t	40t	avg	1tmax
mm	8.3	1.8	4.1	4.4	3.0	4.3	15.1
dsyrk	2.9	6.4	6.5	2.5	2.4	4.1	6.5
jacobi-2d	14.0	9.5	7.6	14.2	13.7	11.8	23.6
3d-stencil	8.6	17.1	75.1	6.6	15.8	24.6	26.7
n-body	0.7	0.3	0.5	0.4	0.5	0.5	1.5

Barcelona Architecture

	% Avg. Perf. Loss of Best Params for							
Kernel	1t	2t	4t	8t	16t	32t	avg	1tmax
mm	9.9	8.4	8.3	13.7	5.4	11.3	9.5	17.9
dsyrk	7.6	9.8	4.6	6.1	2.1	3.3	6.7	14.2
jacobi-2d	38.6	28.9	17.7	17.2	50.3	19.5	28.7	89.2
3d-stencil	70.8	5.9	2.4	3.8	2.4	3.0	14.7	119.4
n-body	112.1	116.2	110.4	29.7	28.4	27.6	70.7	293.0

Several conclusions can be extracted from these results. First: our optimizer always computed more configurations than both the brute force algorithm and the random search. Second: the number of configurations evaluated by our optimizer were between 99% and 90% lower than the evaluations required by brute force. Third: the hypervolumes of the Pareto sets computed by our technique are comparable to the volumes of the sets obtained by brute force. In the case of the Westmere architecture, sets exceeding the quality of the brute force solutions could be obtained for all the analyzed kernels. In the case of the Barcelona architecture our approach computed Pareto sets of higher quality for the *jacobi-2d* problem. Only the *n-body* problem on Westmere posed some difficulties to our technique. These difficulties could be related to the shape of the search space of that particular application. Analyzing and resolving this issue in detail will be the focal point of future work. For the sake of completeness, it should be noted that the results obtained by the optimizer clearly outperform the results of the random search in all cases.

VI. RELATED WORK

A large number of auto-tuning solutions have been proposed in the past. The solutions range from self-tuning libraries like ATLAS [2], OSKI [3], SPIRAL [23] or FFTW [24], over application specific solutions including Active Harmony [25], Sequoia [26], PetaBricks [27], [28], and frameworks for tuning parallel stencil computations [29], [30] to compiler-based concepts like our proposal. Unlike self-tuning libraries and application specific approaches, compiler-based solutions do not depend on the programmer to establish the search space e.g. by adding parameters to the input program or using tuning libraries. For instance, the LeTSeE [31] proposal automatically obtains code variations by manipulating the affine loop scheduling of loop nests while the framework presented in [7] tunes parametrized CHiLL [32] recipes similar to our skeletons. Recently, compiler-based techniques have even been extended to dynamic code generation [11], [33].

TABLE VI
PARETO FRONTS OBTAINED USING DIFFERENT OPTIMIZATION ALGORITHMS

Westmere Architecture

	Brute Force			Random			RS-DGE3		
Benchmark	E	$\|S\|$	$V(S)$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$
mm	71290	5	0.84	780	2.0	0.03	724	9.4	0.88
dsyrk	71290	5	0.75	1200	4.4	0.00	1186	11.2	0.81
jacobi-2d	23805	4	0.69	825	10.8	0.73	1027	21.2	0.83
3d-stencil	10580	5	0.77	1000	8.0	0.52	852	21.4	0.86
n-body	26136	5	0.86	1350	3.2	0.65	1334	28.6	0.95

Barcelona Architecture

	Brute Force			Random			RS-DGE3		
Benchmark	E	$\|S\|$	$V(S)$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$
mm	85548	6	0.83	800	2.0	0.01	740	10.4	0.76
dsyrk	85548	6	0.87	1200	3.6	0.00	1152	11.0	0.78
jacobi-2d	28566	6	0.73	675	10.8	0.81	674	17.2	0.88
3d-stencil	12696	6	0.87	850	9.4	0.78	822	17.0	0.85
n-body	21780	6	0.78	1100	3.6	0.51	1063.6	25.6	0.68

Also, projects like PERI [34] establish interfaces for the various tools involved in the auto-tuning process to enable future interoperability.

The main difference between our work and all the compiler-based solutions mentioned above is our focus on multi-objective optimization. While underlying components, including the involved transformations and the restriction of the search space using transformation skeletons have been presented before, none of the related works optimized for multiple objectives simultaneously.

To the best of our knowledge, only little work on multi-objective optimization in the proximity of compiler research has been published. Existing approaches focus on selecting subsets of transformation passes to balance the trade-offs between compilation time, execution time and code size on a global level. Hoste et al. [35] proposes to apply evolutionary algorithms to automatically determine the subsets of passes to be included within predefined compiler optimization levels (e.g. -O2). Fursin et al. [5] evaluated multi-objective trade-offs between all three objectives. However, the proposed machine-learning based optimization framework yields a single configuration instead of a full Pareto set. Lokuciejewski et al. [36] tunes compiler flags for embedded applications for execution time and code size. A major drawback of these approaches is that a single set of selected compilation flags are applied to an entire translation unit while our framework allows for addressing individual code regions. Heydemann et al. [37] tuned unrolling factors to balance between performance and code size. Since code size is one of the objectives, multi-versioning could not be applied and a single trade-off had to be selected statically. None of the multi-objective work focuses on parallel applications, nor on optimizing performance and efficiency at the same time.

Besides the infrastructure required to realize auto-tuning, the actual optimization technique used for selecting transformation parameters is also a widely studied topic. In the literature the problem has been tackled using two different approaches: machine learning and search based techniques. Machine learning techniques have been applied with success by Agakov et al. [38] or Rahman et al. [39]. Although these techniques allow for fast compilation times, the requirement of obtaining large training data sets can be prohibitive, especially when considering combinations of tunable transformations. Search based techniques usually require more time during compilation. However, without prior knowledge, they are capable of dealing with extremely large search spaces while only exploring small fractions, as in [7], [10].

Finally, code specialization, as required by the last step within our framework, is generally achieved using one out of two methods: parameterization and multi-versioning. Parameterized tiling has been described in [9], [40]–[42], but not in the context of multi-objective optimization. Multi-versioning for more adaptable code is performed by Mars et al. [43], but only in the serial case and limited to GCC's existing flags. Chen et al. [44] use machine learning to find optimal thread counts, but do not perform specific optimization for each version.

VII. CONCLUSION

Within this paper we presented our proposal for a compiler infrastructure capable of optimizing programs according to multiple, potentially conflicting objectives simultaneously. By exposing a set of individually tuned versions of a single code region to the runtime system, trade-off decisions can be deferred until the actual execution. This way, our system enables application developers to tune their codes automatically for multiple objectives without the requirement of fixing any priorities. This decision is left to the end user.

We demonstrated the capabilities of our framework, and in particular of our optimization algorithm, by tuning tiling parameters and thread-counts for five different parallel codes across two hardware platforms. Our search algorithm generally achieves results on par with or better than a brute force search while using 90% to 99% fewer evaluations. Additionally, our experiments provide evidence for the importance of considering the number of involved threads when tuning tile sizes. Failing to do so can decrease performance by up to a factor of 4.

REFERENCES

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

[2] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–27.

[3] R. Vuduc, J. Demmel, and K. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16. IOP Publishing, 2005, p. 521.

[4] K. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2001.

[5] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[6] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *ETAPS International Conference on Compiler Construction (CC'12)*. Tallinn, Estonia: Springer Verlag, Mar. 2012.

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

[8] L. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative optimization in the polyhedral model: Part i, one-dimensional time," in *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE, 2007, pp. 144–156.

[9] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, "Parameterized tiling revisited," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 200–209.

[10] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 237–.

[11] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 2011, pp. 879–892.

[12] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[13] R. Storn and K. Price, "Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces." *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[14] Z. Pawlak, "Rough sets," *International Journal of Parallel Programming*, vol. 11, no. 5, pp. 341–356, 1982.

[15] "Insieme comiler and runtime infrastructure." Distributed and Parallel Systems Group, University of Innsbruck. [Online]. Available: http://insieme-compiler.org

[16] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, Jan. 1965.

[17] S. Kukkonen and J. Lampinen, "Gde3: the third evolution step of generalized differential evolution," in *IEEE Congress on Evolutionary Computation*. IEEE, 2005, pp. 443–450.

[18] J. J. Durillo, A. J. Nebro, F. Luna, C. A. Coello Coello, and E. Alba, "Convergence speed in multi-objective metaheuristics: Efficiency criteria and empirical study," *International Journal for Numerical Methods in Engineering*, vol. 84, no. 11, pp. 1344 – 1375, December 2010.

[19] C. A. C. Coello and G. T. Pulido, "A micro-genetic algorithm for multiobjective optimization," *Optimization*, vol. 7, no. 5, pp. 126–140, 2001. [Online]. Available: citeseer.ist.psu.edu/444668.html

[20] L. V. Santana-Quintero, A. G. Hernández-Díaz, J. M. Luque, C. A. C. Coello, and R. Caballero, "Demors: A hybrid multi-objective optimization algorithm using differential evolution and rough set theory for constrained problems," *Computers & OR*, vol. 37, no. 3, pp. 470–480, 2010.

[21] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[22] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[23] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[24] M. Frigo, "A fast fourier transform compiler," in *Acm Sigplan Notices*, vol. 34, no. 5. ACM, 1999, pp. 169–180.

[25] C. Tapus, I. Chung, and J. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 44–44.

[26] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally *et al.*, "Sequoia: programming the memory hierarchy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 4–4.

[27] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[28] S. P. Amarasinghe, "Petabricks: a language and compiler based on autotuning," in *HiPEAC*, M. Katevenis, M. Martonosi, C. Kozyrakis, and O. Temam, Eds. ACM, 2011, p. 3.

[29] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the 2011 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, 2011, pp. 676–687.

[30] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations." in *IPDPS*, 2010, pp. 1–12.

[31] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part ii, multidimensional time," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 90–100.

[32] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," *U. of Southern California, Tech. Rep*, pp. 08–897, 2008.

[33] K. Hoste, A. Georges, L. Eeckhout, and L. Eeckhout, "Automated just-in-time compiler tuning." in *CGO*. IEEE, 2010, pp. 62–72.

[34] D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin *et al.*, "Peri auto-tuning," in *Journal of Physics: Conference Series*, vol. 125. IOP Publishing, 2008, p. 012089.

[35] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008, pp. 165–174.

[36] P. Lokuciejewski, S. Plazar, H. Falk, P. Marwedel, and L. Thiele, "Multi-objective exploration of compiler optimizations for real-time systems," in *ISORC*, 2010, pp. 115–122.

[37] K. Heydemann and F. Bodin, "Iterative compilation for two antagonistic criteria: Application to code size and performance," in *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.

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

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

[40] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan, "Parametric multi-level tiling of imperfectly nested loops," in *ICS*, M. Gschwind, A. Nicolau, V. Salapura, and J. E. Moreira, Eds. ACM, 2009, pp. 147–157.

[41] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Dyntile: Parametric tiled loop generation for parallel execution on multicore processors," in *IPDPS*. IEEE, 2010, pp. 1–12.

[42] L. Renganarayanan, D. Kim, S. V. Rajopadhye, and M. M. Strout,

"Parameterized tiled loops for free," in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 405–414.

[43] J. Mars and R. Hundt, "Scenario based optimization: A framework for statically enabling online optimizations," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 2009, pp. 169–179.

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

[45] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.