Context-aware Composition of Parallel Components

Welf Löwe and Christoph Kessler

1 Purpose and Aims

Multi- and many-core processors will be the predominant technology for virtually all system platforms in the foreseeable future, and the performance of software will increase if (and only if) it exploits parallel execution. This will add another level of complexity to software development. Reusable components are a well-proven means of handling software complexity. However, when maximizing reusability, we construct general components that don't fit any particular context well. Hence, optimization as a technique ought to be an integral part of building systems from components.

However, we observe that composition and optimization are currently discussed almost isolated in two different scientific communities. Composition is understood as a problem in software engineering for general-purpose computing; techniques include meta-/generative programming, configuration and (self-)adaptation, architecture and component systems. Optimization is typically understood as a problem of high-performance and stream computing, and compiler construction; techniques include autotuning, scheduling, parallelization, just-in-time compilation, runtime optimizations, and performance prediction.

The **purpose** of this project is to combine solutions of these two research areas. Using variant malleable task scheduling, we co-optimize schedule, resource allocation, and algorithm selection of predefined parallel components. Thereby, we achieve both (i) cost-efficient software production due to the reuse of predefined general components and, (ii) high quality of the composed system due to optimization of the components and the whole systems in their specific reuse contexts.

It is the **aim** of this project to contribute with experimentally validated basic research on general methods and tools for the automated optimization of programs built from components. We aim for solutions for parallel components and target systems, especially multi-core systems. Additionally, we aim for solutions applicable to static system environments (the hardware configuration is not changing after the program is deployed) and dynamic system environments (changing at runtime), and static and dynamically changing optimization goals.

We consider the project a success if systems composed and automatically optimized with our methods achieve a quality comparable with carefully manually optimized systems. Pre-studies indicate that this is realistic. To evaluate success, we optimize the runtime performance and memory consumption of parallel benchmark applications on a number of parallel machines.

2 Survey of the Field

Components are a unit of composition with contractually specified interfaces and explicit context dependencies only [15]. Composition is a recursive process: a component may be basic or constructed by composing other (sub-)components.

Components suitable for multi/many-core and other parallel platforms should implement the composition interface with explicitly parallel code. This means that a component may contain sub-components called and executed in parallel. The tasks defined by calls to a component interface are known as *malleable tasks* in scheduling theory. Malleable tasks may

be executed on one or many processors; their execution time (usually) decreases with the number of allocated processors.

Different component variants may implement the same functionality and interface but show different performance depending on the call context, i.e., on the actual parameters (determining, e.g., the problem sizes) and on the available resources (e.g., the number of processors available). Such component variants could each contain a different algorithm solving the same problem, wrap the use of hardware accelerators (e.g., GPUs) where available, or emanate from each other by applying compiler transformations. Note that the variants may sometimes implement the interface only in special cases, i.e., under restricted context preconditions.

The selection of the most suitable component variant in a composition context is an optimization problem [W29,C35][1]. Moreover, component variants that contain parallel code profit from resource allocation and scheduling [C11,W27], an optimization problem as well. More specifically, for each composition context (defining a required component interface), the composition needs to select: (1) the expectedly best component implementation variant, (2) the number of processors to spend on the selected variant, and the schedule for the parallel code of the selected variant, such that the overall expected execution time is minimized. We refer to this optimization problem as the *variant malleable task scheduling* problem. It is NP-hard as it contains the malleable task scheduling problem as a special case. The problem of scheduling malleable tasks without (with) precedence constraints is strongly NP-hard even for five (three) processors [5]. We proposed and implemented an exact solver of the variant malleable task scheduling problem with exponential complexity based on dynamic programming [W27], but faster heuristic algorithms are required for larger problem instances.

We claim that any approach to our aims should meet three requirements: First, it ought to solve the variant malleable task scheduling problem. This is necessary to cope with the recursive nature of composition and to optimize parallel component variants. Second, it should be general in the sense that the composition and optimization methods must not exploit application domain knowledge. Third, it ought to revise the composition decision in dynamically changing system environments or when the optimization goal changes dynamically.

Automatic program specialization has been a great concern to the composition community for many years. The costs of a system composed from general components are sometimes unacceptable and more efficient, specialized components are preferred in specific system environments. Svahnberg *et al.* present a taxonomy of techniques for variability realization and specialization [14]. For object-oriented languages, Frigo *et al.* [6] and Schultz *et al.* [13] demonstrate how advices from the developer can be used to automatically specialize libraries and applications. These works address static system environments. Our work additionally addresses runtime change of special algorithms and data-structure when admissible [9]. The automated derivation of a *globally* optimal change strategy was left to future work.

Domain-specific library generators achieve adaptive optimizations by using profile data gathered during offline training processes to tune key parameters, such as loop blocking factors to adapt to, e.g., cache sizes. This technique is used for generators that target sequential and parallel libraries, but only works for static system environments. For smaller problem instances, such methods can be compared directly to optimal compositions computed by our dynamic programming based algorithm [W27]. Examples are ATLAS [18] for linear algebra computations, and SPIRAL [10] and FFTW [7] for Fast Fourier Transforms (FFT) and signal processing computations. With its concept of composition plans computed at runtime, FFTW also supports optimized composition for recursive components. SPIRAL

[1] For the citations W_ and C_ refer to W. Löwe's and C. Kessler's publications in Appendix C.

is a generator for library implementations of linear transformations in signal processing. Given a high-level formal specification of the transformation algorithm and a description of the target architecture, SPIRAL explores a design space of alternative implementations by combining and varying loop constructs, and produces a sequential, vectorized, or parallel implementation combining the fastest versions for the target platform. Li *et al.* [8] use dynamic tuning to adapt a sorting library to target machines. They use a number of machine parameters (e.g., cache size, input size, and the distribution of the input) as input to a machine learning algorithm for optimizing the system's performance. The machine learning algorithm is trained to pick the fastest algorithm for any considered scenario. Still the system environment is considered static.

In contrast to these domain-specific auto-tuning systems, our own context-aware composition approach [W27] is general-purpose, and its optimization scope is not limited to libraries but can be applied to complete applications and arbitrary optimization goals.

For parallel target platforms, several approaches combine the optimal selection of algorithms and data representation with resource allocation or scheduling. Again, they are often limited to specific domains. No approach supports the combinations of all these techniques.

For Single Program Multiple Data (SPMD) parallel systems, different approaches to dynamic algorithm selection based on predicate functions generated from training data have been proposed. Brewer [4] investigated such a system for sorting and PDE solving, and also considered limited support for the dynamic selection of array distributions on distributed shared memory machines. Yu and Rauchwerger [19] investigated dynamic algorithm selection for reductions and in the STAPL [16] library for sorting and matrix computations. In these approaches, dynamic selection is only applied for flat composition; calls within the library itself are not considered.

Olszewski and Voss [11] proposed a dynamic adaptive algorithm selection framework for divide-and-conquer sorting algorithms in a fork-join parallel setup. This method is limited to threaded parallel divide-and-conquer algorithms that are, barring the possible parallel execution of subproblems, identical to their sequential counterparts, such as Quicksort with sequential partitioning and Mergesort with sequential merging. Such algorithms do not scale well to larger numbers of processors.

Various static scheduling frameworks for malleable parallel tasks and task graphs of modular SPMD computations with parallel composition have been considered including [12, 20] and our own contributions [W6]. Most of these require a formal machine-independent specification of the algorithm that allows prediction of execution time by abstract interpretation.

Ansel *et al.* suggested an implicitly parallel language PetaBricks [3], allowing multiple implementations of algorithms to solve a problem. The algorithmic choice is a first class construct of the language. Programmers can make choices explicitly or let a compiler (statically) autotune the system. Wernsing and Stitt [17] present an approach that builds upon PetaBricks but targets multi-core CPU systems with FPGA-based accelerators. Like PetaBricks, this approach relies on dynamic scheduling and the learning is done off-line, and like in our approach [W27,W11], the learned execution plan is stored in table form. To support static prediction, computations and execution platforms must be statically predictable with sufficient accuracy. Our recent work compares decision time, execution plan size, and decision accuracy of the table based approach with other approaches known from machine learning including Bayesian classifiers, support vector machines, decision trees and decision diagrams [W10].

Our parallel extension of context-aware composition [W11, W27] is general-purpose and combines algorithms and even data representation selection with resource allocation and scheduling. However, it is still limited to static system environments. In self-adaptive soft-

ware systems – Andersson *et al.* provide a systematic classifications [1] – changing environments and possibly even changing optimization goals are considered at design time. This makes systems resilient with respect to future changes in the requirements and properties of the system environment. Optimizations are often used as means to achieve and readjust non-functional properties of the systems. Significant progress has been made in how self-adaptive systems are designed, constructed, and deployed. Our own contribution for self-adaptive sensor networks and scientific computations, called dynamic model driven architectures [W46–W48], is just one among several approaches. Still there is a lack of fundamental construction concepts [2]. First attempts to introduce self-adaptivity to context-aware composition are reported in [W19].

3 Project Description

3.1 The Context-Aware Composition Approach

Our approach is based on a combination of *learning* and *composition*. The learning phase prepares the composition and excludes some suboptimal solutions. The composition phase performs the actual composition by selecting the (approximative) optimal components variant and determining its resource allocation.

Static learning and composition (at deployment time) are preferred as they lead to faster programs. However, many scenarios relevant for our aims require dynamic composition (at runtime), some even dynamic learning. Static learning is admissible if the component variants and hardware resources are statically known and will not change during runtime. Moreover, the system's optimization goal (a weighted combination of runtime performance and memory consumption and other non-functional system properties) must not change in this case. Otherwise, learning is part of the program execution. Composition may happen statically, if static learning was admissible and if the context properties relevant for the optimal component variant selection and resource allocation can be evaluated statically, too. This is rather unlikely and, hence, composition is almost always done at runtime.

The learning phase generates composition code which selects the expectedly optimal component and allocates resources. More precisely, composition code selects the component variants, the processors allocated for them and their schedules depending on the actual composition context, e.g., the actual parameters and the number of processors available etc.

The composition code can be understood as a dispatch table: for each context, the table contains a pointer to the correspondingly best component variant and resource allocation. More specifically, each relevant composition *context property* (e.g., each relevant formal parameter and the number of processors available) corresponds to a dimension of the dispatch table. For each such context property, we abstract the relevant context property *values*. For instance, we classify actual parameters by their sizes and the actual number of processors available in certain intervals. This leads to the different entries in the dispatch table, each containing an expectedly optimal variant and resource allocation. Note that the dispatch table is just one implementation variant of the composition code.

Learning requires assessing a large number of actual contexts, comparing the quality of different variants and resource allocations in each such context and capturing the champion component variant and resource allocation in a dispatch table. The contexts assessed may be generated systematically in offline training or monitored execution situations. The quality of different component variants and their different resource allocations may be profiled, derived analytical using machine and cost prediction models, or a combination thereof.

Our approach is generic and could be used with any component-based or modular, parallel

programming model that provides the interface concept and the concept of implementation variants using, e.g., inheritance. It applies to sequential programs as a special case.

3.2 Issues of Context-Aware Composition to Address

It is a general issue to find a trade-off between the benefit and the complexity of optimizations with several aspects. In the settings with changing environments and optimization goals, we additionally trade off simplicity of learning and optimization against the quality of the optimized solution. Only few trade-off problems can be decided analytically. Hence, prototype implementations and case studies are required to evaluate the alternatives.

Optimization benefit vs. complexity A space-efficient representation of the dispatch tables is important. Multidimensional dispatch tables [W27] allow for very fast lookup but, given limited runtime storage, they cannot be used in large systems. We will investigate alternative machine learning approaches and table compression techniques and compare their dispatch behavior to tables, cf. [W10] for first results. In general, we bias between dispatch speed and storage size and between these dispatch properties and the overall optimization benefit.

It is also an issue to find out how much we can gain by combining the subproblem's algorithm selection, scheduling with resource allocation, and representation selection, compared to solving these problems separately in some order. Is the increased problem complexity worth the expected improvements? What are the trade-offs? Under what conditions can we identify appropriate solutions based on a *partial* integration of solutions for subproblems?

Another trade-off regards the modeling of the target architectures. The different parallel and multi-core architectures including their storage units and hierarchies should be modeled appropriately in order to generate realistic expectations of the costs of the variants under certain resource allocations and schedules. In particular, the modeling of the performance effect of limited off-chip memory bandwidth and other limited shared resources in the framework is an open issue. However, more complex machine models require more complex resource allocation and scheduling heuristics. What is an acceptable trade-off between the accuracy of the models and effects of the heuristics?

Finally, we could statically pre-cluster components and allow dynamic composition only for these clusters. A possible approach could merge contiguous subtrees of the dynamic call graph that will be internally composed statically and treated as an atomic component for dynamic composition. This will somewhat decrease quality as we miss some better choices of variants within these clusters, but, it will save dynamic composition overhead.

Changing Environments and Optimization Goals In a non-adaptive setting, as adopted by virtually all previous approaches, the dispatch tables are fixed after offline learning. However, their accuracy depends on whether the given offline learning data really represented the actual executions. This cannot always be guaranteed as the environment of a system may change at runtime. For instance, new applications may require hardware resources and, hence, the dispatch table becomes invalid, in general. Re-learning the system is then necessarily an online activity.

Even the optimization goal may change dynamically. For instance, the user of a long-running (stream) application may upgrade the service level agreement with a system provider such that memory limitations do not apply any longer. Of course the dispatch table is outdated then and dynamic re-learning is required. In the simplest case, the relevant context properties remain the same. In this case, the optimization goal function is a black box that

can be replaced dynamically before re-learning. The old and the new optimization goal functions conform to the same signature; both are stateless functions. Hence, substitution may use standard means like dynamic class loading.

In the more complex case, the relevant context properties change as well, i.e., the new optimization goal is based on new properties of the contexts. For instance, in the case where the user downgrades the service level agreement such that the memory is limited now, the memory available might not have been monitored as a context property before. Still, we need to change the black box representing the optimization goal function. However, its signature has changed in addition to its implementation. Moreover, we ought to deploy new infrastructure accessing the new context properties.

Prototype implementations and case studies for the selected target platforms demonstrate the principles of our methods and solution approaches and quantitatively evaluate their success. Benchmark components will include sorting, dense and sparse matrix computations with standard dense, sparse and recursive matrix layout, and block/cyclic/replicated array distributions where appropriate. Also, we include FFT and signal processing in the benchmarks, possibly even string matching, ODE solving, and combinatorial optimization. Extending the core benchmark suite and accelerating it for specific parallel target platforms are very suitable Master's thesis projects.

3.3 Methods, Tasks, and Preliminary Schedule

An analytic assessment of the many trade-offs is not expected to work. Hence, the research method will be iterations of modeling (hypothesis building) and validating the models (experimental hypothesis testing). As experiments constitute a large part of the work, we accept to spend some time with developing an efficient experimental framework allowing us to vary applications and target platforms as well as performing static and dynamic learning and composition. The framework should also allow simply plugging-in the different trade-off decisions of optimization benefit vs. complexity into corresponding variation points.

WP 1: Building the experimental framework – 6 Person Months (PM) The framework should be defined and implemented. To verify the framework, all prototypes for experiments, cf. Section 5, should be integrated into the framework and repeated for more benchmark applications and target platforms.

WP 2: Optimization benefit vs. complexity – 12 PM For the different decision points discussed in Section 3.2, trade-off hypotheses will be defined, implemented correspondingly as plug-ins into the experimental framework, and validated. In WP 2, we will trade off optimization benefits against simple and fast optimizations. This is a preparation for WP 3/4, requiring dynamic learning (including optimization) and composition.

WP 3: Changing environments and optimization goals, Baseline – 6 PM We extend context-aware composition such that the system environment and optimization goal may change at runtime. In WP 3, we aim for baseline solutions, i.e., we are satisfied with naive approaches as long as they outperform the systems that do not adapt to change. This goes along with an extension of the experimental framework: new variation points will be added, e.g., for the optimization goal function, with the respective naive solution as the first plug-in. The extended experimental platform will be evaluated for all benchmark applications and target platforms used in WP 1/2.

WP 4: Changing environments and optimization goals, Advanced – 12 PM Improved methods discussed before in Section 3.2 will be defined (hypotheses), implemented as plug-ins into the extended framework, and validated. Validation is successful if improvement over the baseline approaches can be shown for all benchmark programs and target platforms.

4 Significance

Multi- and many-core processors will dominate not only in high-performance computing but also the desktop, digital signal processing, gaming, and embedded domains. With new hardware systems in these domains, the performance of software will increase if (and only if) applications exploit the parallel execution of independent application tasks. Considering the ever growing complexity of software, novel component-based software development methods are required that allow for efficient programming of massively parallel systems and, at the same time, lead to composed systems that effectively exploit the hardware resources of these parallel platforms. This project contributes with such methods and prototype tools.

On the other side, the project will promote the parallel component concept in high-performance computing. It will simplify the design and use of parallel component libraries. This will encourage future research in the design and analysis of parallel algorithms and data structures. In particular, it will allow quantitatively documenting the landscape of parallel algorithms over parallel architectures leading to a deeper understanding of the interaction between algorithms and architectures. The methods and tools built in the project will contribute to this vision.

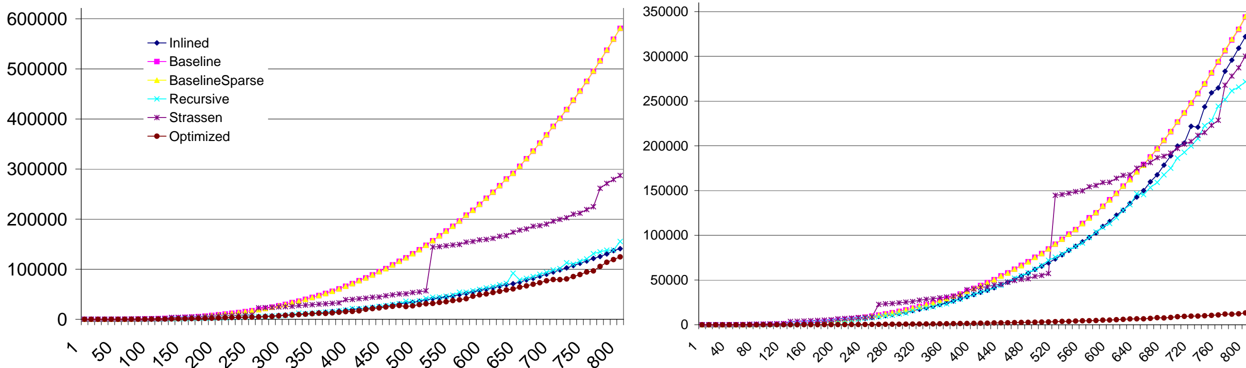
5 Preliminary Results

Optimal scheduling of independent malleable tasks is NP-hard and, hence, way too expensive to be applicable in general. However, good approximation algorithms [W6] could even be generalized to time-efficient heuristics solving the more general variant malleable task scheduling problem [W27,W11].

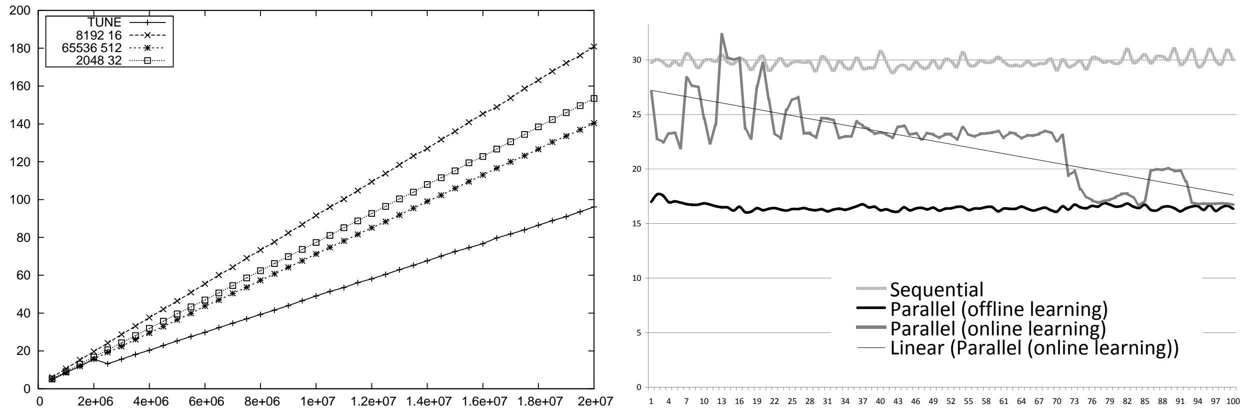
In proof-of-concept studies, we implemented our context-aware composition: approach controlling different variables. *Platform*: we composed sequential and parallel components on a Linux cluster [C11], a simulated shared-memory parallel computer [W27,W11], an Intel Dual Core T2300 PC, an Intel Core i5 Mac, an Intel 8 Core 5450 IBM Blade Server [W10], an Nvidia C2050 (Fermi) GPU [C35]. *Programming model*: We implemented the variants in Java, Fork, C using native threads, CUDA, OpenCL, OpenMP. *Domains*: We evaluated the work empirically with example components from the sorting domain [C11,W27], from matrix computations with dense and sparse representations [W29], and various operation skeletons [C35]. *Learning time*: We used both offline and, most recently, online learning (not yet published). In all pre-studies, our measurements demonstrated a significant potential of performance gains for the composed and optimized programs compared to the classical component-based implementations. Figure 1 shows selected results from these studies.

Fig. 1(a) shows matrix multiplications in Java on an Intel Dual Core T2300 processor. Learning was done offline, composition online. Automatic algorithm selection at each call yields a decent speedup (left). Combined with data-structure selection, it yields a tremendous speedup (right). Here, a sparse multiply algorithm is automatically selected after automatically converting a sparse operand to the appropriate sparse data structure.

Fig. 1(b)(left) shows results of element-wise parallel addition using a generic CUDA component (map) from the SkePU skeleton library on an Nvidia C2050 (Fermi) GPU. The map skele-



(a) Matrix multiplication algorithm and data-structure selection for different matrix sizes. x-axis displays number of matrix lines and rows. y-axis displays time in $msec$.



(b) Simple vector map operations; x-axis displays number of vector elements, y-axis time in $msec$ (left). Sorting of arrays with 10^5 elements; x-axis displays iterations of calls to sort, y-axis time in $msec$ (right).

Figure 1: Selected experimental results from pre-studies.

ton was tuned offline from measured training data for optimal back-end selection, resource allocation and choice of tunable parameters (for CUDA components: number of threads and thread block size). The composed code (TUNE) outperforms any fixed choice of back-end, resource allocation, and tunable parameters. Further examples combine this with variant selection (single vs. multi-GPU CUDA, OpenCL, OpenMP, sequential), cf. [C35].

Fig. 1(b)(right) shows results of sequential and parallel variants of Quicksort, Mergesort, and Insertion sort on an Intel i5 processor (2 cores). The fastest individual variant (sequential Quicksort) is outperformed by the two optimizing approaches selecting automatically sequential and parallel variants. One of these optimizing approaches learns offline, the other one online. The latter is slow only at the beginning when it tries suboptimal variants but, its performance converges to the one of the offline approach after some iterations.

6 Part of project cost

The project is planned for three years; we apply for $\approx 87\%$ of the total project cost.

7 Budget Motivation

Project costs include the salaries for a PhD student (80% research in the project) and for the applicants (20% research in the project each). However, 10% of Christoph Kessler's

salary are taken from faculty funding resources. Additionally, the costs include a conference travel budget for guaranteeing the scientific exchange within the project and the scientific community. In the first year, the costs for a PC for the PhD student add to the annual costs. Overhead costs of 52% on the salary cover office space for all researchers involved and the access to administrative services at the host university.

8 Equipment

The applicants have access to a variety of parallel architectures, e.g., to 8 core IBM Blade and Xeon Quadcore servers at DFM Växjö, to large clusters of these in Uppsala (56 CPU IBM JS20 Blade Center) and NSC Linköping, respectively, and to a Supermicro 8-core (2 Intel Xeon) server with 2 Nvidia C2050 (Fermi) GPUs at PELAB Linköping.

9 International and National Collaborations

Our project is closely related to the EU FP7 project PEPPHER which applies the approach to performance portability and programmability of heterogeneous many-core systems, including GPU and Cell-like architectures. Partners include 4 universities (Vienna, Chalmers, Karlsruhe and Linköping), INRIA Bordeaux, Intel Deutschland GmbH, Codeplay Ltd., and Movidius Ltd. In this proposed VR project, we will instead focus on homogeneous multi-core systems and clusters, on more explicit parallel resource management, and on more dynamic scenarios. Another major difference of the proposed project and PEPPHER is that the latter uses a run-time system for dynamic scheduling and resource allocation that has been designed for sequential tasks and currently does not work well for parallel malleable tasks, and thereby disregards statically co-optimizing these aspects with algorithm selection.

The applicants collaborate (joint publications and projects) with other experts in the fields of composition and optimizations, e.g., with Prof. Uwe Aßmann at Dresden University of Technology, Germany, Dr. Martti Forsell, VTT Oulu, Finland, Prof. Sergei Gorlatch, University Münster, Germany, Prof. Jörg Keller at University Hagen, Germany, Prof. Bo Thidé, Uppsala University, Prof. Denis Trystram, Grenoble Institute of Technology, France, and Prof. Wolf Zimmermann, University Halle, Germany.

In 2010, the applicants together with Prof. David Padua, University of Illinois, USA, and Prof. Markus Püschel ETH Zürich, Switzerland, organized a Dagstuhl Seminar on *Program Composition and Optimization: Autotuning, Scheduling, Metaprogramming and Beyond* leading to further collaborations.

10 Other Grants

PEPPHER Performance Portability and Programmability for Heterogeneous Many-core Architectures, EU FP7 project, Jan. 2010 – Dec. 2012. Kessler is leading the workpackage *Compositional parallel software development*. Löwe is associated consortium member.

Optimal code generation for loops integrating instruction selection, cluster assignment, scheduling and register allocation including optimal spill code generation and scheduling, for embedded, VLIW and clustered VLIW processors. VR 2010 – 2012. Kessler is PI.

References

[1] Jesper Andersson, Rogério Lemos, Sam Malek, and Danny Weyns. Modeling dimensions of self-adaptive software systems. pages 27–47, 2009.

[2] Jesper Andersson, Rogério Lemos, Sam Malek, and Danny Weyns. Reflecting on self-adaptive software systems. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems, colocated with ICSE'2009*, pages 38–47, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Not.*, 44(6):38–49, 2009.

[4] Eric A. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Not.*, 30(8):80–91, 1995.

[5] Jianzhong Du and Joseph Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[6] J. Frigo, R. Neumann, and W. Zimmermann. Mechanical generation of robust class hierarchies. In *Tools-23*, page 282ff. IEEE Computer Society, 1997.

[7] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue "Program Generation, Optimization, and Platform Adaptation".

[8] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proc. Int. Symposium on Code Generation and Optimization (CGO'04)*, page 111ff. IEEE Computer Society, 2004.

[9] Welf Löwe, Rainer Neumann, Martin Trapp, and Wolf Zimmermann. Robust dynamic exchange of implementation aspects. In *Tools-29*, page 351ff. IEEE Computer Society, 1999.

[10] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.

[11] Marek Olszewski and Michael Voss. An install-time system for automatic generation of optimized parallel sorting algorithms. In *Proc. PDPTA*, pages 17–23, 2004.

[12] Thomas Rauber and Gudula Rünger. Compiler support for task scheduling in hierarchical execution models. *J. Syst. Archit.*, 45(6-7):483–503, 1999.

[13] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In *Proc. 13th European Conf. on Object-Oriented Programming (ECOOP'99)*, pages 367–390. Springer, 1999.

[14] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software–Practice and Experience*, 35(8):705–754, July 2005.

[15] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 277–288, 2005.

[17] John Robert Wernsing and Greg Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proc. ACM SIGPLAN/SIGBED conf. on Languages, compilers, and tools for embedded systems (LCTES'10)*, pages 115–124. ACM, 2010.

[18] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[19] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1084–1096, 2006.

[20] Lei Zhao, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd. Predictive performance modelling of parallel component composition. In *Proc. 19th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 15*, 2005.