Program Composition and Optimization: An Introduction

Christoph W. Kessler[1], Welf Löwe[2], David Padua[3], and Markus Püschel[4]

[1] Linköping University, Linköping, Sweden, `chrke@ida.liu.se`
[2] Linnaeus University, Växjö, Sweden, `welf.lowe@lnu.se`
[3] University of Illinois at Urbana Champaign, USA, `padua@uiuc.edu`
[4] Carnegie Mellon University, USA, `pueschel@ece.cmu.edu`

1 Topic Overview

Software composition connects separately defined software artifacts. Such connection may be in program structure (such as inheritance), data flow (such as message passing) and/or control flow (such as function calls or loop control).

In the classical sense of connecting black-box software components, *composition* denotes just the process of binding a call to a callee or of a message producer to a message consumer in another component. In a broader interpretation, it also includes the binding of certain design decisions connected with calls, such as the choice of appropriate data structures for operands, the choice among several applicable implementations for the task to be computed, the allocation of resources (e.g., remote computers on a grid, additional processor cores, or special hardware accelerators) for the task to be performed by the call, and scheduling of multiple calls. Finally, composition may abstract completely from call site and mechanism and combine arbitrary software artifacts such as type or code fragments at arbitrary locations anticipated for composition. This general view includes meta-programming, program generation, aspect composition, and invasive software composition, but also traditional compiler optimizations and program transformations. For instance, it allows to connect sequential problem-specific code fragments with generic code templates for the platform-specific coordination of independent loop iterations to enable parallel execution, as adopted in the skeleton programming approach.

Beyond the narrow sense of classical call binding, all these composition decisions may involve choices between different alternatives, which generally affect the cost (e.g., execution time, memory requirements, energy consumption) of the resulting program, and may also involve trade-offs between, e.g., execution time and energy consumption. Such optimizing composition decisions should preferably be delayed until enough information is available to provide reasonable predictions to compare the alternatives, which may be at compile time, deployment time, load time, or run time. The decisions may also be subject to global constraints such as program length, memory capacity or energy budgets. Some decisions may be local, but in general, choices for one composition/optimization issue may influence choices for others. In other words, we are dealing with complex global integrated program optimization problems.

Dagstuhl Seminar Proceedings 10191
Program Composition and Optimization : Autotuning, Scheduling, Metaprogramming and Beyond
http://drops.dagstuhl.de/opus/volltexte/2010/2573

However, straightforward late binding for optimization also incurs additional overheads, for instance, when predicting performance and choosing between different implementations or schedules at run-time. It is possible to keep such overheads relatively low by anticipating much of the prediction and evaluation effort at deployment time, for instance in the form of *auto-tuning* of libraries, where, for instance, the fastest algorithm or implementation for a specific computation instance or the best blocking factor for a loop is determined as a function of a few characteristic problem parameters. This selection function may be constructed at deployment time by, for instance, a machine learning algorithm that uses training data obtained by profiling on the target platform. Today, auto-tuning approaches are successfully used for the generation of optimized, domain-specific libraries, e.g., for signal processing or linear algebra computations. We believe that auto-tuning has, beyond these special cases, a considerable potential of generalization, namely towards software composition in general, including scheduling and resource allocation issues.

The design optimization space of possible implementation variants for a single specific computation can already be very large, as it also includes the application of various compiler transformations, various representations for operand data types, or the use of hardware accelerators. Combining the implementation selection problem with scheduling, resource allocation and other optimization problems additionally increases the optimization potential, but also the complexity. The space of alternatives may be generated and evaluated systematically at deployment time, for instance by using meta-programming techniques.

2 Questions and Challenges

In the following, we list some of the challenges and open questions that we raised as part of the preparation material for the seminar *10191 Program Composition and Optimization: Autotuning, Scheduling, Metaprogramming and Beyond*:

- What structures and artifacts in programs or program components are suitable variation points for optimizations? How should they be exposed, explicitly or implicitly? How can the programmer of a (third-party) component influence the optimization spectrum or the predictions during the composition process? What are necessary extensions to component interfaces and contracts to enable more aggressively optimized compositions? How should software architectures be organized to support optimized composition? Are there different appropriate solutions for different target platforms, e.g. for many-core processors vs. computational grids?
- What is the limit for the application of auto-tuning methods? Are they confined to a few “benign” application domains, or can they be generalized to a much wider area? How can they support optimizing composition in the general sense?
- Which program optimizations and optimizing composition issues should be considered together (solved globally) for what type of target platform? Which problems can be solved locally under what conditions?

- How can we reduce the complexity of the optimization space exploration especially for combined problems? What methods have been useful in previous approaches, and how can they be adapted or improved?
- Is performance compositional?
- What are the appropriate prediction methods for time, energy or space on various platforms to be used when deciding about optimizing composition? What are the trade-offs between analysis cost and accuracy? How much can or should be precomputed earlier in the composition process, e.g. at deployment time? How can such precomputed information be stored and retrieved efficiently?
- What is the complexity of the variant malleable task scheduling problem that appears when optimizing several calls to functions with multiple parallel and sequential implementation variants? What are suitable solution methods for it?
- How can adaptivity to changes in the available resources (e.g., handling grid nodes joining or leaving dynamically, discovering new system components, or handling failure) or software components (dynamic update or addition) be properly taken into account in optimizing composition?
- How can this technology be transferred to sensor-grid and ubiquitous computing applications?

References

1. G. Almasi, L. DeRose, B. Fraguela, J. Moreira, and D. Padua: Programming for Locality and Parallelism with Hierarchically Tiled Arrays. Proceedings of the Sixteenth International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), pp. 162-176, College Station, Texas, October 2-4, 2003. Springer *Lecture Notes in Computer Science* vol. 2958, 2004.
2. Jesper Andersson, Morgan Ericsson, Welf Löwe: Reconfigurable Scientific Applications on GRID Services. European Grid Conference 2005. Amsterdam. Published in LNCS Vol. 3470, Advances in Grid Computing: EGC 2005, Springer, P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld and M. Bubak (eds.), 2005.
3. Jesper Andersson, Morgan Ericsson, Christoph Kessler, Welf Löwe: Profile-Guided Composition. Proc. 7th Int. Symposium on Software Composition (SC 2008) at ETAPS, Budapest, Hungary, March 2008. Springer LNCS 4954: 157-164.
4. Uwe Aßmann: *Invasive Software Composition*. Springer, 2003.
5. Srinivas Chellappa, Franz Franchetti and Markus Püschel: How To Write Fast Numerical Code: A Small Introduction Proc. Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), Lecture Notes in Computer Science, Springer, Vol. 5235, pp. 196-259, 2008
6. Morgan Ericsson, Welf Löwe, Christoph Kessler, Jesper Andersson: Composition and Optimization. Proc. Int. Workshop on Component-Based High Performance Computing (CBHPC-2008), Karlsruhe, Oct. 2008.
7. Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman: ACME: adaptive compilation made efficient. Proc. ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems (LCTES '05), Chicago, Illinois, USA, pp. 69–77, 2005.

8. Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle: Portable compiler optimisation across embedded programs and microarchitectures using machine learning. Proc. 42nd Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO-42), pp. 78–88, ACM, 2009.
9. Franz Franchetti, Yevgen Voronenko and Markus Püschel: A Rewriting System for the Vectorization of Signal Transforms Proc. High Performance Computing for Computational Science (VECPAR), Lecture Notes in Computer Science, Springer, Vol. 4395, pp. 363-377, 2006
10. Franz Franchetti, Yevgen Voronenko and Markus Püschel: FFT Program Generation for Shared Memory: SMP and Multicore Proc. Supercomputing (SC), 2006
11. Matteo Frigo and Steven G. Johnson: FFTW: An adaptive software architecture for the FFT. Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP), vol. 3, pages = 1381–1384, 1998.
12. Matteo Frigo and Steven G. Johnson: The Design and Implementation of FFTW3. *Proc. of the IEEE* **93**(2):216–231, special issue on "Program Generation, Optimization, and Adaptation", 2005.
13. E.-J. Im, K. Yelick, and R. Vuduc: Sparsity: Optimization Framework for Sparse Matrix Kernels. *Int'l J. High Performance Computing Applications* **18**(1), 2004.
14. Christoph Kessler and Welf Löwe. A Framework for Performance-Aware Composition of Explicitly Parallel Components. Proc. ParCo-2007 conference, Jülich/Aachen, Germany, Sept. 2007.
15. Christoph Kessler and Welf Löwe. Optimized Composition of Performance-Aware Parallel Components. Proc. 15th Int. Workshop on Compilers for Parallel Computers (CPC-2010), Vienna, Austria, July 2010 (to appear).
16. T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle: Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Washington, DC, USA, p. 237, IEEE Computer Society, 2000.
17. P. M. W. Knijnenburg and T. Kisuki and M. F. P. O'Boyle: Iterative compilation. In: *Embedded processor design challenges: systems, architectures, modeling, and simulation (SAMOS)*, pp. 171–187, Springer, 2002.
18. Xiaoming Li, María J. Garzarán, and David Padua: A Dynamically Tuned Sorting Library. Proc. Int. Symposium on Code Generation and Optimization (CGO'04), San Jose, California, March 20-24, 2004, pp. 111–124.
19. Xiaoming Li, María J. Garzarán, and David Padua: Optimizing Sorting with Genetic Algorithm. Proc. 3rd Int. Symposium on Code Generation and Optimization (CGO-2005), pp. 99-110, San Jose, CA, USA, 2005.
20. Peter A. Milder, Franz Franchetti, James C. Hoe and Markus Püschel: Formal Datapath Representation and Manipulation for Implementing DSP Transforms Proc. Design Automation Conference (DAC), pp. 385-390, 2008
21. Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson and Nicholas Rizzolo: SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE* **93**(2):232–275, special issue on "Program Generation, Optimization, and Adaptation", 2005.
22. Yevgen Voronenko, Frédéric de Mesmay and Markus Püschel: Computer generation of general size linear transform libraries. Proc. International Symposium on Code Generation and Optimization (CGO), pp. 102–113, 2009.
23. Richard Vuduc, James W. Demmel, and Katherine A. Yelick: OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* **16**:521–530, Institute of Physics Publishing, 2005.

24. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra: Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing* **27**(1–2): 3–35, 2001.
25. Jianxin Xiong, Jeremy Johnson, Robert W. Johnson and David Padua: SPL: A Language and Compiler for DSP Algorithms. Proceedings of the 2001 ACM Conference on Programming Language Design and Implementation (PLDI 2001), pp. 298-308, Snowbird, Utah, June 20-22, 2001.
26. Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill: Is Search Really Necessary to Generate High-Performance BLAS?, *Proceedings of the IEEE* **93**(2), special issue on "Program Generation, Optimization, and Adaptation", 2005
27. Wolf Zimmermann and Welf Löwe: Foundations for the integration of scheduling techniques into compilers for parallel languages. International Journal of Computational Science and Engineering 1(3/4), 2005.