Program Composition and Optimization: 
An Introduction

Christoph W. Kessler\textsuperscript{1}, Welf Löwe\textsuperscript{2}, David Padua\textsuperscript{3}, and Markus Püschel\textsuperscript{4}

\textsuperscript{1} Linköping University, Linköping, Sweden, chrke@ida.liu.se
\textsuperscript{2} Linnaeus University, Växjö, Sweden, welf.lowe@lnu.se
\textsuperscript{3} University of Illinois at Urbana Champaign, USA, padua@uiuc.edu
\textsuperscript{4} Carnegie Mellon University, USA, puschel@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.
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


