Software Synthesis for High Productivity ExaScale Computing

Armando Solar-Lezama (MIT), Rastislav Bodik and James Demel (UC Berkeley)

This project builds on recent results on software synthesis to develop a programming model that integrates validation, synthesis and autotuning. The programming model is based on the concept of programmer guided synthesis; the idea is to leverage synthesis and validation technology while leaving programmers in control of the high-level implementation decisions. The goal is to simplify the process of programming on high performance heterogeneous architectures.

Key components of the programming model

Program Refinement: The programming model supports stepwise development through refinement. A refinement of a program P is program Q that computes the same task as P. Often Q is obtained from P by optimizing a specific aspect of P's computation. For example, program P may assume a simple shared memory model while Q implements a more scalable strategy for partitioning data over distributed memory.

Many high-performance applications are already developed this way; developers start with a simple reference implementation, and optimize different aspects of the program one at a time, checking their work after each step by testing each new version of the program against the previous one. Our programming model provides direct support for this development strategy by allowing programmers to describe these refinement relationships, and leverages this information to help with the more difficult aspects of implementing high-performance code.

Functional Equivalence Checking: One of the ways we support the refinement model is by helping programmers check that each new iteration of the system is indeed a refinement of the previous version. This validation is performed using model checking algorithms implemented on top of SAT/SMT solvers, as well as automated testing techniques that have proven effective for both hardware and software verification. The key research challenge is the application of these techniques to exascale programs with billion-way parallelism. The key to scalability in this setting lies in symmetry reduction, exploiting the fact that most threads in such a program are doing more or less the same thing.

Synthesis with partial programs: When performing a refinement, programmers are expected to have a high-level idea of the overall goal of the refinement—e.g. replace accesses to a global array with local accesses followed by a global exchange step. However, the low-level details of such refinement are often difficult to derive by hand. Synthesis can automatically derive these low-level details when given adequate information about the high-level structure of the solution. The synthesizer can also be forced to focus on solutions that satisfy important resource constraints, such as avoiding bank conflicts or ensuring a balanced partitioning of a data-structure. In this way, programmers can focus on the high-level implementation strategies without having to worry about the low-level details.

Autotuning: When multiple correct completions to a template exist, they will be performance-evaluated with an autotuner through empirical evaluation. In this way, autotuning and synthesis collaborate with each other to enumerate and explore large spaces of correct implementations to arrive at an efficient implementation of the high-level idea provided by the programmer.

The Language Infrastructure

For this project, we have been experimenting with a language called Sketch, which is similar to C but includes number of new features that make it easier to develop high-performance code, such as support for multi-dimensional arrays and the ability to introduce high-level abstractions without incurring runtime cost.

Sketch is able to produce clean C++ with MPI, simplifying integration with existing code bases. We have now been using our infrastructure to implement NAS parallel benchmarks such as Conjugate Gradient, FFT and MultiGrid, and our infrastructure is now capable of generating efficient C++/MPI code.

The language provides high-level support for refinement, including the ability to refine from a sequential to a distributed memory implementation.

The infrastructure developed as part of this project is currently being integrated into Rose as part of the D-tec project in order to provide synthesis functionality to embedded DSLs in C++ and Fortran.

Exploring Synthesis Enabled DSLs

We have also been exploring new mechanisms to make synthesis functionality available to DSL designers to simplify the development of Domain Specific Languages. We have been working on a system called Rosette which is built on top of Racket which allows synthesis functionality to be added to domain specific languages.

With Rosette, all a designer has to do is write a simple interpreter for a DSL, and that is enough to allow the DSL to make use of all the synthesis functionality. Moreover, the DSL designer can provide domain specific rules to enable the synthesizer to leverage domain specific properties to make the synthesis process more efficient.