H2020 FETHPC-1-2014

An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism

Project Number 671603

D3.4 – Full AllScale Compiler Prototype

WP3: High-level parallel API and API-aware optimising source-to-source compiler

Version: 1.0
Author(s): Peter Thoman (UIBK)
Date: 2018-04-30

Due date:	PM31
Submission date:	2018-04-30
Project start date:	2015-10-01
Project duration:	36 months
Deliverable lead organization	UIBK
Version:	1.0
Status	Final
Author(s):	Peter Thoman (UIBK)
Reviewer(s)	Fearghal O'Donncha (IBM) Kiril Dichev (QUB)

Dissemination level	
PU	*Public*

Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UBIK), FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN'S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMECA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information

Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	2018-04-12	First draft	Peter Thoman
0.2	2018-04-27	Integrate IBM feedback	Fearghal O'Donncha, Peter Thoman
1.0	2018-04-30	Integrate QUB feedback, finalize document	Kiril Dichev, Peter Thoman

Table of Contents

Executive Summary

This document describes the status and structure of the Full AllScale Compiler Prototype (D3.4). This is a **source code deliverable**.

Instructions on how to obtain, build and run the prototype will be provided as well as an overview of its implementation and components.

1 Introduction

Deliverable D3.4 is a source code deliverable providing the full prototype of the AllScale Compiler. This prototype maps the basic core primitives of the most recent iteration of the AllScale API (D3.2) – the *pfor* and *fun* calls as well as *treeture*s and operations on them – to the runtime interface specified by deliverables D4.1 and D4.2.

Significantly, compared to the initial prototype (D3.3), this full prototype includes sophisticated constraint-based analysis to determine the symbolic data requirements associated with each parallel region. These are provided in the output program to serve as input for the runtime data management system (D4.4).

The compiler fits into the overall AllScale software infrastructure as indicated in Figure 1, depending on the external Insieme compiler as an underlying implementation technology.

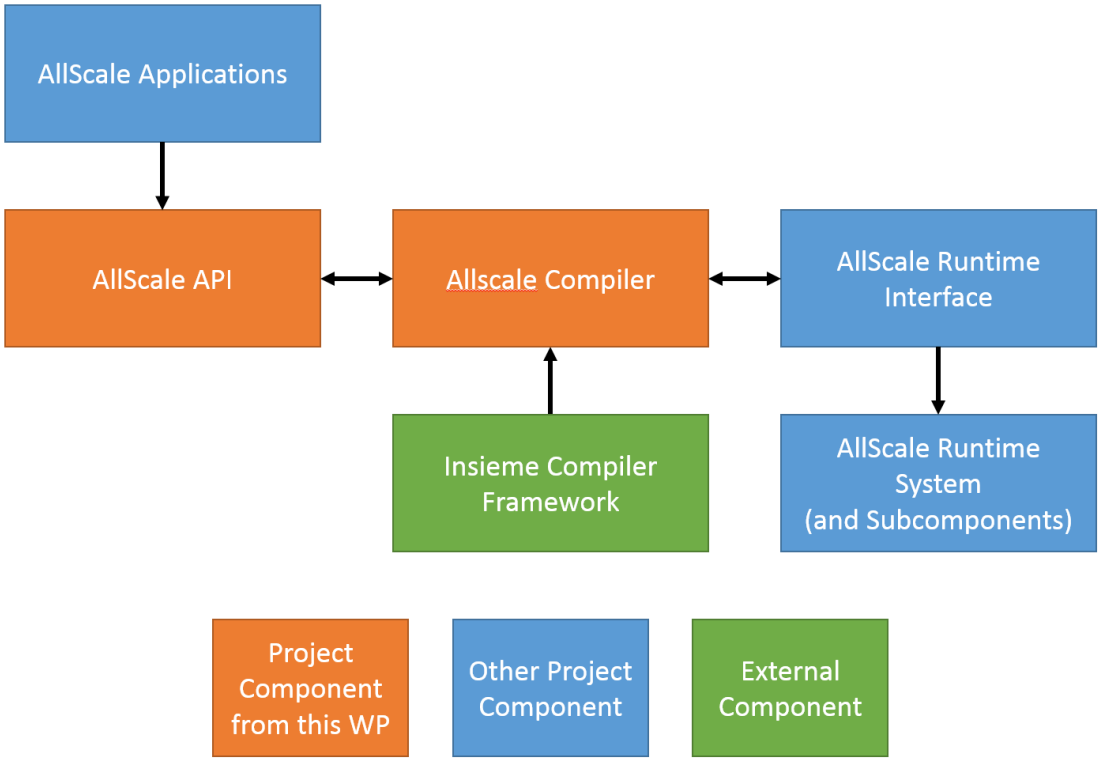

Figure 1: AllScale Compiler Relationship to other Software Components

The major components of this prototype are the intermediate representation of the updated AllScale Core API, its semantics-aware frontend translation, the core analysis determining data requirements, and the backend code generation conforming to the runtime interfaces.

The remainder of this document will provide instructions on how to build the prototype as well as an overview of the implementation.

2 Build instructions

The AllScale Compiler is based on the Insieme research compiler infrastructure, which is an open source project publicly available on github: https://github.com/insieme/insieme

The source code of the Allscale Compiler Prototype is hosted on github at this URL: https://github.com/allscale/allscale_compiler.
Note that the repository uses submodules, so it should be cloned with `git clone --recursive`.

As the full project has a rather large set of dependencies, several installation aids are provided, including a quickstart script, a list of dependencies, and a dependency installation tool. See the README.md file for details.

Once the sources have been obtained either via the Git SCM or a downloadable snapshot, and the dependencies have been fulfilled, the build process is configured using CMake, which interfaces with the native build tools of the given platform. A set of unit tests is included, and it can be built and executed by calling `make test`.

The main compiler driver is `allscalecc`, which can be executed with the same flags as those supported by most C/C++ compilers.

The code is organized as follows (major components only):

- <project root>
  - **api** – a submodule housing the AllScale API
  - **code/compiler** – containing the source for the AllScale Compiler
    - **include** – headers and interfaces
    - **src** – compiler source code
    - **test** – unit tests
    - **resources** – contains resources fur tools included with the compiler, i.e. the web-based reporting tool
  - **insieme** – a submodule for the Insieme compiler infrastructure
  - **runtime** – containing the AllScale Runtime and HPX for testing code generated by the compiler backend
  - **scripts** – containing scripts used for continuous integration and coverage testing, setup and deployment
  - **test** – input files for integration testing
  - README.md – short summary and usage hints

3 Component Overview

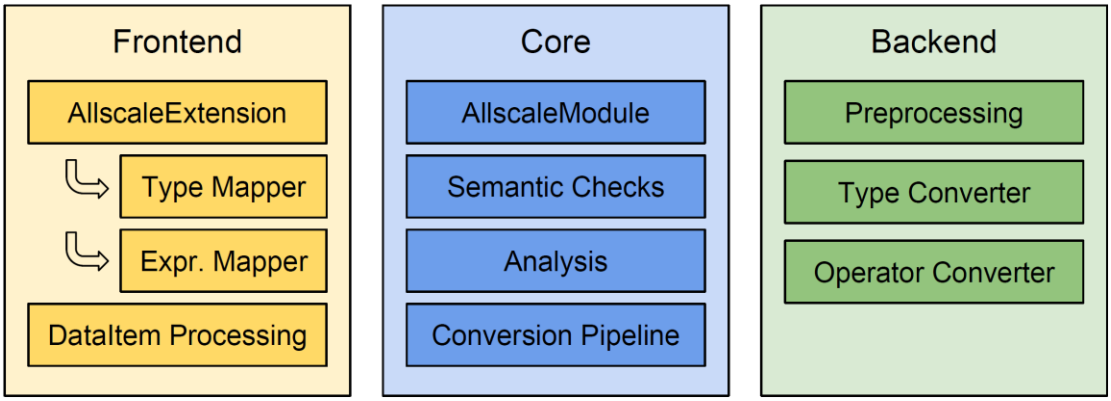

Figure 2: AllScale Compiler Prototype Components

The AllScale Compiler Prototype is split into three distinct parts:

- the Allscale Compiler **Frontend**, tasked with semantic translation of the AllScale core API to the compiler intermediate representation (IR), mapping Allscale-specific types and operators to their IR representation and recognizing data items from their C++ structure.
- the Allscale Compiler **Core**, which provides
  - an intermediate representation module (*AllscaleModule*) of the semantic information relevant for the compilation of AllScale programs
  - a set of semantic checks ensuring the integrity of that representation
  - a sophisticated analysis module determining the data requirements for each parallel region
  - a conversion pipeline which performs AllScale-specific transformations on order to prepare parallel regions for runtime-managed execution
- and the AllScale Compiler **Backend**, which performs all tasks required to generate the program representation expected by the AllScale Runtime System from the intermediate representation of a program.

3.1 The AllScale Compiler Core

As the intermediate language provided by this component is both the target output for the frontend as well as the input for the backend, we will start by providing an overview of this component.

We will use INSPIRE syntax in the descriptions in this section, for reference consider the 2014 thesis by Jordan[1]. The central concepts modeled by the compiler, such as recursive tasks and treetures, are explained in detail in Deliverable D2.4 (AllScale System Architecture). Both of these references are required to fully understand the core compiler functionality.

[1] Insieme - A Compiler Infrastructure for Parallel Programs. UIBK 2014.

The AllScale IR module defines several types central to AllScale, as well as the operations to build and manipulate these types. They are:

- *recfun<'a,'b>*, which describes a recursive task with parameter *'a* and return type *'b*.
- *treeture<'t,'r>*, describing a tree-synchronized future ("treeture") of type *'t* which can be either released (*'r = t*) or unreleased (*'r = f*). The latter indicates a treeture which can receive additional dependencies, while the former is already launched.
- *task_ref*, representing a task reference obtained from subdivision of a treeture. This is only used for synchronization or further subdivision.
- *dependencies*, a type representing opaque user/API-defined fine-grained task dependencies.
- *precfun*, representing the function generated by a prec operator, which can be converted to a lambda callable with or without user-defined dependencies.

The central **prec** operator is then typed as follows:

```
( (recfun<'a,'b>, 'c...) ) -> precfun<'a,'b>
```

In addition to these language definitions and operations (in the *lang* namespace) as well as some auxiliary helper functions, the core module also contains a set of *checks* designed to ensure the semantic integrity of a given IR fragment that makes use of the operators provided by the AllScale IR module. An example of such a check would be ensuring that the function types of all the individual closures passed to the *prec* operator call fit the overall type of the recursive function being constructed.

3.1.1 Conversion Pipeline

Figure 3: AllScale Compiler Core Conversion Pipeline

As illustrated in the figure, the core processing and transformation pipeline of the current AllScale Compiler prototype consists of 7 steps which are specific to AllScale (implemented in addition to any general processing done by the Insieme compiler framework):

1. C++11 lambdas (which manifest as structures with call operators) are lowered to native INSPIRE lambdas, to ease further processing and analysis.
2. Simple global constant propagation is performed, which allows application developers to use more code directly with the AllScale toolchain, and speeds up subsequent compiler steps.
3. AllScale Data Items and the accesses to them are converted to data item references.
4. AllScale Data item references used within lambdas representing parallel regions are captured by value.

5. Accesses calls to data items are optimized, e.g. by hoisting access operations as far as possible outside of loops. (Inter-procedural)
6. Serialization code for use by the AllScale runtime system data management is automatically generated for all user data structures for which this is viable.
7. Finally, work item descriptions are synthesized from the IR, and their data requirement functions are generated from analysis results.

3.1.2 Analysis

The primary responsibility of the Analysis component is producing the data requirement function for each parallel region – or, failing that, a report that indicates the reasons for failure to the programmer.
This module is based on the Insieme *Haskell Analysis Toolkit* (Insieme-HAT), and the additional analyses required for AllScale are implemented in Haskell (code\compiler\src\analysis\allscale-hat). They extend the full-program constraint-based analysis performed by the base toolkit with additional data representations and constraints tailored to the AllScale API and execution environment.
These additional data analyses trace all direct and indirect accesses to *regions* identified as belonging to a data item, and generate a symbolic representation of their union, which is finally encoded in the data requirement function used by the runtime system.

3.2 The AllScale Compiler Frontend

The Frontend of the AllScale compiler is implemented as a `FrontendExtension` within the Insieme infrastructure. This mechanism allows fine-grained control over the translation of C++ types, expressions and declarations to the INSPIRE intermediate representation used in Insieme.

This component is tasked with **semantics-aware translation** of the AllScale API primitives. In practice, this means that the translation process is aware of the meaning of each core API primitive *independent of its specific C++ implementation*. Individual components present in the C++ program are translated to a semantically (but often not syntactically) equivalent representation based on INSPIRE and the AllScale IR module.

As a post-processing step, the frontend identifies types that need to be treated as Data Items, and rewrites their accesses in a format which is usable by the core analysis passes.

3.3 The AllScale Compiler Backend

Similarly to the AllScale Compiler Frontend, the AllScale Compiler Backend makes use of the various mechanisms for extending the base compilation process provided in the Insieme infrastructure.

The intermediate representation generated by the core transformation pipeline is processed by the general backend to generate a C++ abstract syntax tree (AST).

In this process, additional *type handlers* and *operator converters* are installed. E.g. for types, the IR `treeture` type is translated to the AllScale runtime `treeture`, and INSPIRE tuples are translated to the HPX `tuple` type.
During operator conversion, work item spawning as well as operations on treetures are translated to the corresponding runtime system invocations, and IR representations of data item access functions are replaced by calls to the data management interfaces.

4 Usability and Tooling

In order to make the full compiler prototype more usable in real world scenarios, we greatly extended tool support.
The most crucial improvement in terms of usability is the **conversion report**, which provides an interactive interface to per-region conversion results at various levels of granularity and with detail levels that can be adjusted to the familiarity of the user (i.e. optional details for toolchain developers).

Figure 4 shows an example conversion report for a program with a total of 6 recursively parallel regions, which were converted successfully. The first region is unfolded, allowing inspection of the original C++ source code generating it. The full backtrace leading to this code generation can be explored, and the original source can be related to the compiler intermediate representation which generated it, in order to aid toolchain developers.

Figure 4: Conversion and Analysis Report Example Screenshot

Another usability improvement concerns the execution time / speed of the AllScale compiler. Running the complete compilation pipeline, including full program analysis, on the AllScale pilot applications is a large-scale task only made possible by series of performance improvements (both algorithmic and in terms of implementation). These have, in total, resulted in runtime reductions by several orders of magnitude for some algorithms and inputs compared to the initial compiler prototype.

5 Summary and Future Work

The full compiler prototype as of the submission of this deliverable is capable of translating programs using the AllScale API to the interfaces provided by the AllScale Runtime System. It represents all relevant semantics in the INSPIRE intermediate representation, and performs highly sophisticated data dependency analysis on this representation.
Besides the ongoing task of adapting compiler frontend and backend to support future API and runtime functionality, another opportunity for improvement is the quality of reporting to users. Here, the challenge is to ensure that all diagnostics can be traced back to the original program source, without knowledge of toolchain internals.
Another opportunity for extension are additional core transformation and optimization passes, in a similar vein as the currently implemented data access hoisting pass.