Final thesis

Translating C/C++ applications to a task-based representation

by

Lu Li

LIU-IDALITH-EX-A--11/036—SE

2011-10-03
Final Thesis

Translating C/C++ applications to a task-based representation

by

Lu Li

LIU-IDALITH-EX-A--11/036—SE

2011-10-03

Supervisor: Usman Dastgeer
Examiner: Christoph W. Kessler
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Lu Li.
Abstract

GPU-based heterogeneous architectures have been given much attention recently. How to get optimal performance out of those architectures with affordable programming effort remains a complex challenge. The PEPPHER framework is one possible solution. Within the PEPPHER framework, the StarPU run-time system is used to decrease such programming efforts, and at the same time to ensure near optimal performance by efficient scheduling over such architectures. However, adapting a normal C/C++ application to the StarPU runtime system requires additional programming effort.

This thesis implements and tests a composition tool for automatic adaptation of normal C/C++ applications with PEPPHER components to StarPU. This composition tool requires XML annotation for applications and several trivial changes to applications, which take limited efforts. Our results obtained by three test cases (vector scale, sorting, and matrix multiplication) show that automatic adaptation works well on different platforms that StarPU supports. It is also shown that besides StarPU’s dynamic composition, this tool facilitates static composition to improve performance portability of normal C/C++ applications.
Acknowledgments

I would like to thank my supervisor Usman Dastgeer and examiner Christoph Kessler, for their time and advices during the project, the PELAB Fermi maintenance team, for letting me use their machine and helping me getting familiar with CUDA and OpenCL programming environments, and the StarPU development team, for trouble shooting on the StarPU run time system.
Contents

Copyright ............................................................................................................................................................................. - 3 -
Abstract .................................................................................................................................................................................. - 4 -
Acknowledgments ................................................................................................................................................................. - 5 -
1. Introduction ........................................................................................................................................................................... - 8 -
   1.1 Intended Audience ...................................................................................................................................................... - 8 -
   1.2 Purpose and Methodology ........................................................................................................................................... - 8 -
   1.3 Project goal and Requirements ..................................................................................................................................... - 9 -
   1.4 Thesis Outline ............................................................................................................................................................. - 9 -
2. Background ........................................................................................................................................................................... - 10 -
   2.1 Heterogeneous architecture ........................................................................................................................................ - 10 -
      2.1.1 CPU ................................................................................................................................................................. - 10 -
      2.1.2 GPGPU ............................................................................................................................................................ - 10 -
      2.1.3 Performance portability ....................................................................................................................................... - 11 -
   2.2 PEPPHER ....................................................................................................................................................................... - 11 -
      2.2.1 PEPPHER components ........................................................................................................................................ - 11 -
      2.2.2 PEPPHER component meta data ....................................................................................................................... - 12 -
      2.2.3 Staged Composition .............................................................................................................................................. - 13 -
      2.2.4 Composition tool .................................................................................................................................................. - 14 -
      2.2.5 StarPU run-time system ....................................................................................................................................... - 14 -
   2.3 Xerces XML parser .................................................................................................................................................... - 15 -
      2.3.1 XML abstract syntax tree ..................................................................................................................................... - 15 -
   2.4 Intermediate Representation ........................................................................................................................................ - 16 -
3. Implementation ...................................................................................................................................................................... - 17 -
   3.1 Interlayer ..................................................................................................................................................................... - 17 -
   3.2 High level architecture ............................................................................................................................................... - 17 -
   3.3 Workflow of the composition tool ............................................................................................................................. - 19 -
      3.2.1 XML parsing ........................................................................................................................................................ - 19 -
      3.2.2 Intermediate Representation Construction .......................................................................................................... - 21 -
      3.2.3 Static composition ............................................................................................................................................... - 22 -
      3.2.4 Code generation .................................................................................................................................................. - 23 -
   3.4 Error handling ............................................................................................................................................................. - 25 -
   3.5 Verbose Mode ............................................................................................................................................................. - 25 -
4. Evaluation ............................................................................................................................................................................. - 26 -
   4.1 Required modification of the entry component ........................................................................................................... - 26 -
   4.2 Test results ................................................................................................................................................................. - 26 -
      4.2.1 Code generation .................................................................................................................................................. - 26 -
      4.2.2 Compiling and linking the application with the interlayer ................................................................................ - 30 -
      4.2.3 Testing the application on the StarPU platform .................................................................................................. - 31 -
   4.3 Comparison between generated code and hand-written code .................................................................................. - 33 -
   4.4 Programming effort .................................................................................................................................................... - 33 -
   4.5 Performance portability ............................................................................................................................................... - 34 -
   4.6 Design choice motivation .............................................................................................................................................. - 34 -
1. Introduction

With development of computer industry, software applications are growing larger and larger. However, traditional single-CPU computer systems are reaching its speed limit, thus performance requirements from software industry are hard to satisfy continuously. Parallelization is a natural solution for that dilemma. Therefore multi-core systems appear and software is especially designed to utilize these computational resources.

However, a CPU is not efficient enough for all kinds of computations, such as image processing. Therefore, some special purpose processors are designed. GPU (graphics processing unit) is one kind of special purpose processor and commonly used as a coprocessor for a CPU in a normal desktop computer. It can offload computation intensive applications such as image processing so that the CPU is given more space for other operations. But GPU has its own disadvantage too, such that it have large data throughput but can not execute a single task as fast. Thus a homogeneous architecture which consists of only one kind of processors may not be efficient enough for all kinds of computations. For this reason, heterogeneous architectures which consist of processors of different types such as CPU, GPU etc, are given more attentions in recent years. In a heterogeneous architecture, an application could be partitioned into tasks which can be scheduled on most suitable processors in parallel, and thus shows a remarkable decrease in time and power consumption. \[8\]

However, heterogeneity increases programming effort on efficient exploitation of available computational resources of different kinds. For that reason, StarPU is designed to offer a unified execution model for different processor types and dynamic scheduling over those processors which helps applications transparently utilize heterogeneous architectures.

Porting normal C/C++ application to StarPU is not an easy task, which increases programming effort on the run-time system. Programming effort involves a unified task structure initialization including explicit registration of data used by that task, task submission, data unregistration, etc. This thesis implements and tests a composition tool that performs automatic adaptation for normal C/C++ applications to execute on StarPU by adaptation code generation, and also provides easy ways to apply static composition. However, these features require knowledge of some important properties of the application, and fetching these properties through static analysis of C/C++ application might be too complicated for a first step. Thus, our composition tool requires explicit meta data in XML format provided along with the application itself. It also requires some trivial modifications of the application for linking application to adaptation code generated by our composition tool.

1.1 Intended Audience

The reader is assumed to have a basic knowledge of computer architecture, C/C++ programming and design patterns.

1.2 Purpose and Methodology

The purpose of this work is to decrease the programming effort for porting C/C++ application to the StarPU runtime system and provide tool support for static composition.

This is done by implementing and testing a composition tool. Three test applications (Vector scale, matrix
multiplication and sorting) and two examples of static composition are used for evaluation.

### 1.3 Project goal and Requirements

An interface can be considered as a function declaration (either implicit or explicit), and a function that implements the declaration is defined as an implementation of the interface. XML annotations refer to meta data of an application in XML format. Given XML annotations for interfaces and implementations used in a normal C/C++ application, develop a composition tool to convert the application to one which can compile and run with the StarPU run-time system. The semantics of the original application should remain unchanged.

Every implementation and interface within the application is annotated by XML files which conform to a predefined XML schema. The XML schema seldom changes.

A component is a modular part of a system, which encapsulates one or more implementations of the same interface. A component defines its behavior by provided and required interfaces. A component model describes how to write components. The PEPPHER Component model (see Section 2.1.1) is used for writing components in applications to be adapted.

One interface may have multiple implementations. Mapping an interface call to one of its implementation candidates is called a composition choice. Refining the composition choice before program execution is called static composition. In this composition tool’s conversion process, it should be easy to apply static composition.

The application to be adapted can require limited modifications.

This composition tool should run on the Linux operating system.

### 1.4 Thesis Outline

This thesis has the following structure:

A short introduction of concepts used in this thesis will be given in Chapter 2.

Since the main project goal is to develop a composition tool, Chapter 3 discusses the main idea, high level architecture and implementation details.

After that, an evaluation of this composition tool is presented. It consists of test results, and a discussion of decreasing programming effort and increasing performance portability.

A lot of research work has been done in the area of programming models and optimization approaches on heterogeneous architectures. The topic of Chapter 5 is therefore related work.

The limitations of this composition tool and suggestions for future improvements are discussed in Chapter 6.

Conclusions are covered in Chapter 7.
2 Background

2.1 Heterogeneous architecture

A heterogeneous architecture consists of different types of processing units. A processing unit could be a General Purpose Processing Unit such as CPU, or a special-purpose accelerator such as GPU, etc. The demand for increased heterogeneity in computing systems is partially attributed to the need for high performance and power efficient systems. In the past, higher performance was mainly achieved by increasing the speed of a single computational core. However, as the speed of a single core reaches its limit, extra performance is obtained mainly by multi-core systems. Experiments show that different types of computational units are suitable for different types of computations, for example, GPUs show better performance than CPUs with respect to computation intensive applications such as image processing. Depending on the problem to be computed, heterogeneous systems show significant advantage over homogeneous parallel systems on time and power consumption by using the most appropriate cores. [7]

However, programmers have to design platform-specific applications in order to exploit heterogeneous architectures efficiently. This not only requires in-depth knowledge of a specific heterogeneous system (high programming effort), but also makes the application tightly couple with the heterogeneous system (low performance portability), which means performance decreases remarkably if the program is ported to a different heterogeneous system.

2.1.1 CPU

CPU (Central Processing Unit) is a kind of computational device that is most commonly used in computer systems. For a long time, extra performance was gained by increasing its clock rate, however, as it has reached physical limit, current trend is to use multiple CPUs and design programs to execute in parallel.

CPU is mainly designed for general purpose computations, so in some domains, accelerators such as GPUs (see Section 2.1.2) may show better performance than CPUs.

2.1.2 GPGPU

GPU (graphics processing unit) is a specialized computing device developed and traditionally used for processing computer graphics. In a desktop computer, this device often acts as a co-processor to offload image processing from CPUs. A GPU’s highly parallel structure makes it more effective than general-purpose CPUs for many data-parallel algorithms. Unlike CPUs, a GPU’s parallel throughput architecture is more suitable for executing many concurrent threads, rather than executing a single thread.

GPGPU (General-purpose computing on Graphics Processing Unit) is the technique that allows porting computation traditionally executed by a CPU to a GPU, which traditionally handles only computations for computer graphics. GPGPU makes it possible to offload general purpose computations from CPU besides image processing, thus if given such conditions: CPUs and GPUs are combined to a heterogeneous system, both a CUDA implementation and a CPU implementation for a functionality are available, then a scheduler can be designed to map this functionality to an arbitrary execution unit by dynamic composition. With GPGPU, a unified execution model in a heterogeneous system becomes possible.

2.1.2.1 CUDA

CUDA is a parallel computing framework developed by Nvidia. Using CUDA, programmer can easily access Nvidia GPUs.
A kernel is a function executed on special purpose processing unit. Kernel execution in a CUDA program consists of 4 steps. First, the CPU copies data from the main memory to the GPU memory. Second, the CPU sends control signals and the computational kernel to the GPU. Third, the GPU executes in parallel on its cores. Last, the CPU copies result back to the main memory.

2.1.2.2 OpenCL

OpenCL is a framework for writing programs for heterogeneous architectures. It contains a language for writing kernels and an API to control different types of processors. OpenCL provides a unified execution model for different types of processors, such as CPU, GPU and other processors.

This framework is adopted by many major companies from hardware industry, such as AMD/ATI, Nvidia, Intel, Apple, etc.

2.1.3 Performance portability

Performance portability refers to how well an application performs when ported to different architectures. This is an important indicator for evaluating applications running on heterogeneous architectures. Normally heterogeneous architectures differ remarkably from each other, so if an application is ported to a new one without architecture specific optimizations, performance will very likely decrease remarkably and computational resources within the architecture may not be utilized efficiently.

2.2 PEPPHER

PEPPHER [4] is an EU FP7 project which suggests a framework for developing applications on architecturally diverse, heterogeneous many-core processors to ensure performance portability. PEPPHER mainly addresses on how to write code and performance portable applications on heterogeneous systems with affordable programming effort.

The PEPPHER framework, as far as relevant for this thesis project, mainly consists of a component model, a composition layer and a run-time system. The component model suggests how to write PEPPHER components; the composition layer mainly performs adaptation of PEPPHER components for run-time system and static composition; the run-time system handles dynamic composition and scheduling of component invocations during execution of a PEPPHER application.

2.2.1 PEPPHER components

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

- C. Szyperski, ECOOP Workshop WCOP 1997

From this definition, we can see that a component, shown in Figure 2.1a, has an externally visible part: interfaces and dependencies, and an externally invisible part: its implementation(s). Note that a component may but is not required to have multiple implementations for an interface.
An interface declares externally available functionalities. One component may have multiple implementations for the same interface (shown in Figure 2.1b) and in this case composition choices (see Section 2.2.2) should be made either statically or dynamically. An application that calls a component may not know which implementation within the component is finally used and where the implementation is executed. Dependencies of a component indicate which other components or itself are needed in one of its implementations.

In the PEPPHER framework, a component model is designed to specify on how to write components for heterogeneous architectures. We call components in this component model PEPPHER components.

A PEPPHER component contains exactly one interface and multiple implementations. Even multiple implementations on the same platform may co-exist for that interface. To have multiple implementations is crucial for implementation selection as an optimization technique. A PEPPHER component also contains explicit meta data (see Section 2.2.3) of itself to guide composition choice.

### 2.2.2 PEPPHER component meta data

PEPPHER component meta data describes important properties of a PEPPHER component, which guide the composition process.

PEPPHER component meta data should be given in XML format which conforms to a predefined XML schema. There are 2 types of component meta data: interface meta data and implementation meta data.

An interface meta data annotation (interface descriptor) contains some important properties of an interface, such as function signature including parameter types and access modes. An example is shown in Figure 2.2:
The access mode (such as read-only) of each parameter is annotated, so the process of transferring parameters can be more efficient by removing unnecessary memory synchronizations. For example, if a parameter is read-only, then memory synchronization after the function execution can be neglected. Another important property in interface meta data is the relative path information where the implementations of the interface are located. If the relative path is not annotated, then by default the implementations are located in a directory named as the interface name under the root directory of the application. Attribute “numElements” shows the parameter name in the interface which describes the size of the parameter where the attribute is located. Normally when attribute “numElements” is annotated for a parameter, then the parameter is most likely to be of a pointer type to an array or a matrix.

An implementation meta data includes properties relevant to how to compile source code of this implementation, such as compiler name and flag, and which platform this implementation should execute on, such as CUDA, etc. An example is shown in Figure 2.3:

We can see that compiling relevant properties are under the element of tag “peppher:compilation”, and the element of tag “peppher:targetPlatform” shows this implementation should execute on a CPU.

### 2.2.3 Staged Composition

Normally multiple possible callee candidates are available for a caller that invokes a PEPPHER component. A binding from a call to a callee is called a composition choice, and it can be refined in different stages, before or
during execution.

In a heterogeneous architecture, the performance of an application could differ remarkably if composition choice in the application is refined according to available computing devices and type of computation. Static composition refers to refining the composition choice before the execution of a program, while dynamic composition performs the refinement during the execution.

Refining the composition choice statically can be implemented by writing passes. Similar to its counterpart in compiler technology, a pass is a piece of code that traverses an intermediate representation for one time and modifies properties relevant to composition choices in the intermediate representation (see Section 2.4).

### 2.2.4 Composition tool

Component-based applications can be viewed as compositions of reusable and configurable components. A composition tool such as COMPOST, can facilitate such compositions by programmers’ hints. A composition tool always runs at compile time, and after refining composition choices, it can glue these components together by code generation. A composition tool might also transform applications to port them to a specific platform.

### 2.2.5 StarPU run-time system

StarPU [3] is a runtime system for heterogeneous multi-core architectures. StarPU decouples applications from a specific heterogeneous architecture by applying platform specific optimizations to the applications. This decoupling not only frees programmers from the knowledge of complicated hardware architecture, but also makes applications performance portable, that is, when an application is ported to a different heterogeneous architecture, a near optimal performance can still be achieved by dynamic implementation selection and scheduling.

The main features of StarPU consist of a unified execution model, data management, and dynamic scheduling.

StarPU provides a unified execution model for special purpose computing device (such as GPU) as well as general purpose computing device (such as CPU). The unified execution model enables scheduling tasks among heterogeneous processors. To make a computation to be heterogeneously schedulable, a task structure is designed in StarPU to encapsulate all available implementations on different types of processors, and we call it StarPU task. When a StarPU task is scheduled to a new platform, StarPU can automatically perform dynamic composition so that the task becomes code portable to the new platform. Furthermore, if performance prediction data is available, StarPU can automatically pick the fastest and code portable implementation.

StarPU also provides a data management library that performs data transfer automatically between different kinds of processors and accelerators. For efficient management, StarPU requires communication data to be registered in 3 kinds: read-only, write-only and read-write, so that they can be treated in different ways. For read-write data, StarPU transfers these parameters to the processing unit where implementation executes and transfers them back when execution finishes. For read-only parameters, StarPU only transfers these data when execution starts; while for write-only parameters, it only transfers these data when execution finishes.

These two features above enable code portability for applications. Only code portability may not be enough concerning the performance requirements. In order to obtain optimal performance out of a heterogeneous architecture, efficient mapping between tasks and computational units is required. Some run-time factors such as input data size, and run-time environment such as available computational resources, may significantly influence performance of algorithms, which requires dynamic scheduling. StarPU offers the run-time service which targets at optimal performance when porting to a new architecture. Furthermore StarPU allows programmers to give hints or write their own policies for scheduling if necessary.
Applications running on StarPU always start from and end with CPU code, but may run on accelerators during execution.

### 2.3 Xerces XML parser

Xerces-C++ is a validating XML parser written in a portable subset of C++. Xerces-C++ makes it easy to give an external application the ability to read XML data. A shared library is provided for parsing and validating XML documents.

#### 2.3.1 XML abstract syntax tree

In compiler technology, an abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code that is written in a certain programming language.

After successful parsing of an XML file, an abstract syntax tree for that XML file can be constructed for further refinement. An example XML file named hello.xml is shown in Figure 2.4a:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root name="root_node">
  <left name="left_node">
    <deep_left name="deep_left_node">
      Hello world from deep left node
    </deep_left>
  </left>
  <right name="right_node"> Hello world from right node
  </right>
</root>
```

Figure 2.4a An XML file (hello.xml) example

After parsing, an abstract syntax tree is constructed and shown in Figure 2.4b.

The XML abstract syntax tree shows tags and attributes of XML elements and nesting relationship among these elements.
2.4 Intermediate Representation

An intermediate representation (IR), commonly used in compiler technology, is a data structure constructed from compiler input and in turn determines compiler output. Normally an IR is built in a way facilitating traversing. Any optimizations applied on IR will finally affect output.

Using an IR could bring the following benefits:

1) IR eases code generation. Sometimes direct code generation might be too complicated, but building an IR from compiler input and code generation from an IR are much simpler.

2) An IR can be platform-independent and reused for code generation for multiple platforms.

3) IR facilities optimizations. Normally an IR is built in a structured way and offers easy access which optimization techniques desire.
3 Implementation

3.1 Interlayer

Applications running on StarPU always start from and end with CPU code, so the “main” function of a C/C++ application does not require transformations for code portability to other platforms, leaving the main adaptation for its required components. Normally the “main” function does not include heavy computations which mostly reside in its required components, so porting the “main” function for higher performance may not make much sense.

The main idea for adapting components in a normal C/C++ application to the StarPU run time system is to generate an interlayer (shown in Figure 3.1) in each invocation of a component. The interlayer code generation is based on XML meta data provided along with the application.

![Figure 3.1 Interlayer](image)

Left to the dashed line is the traditional way how an application invokes a specific implementation. Right to the dashed line, the interlayer we designed intercepts the call from the application, then adapts it to StarPU format, and last passes it to the StarPU platform. Adapting a call to the StarPU format includes initialization, submission and deallocation of a StarPU task. Initialization of the StarPU task involves binding itself with one or more implementations. StarPU holds freedom and responsibility to determine which implementation is used and where the implementation is executed at run time.

3.2 High level architecture

Besides the requirement for adapting normal C/C++ applications, another important requirement is that this composition tool should facilitate optimizations such as static composition. Therefore we design an intermediate representation, an optimizable form which determines code generation.

Considering these requirements, we design a workflow of our composition tool as described in Figure 3.2.
First, the XML parser will convert XML annotation files to XML abstract syntax trees, and then IR is built upon these trees. Static composition is an optional module and is responsible for refinement of composition choice. Last, adaptation code is generated. After running the composition tool, the application should be able to directly compile and run on StarPU. More details of this work flow are described in subsequent subsections.

This composition tool serves as the composition layer in the PEPPHER framework. First, users write an application with PEPPHER components, then the composition layer makes composition choices of each PEPPHER components if necessary and adapts the application to the PEPPHER run time system which is presently StarPU, and last the transformed application executes upon the run time system.
3.3 Workflow of the composition tool

3.2.1 XML parsing

The XML parsing module validates all XML annotation files in an application and converts them to abstract syntax trees.

In this project Xerces-C++ is used for XML parsing. Xerces-C++ accepts an XML file and returns XML elements of the XML file in nesting order. We design class Element to encapsulate the attributes and content of XML elements from Xerces-C++. Class Element contains pointer to its own type (Composite pattern) by which abstract syntax tree is formed.

One important issue is that from the parameters given to the composition tool only one XML file is available, however, in order to generate adaptation code for all PEPPHER components in an application, all XML annotation files for both implementations and interfaces should be parsed. Therefore every XML file being parsed should give clues to where further needed XML files are located if there is any.

The only available XML file from the parameters given to the composition tool is the XML annotation file for the entry component. Here we define the entry component as the application’s entry function, normally the “main” function for a C/C++ application. The entry component may require some other components whose implementations and interfaces are annotated by one XML file for each. The set containing exactly one interface, all implementations of that interface and their annotation files is a PEPPHER component (for distinction from entry component) and we call it a component for short.

The differences between entry component and component are as follows: an entry component contains only one implementation without interface while a component has exactly one interface and usually multiple implementations. An entry component normally contains only the “main” function and other non-componentized CPU code so an interface is not necessary. An adapted application always runs at start on a default CPU so the only implementation within its entry component has a fixed execution location, namely the CPU. At least one implementation executing on a CPU is suggested to be contained in each component.

The process for parsing all XML files is shown in Figure 3.3.
Figure 3.3 Parsing process

The parsing process is performed as follows: First, we parse the entry component’s XML given as a parameter of the composition tool. Next, XML files for its required components should be parsed. The first XML file contains information about all components that the entry component requires and these component’s interfaces’ XML files are always located at the root directory of the application, so in the next step these XML files are parsed. An interface’s annotation file contains relative path information where all the implementations for that interface are located. Last those implementation annotation files are parsed. If a component A calls another component B, then parsing continues recursively in the same way for components as described above until no component is depended on. We maintain a list of components parsed so repeated parsing for the same component can be avoided.
3.2.2 Intermediate Representation Construction

The intermediate representation we use here is a set of classes which are built from the abstract syntax trees by the XML parsing module and act as input for code generation. The intermediate representation decouples XML files from code generation, so optimizations could be applied on the intermediate representation which will finally affect the code generation.

Entry component and component are two main kinds of intermediate representation, respectively shown in Figure 3.4 and 3.5.

![Figure 3.4 Entry component](image)

![Figure 3.5 Component](image)

An entry component contains only one implementation and no interface, while a component contains a set of implementations, an interface and a configuration. It is possible to have multiple implementations for each platform in a component so the composition choice of each component is to be determined statically or dynamically. The configuration stores this composition choice, showing which implementation of each platform should be composed by calling components. The implementation that should be composed at composition time is called the default implementation. Currently at most 3 default implementations can be specified in the configuration: one for each platform, i.e. CPU, CUDA and OpenCL. The configuration can be set so that the first added implementation will automatically become the default implementation; otherwise writing passes is needed to manually set the default implementations.

The internal structure of interface and implementation as an IR is shown respectively in figure 3.6 and 3.7.
Interface contains a set of Methods and XML Files. The set of Methods stores all methods of the interface. The set of XML files contain annotation file names of all implementations belonging to the interface used for further parsing.

Implementation contains a set of Compile Statements, an Implementation Type, a Codelet information and a set of Dependencies. The set of Compile Statements contains all compile statements computed by relevant elements of the abstract syntax tree and used for makefile generation. Implementation Type shows which platform the implementation should execute on, such as CPU, CUDA or OpenCL. Codelet information stores OpenCL codelet information and is only applicable if Implementation Type is OpenCL, since StarPU requires OpenCL's computational kernel to be located in a separate codelet file. The set of Dependencies contains names of XML files of all interfaces this implementation requires. An important detail is that Dependency contains a pointer to the corresponding component object by which the entry component and all components are connected as a graph structure.

3.2.3 Static composition

Similar to a compiler framework, various optimizations can be applied on the intermediate representation and finally affect code generation. In this composition tool, optimizations mainly refer to refining composition choices. Since the composition tool performs these optimizations before the applications runs, we call refining composition choices at this stage static composition. Here we discuss two examples of static composition, namely to disable a platform and to change a default implementation.

To disable a platform of a component means that after running the composition tool, no implementation on this platform within the component will be registered on the StarPU task, so at run-time StarPU is not able to find a code portability implementation on that platform and thus the scheduler in StarPU will not consider this platform as a valid option. Although a component may already contain a default implementation for a specific platform in early stages, one can set the default implementation of that platform to be null, thus adaptation code relevant to this platform of a component will not be generated in the adaptation code. This can be done by writing a pass to remove relevant values...
in the component configuration object.

To change a default implementation of a component means changing the composition choice of the component. A component may already contain a default implementation for a specific platform, and one may change the default implementation to another of the same platform type, thus adaptation code will contain the new implementation instead of the old one. Finally, if StarPU schedules on this platform, it will use the new implementation. This can be done by writing a pass to replace relevant values of the component configuration object.

Each composition is applied by writing a pass. If several passes happened to change the same value of the intermediate representation, then only change by the pass executed latest will take effect in the code generation.

StarPU can refine the composition choice during run-time, which may be classified as dynamic composition. Static composition has its own advantage compared to dynamic composition. Currently dynamic composition mostly depends on an empirical model because a heterogeneous architecture may be too complex for an analytical model. An empirical model has the disadvantage that it can not guarantee precision of composition choices. Although automatic and precise analysis for heterogeneous architecture may not be presently possible, a programmer with knowledge of the internal structure of both application and run-time environment may be able to give accurate composition recommendations. In this case static composition shows its advantage. With composition tool support, programmers can easily apply these static compositions.

### 3.2.4 Code generation

Three kinds of files are generated for normal C/C++ application to compile and execute on StarPU platform: a main header file, a set of wrapper header files, and a makefile. Example codes and relevant discussions are given in Chapter 4.

A main header file includes all wrapper header files (discussed in next paragraph) generated. Including the main header file in entry component gives it access to its all required components by calling interfaces of these components. As discussed in Section 3.2.2, the entry component and its dependencies form a graph structure, which we call component dependency graph, an example is shown in Figure 3.8. The entry component is the entry of component dependency graph, thus it can serve as a Facade to interact with the graph. The main header file is built by traversing the whole graph and includes all wrapper header files.

![Component dependency graph](image)

If an implementation depends on a component, it will contain a reference to the component without knowing
which implementation is finally used and where the implementation is executed. A component contains an interface, a set of implementations, and a configuration that shows the default implementation for each platform.

A wrapper header file is generated for each component, except for the entry component. By traversing one component within the intermediate representation, all necessary information is collected, and then a header file containing wrapper functions is generated. The internal structure of a typical wrapper header file is shown in Figure 3.9.

A wrapper header file contains four wrapper functions, shown in Figure 3.9: the main wrapper, the CPU wrapper, the CUDA wrapper and the OpenCL wrapper. The main wrapper performs initialization of a StarPU task for the called function, including registering interface parameters and the other three implementation wrappers (such as the CPU wrapper) to the StarPU task, then submits it to StarPU, and last unregisters parameters from the StarPU task. When the StarPU task finishes its execution, it will be automatically destroyed. But explicit unregistering of data from the StarPU task is required. A CPU wrapper converts parameters from StarPU format to normal C/C++ format, and then passes these converted parameters to the CPU default implementation. A CUDA wrapper is similar to a CPU wrapper. An OpenCL wrapper handles more things besides a CPU wrapper, such as loading a codelet because StarPU requires an OpenCL computational kernel to be written in a separate file and be loaded at run time.

As seen from Figure 3.9, an implementation wrapper actually contains its corresponding default implementation. When the main wrapper initializes a StarPU task, it binds the StarPU task with the three implementation wrappers. In other words, the StarPU task binds three default implementations which can run on CPU, CUDA and OpenCL respectively. These bindings allow StarPU to perform run-time implementation selection among the three implementation candidates and schedule one on its corresponding platform according to a scheduling policy.

The main wrapper is generated regardless of the component configuration; however, CPU, CUDA and OpenCL wrappers are optional. For example, if CUDA is disabled in component configuration, CUDA wrapper will not be generated in adaptation code. It is suggested that at least one platform should be enabled, otherwise no implementation is available which is similar to the scenario that a function declaration exists but some function body for this declaration can not be found.

Makefile generation is similar to the main header file generation, and only one makefile is generated for the whole application, by traversing whole component dependency graph to collect relevant information and computing a set of values. Only default implementations will be involved in the makefile, so static composition will affect makefile generation as well. One detail is that after initializing IR from abstract syntax trees, a linking statement will be computed and inserted to the IR. The linking statement refers to the linking command in the compilation process.
which links all object files into an executable.

### 3.4 Error handling

Validating XML files against XML Schema is handled by Xerces-C++.

There are 3 levels of error handling for later stages: fatal error, error and warning.

Fatal error means global errors that will cause software failure. When one fatal error is detected, the composition tool terminates and an error message is shown. An example of a fatal error could be “XML schema not found”, caused by some path not being set correctly.

Error means local errors that are not legal according to the PEPPHER component model and current restrictions, which cause incorrect code generation. When one error is detected, the composition tool terminates and an error message is shown. An example of an error could be that a platform specification in implementation annotation file differs from the predefined CPU, CUDA or OpenCL.

Warning handles valid but dangerous actions such as overwriting some critical values. When a warning is detected, the composition tool continues and a warning message is displayed.

### 3.5 Verbose Mode

Except for syntactic errors, others hidden in XML files are hard to find manually. Verbose mode is useful for debugging XML annotation files of C/C++ applications. The intermediate representation is constructed by the output of parsing of XML annotation files, so if some problems hide in XML files, inspecting the intermediate representation might provide valuable clues. Verbose mode is mainly used to print information about the intermediate representation.

Three levels of verbose mode are designed to show information in a more and more detailed way. The first level shows the component dependency graph. The second level shows details of each component and of the entry component besides the first level. The third level also shows computation results which are based on the intermediate representation and used for adaptation code generation.
4 Evaluation

We have developed three test cases: vector scale, matrix multiplication and sorting. Here we discuss the vector scale example in detail, because other two test cases are similar to the vector scale. Vector scale is a simple application that scales a vector. It contains a “main” function which calls only one PEPPHER component whose interface name is vector_scal and implementations execute on CPU, CUDA and OpenCL platforms.

4.1 Required modification of the entry component.

For an application only three things needs to be modified in the entry component manually by programmers in order to utilize the interlayer: first, include the main header file generated; second, replace a traditional call to a specific function implementation (such as a CUDA implementation) with a call to the interface name; and last, add macros PEPPHER_INITIALIZE at the beginning and PEPPHER_SHUTDOWN at the exit of the “main” function. These modifications of the application are rather trivial and do not require any knowledge of StarPU.

An example of vector scale code comparison between original and modified application is shown in Figure 4.1:

![Figure 4.1 A comparison between original and modified vector scale application](image)

Macros PEPPHER_INITIALIZE and PEPPHER_SHUTDOWN are defined in the main header file, while the main wrapper vector_scal() is generated in the wrapper header file included by the main header file. File inclusion of the main header file gives the “main” function full access to its required components in which the composition choices are made at compile time or run-time.

4.2 Test results

4.2.1 Code generation

File lists of the vector scale example before and after running the composition tool is shown in Figure 4.2:
Before running the composition tool, the file list contains 2 parts: the entry component (main.c) and its annotation file (main.xml), the vector scale component that consists of implementations and their annotation files (vector_scal folder), and interface annotation file (vector_scal.xml). The implementation folder contains implementation source files and their annotation files: a CPU implementation (vector_scal_cpu.c) and its annotation file (vector_scal_cpu.xml), a CUDA implementation (vector_scal_cuda.cu) and its annotation file (vector_scal_cuda.xml), an OpenCL implementation (vector_scal_opencl.c and vector_scal_opencl_codelet.cl) and its annotation file (vector_scal_cpu.xml). StarPU requires the OpenCL computation kernel to be written in a separate codelet file, in this case is vector_scal_opencl_codelet.cl.

After running composition tool, three files are generated: a main header file (peppher.h), a wrapper header file (vector_scal_wrapper.h) and a makefile. The interlayer consists of the main header file and the wrapper header file.

The code of the generated main header file (peppher.h) is shown in Figure 4.3:

```
#ifndef PEPPHER_H
#define PEPPHER_H

#include <starpu.h>
#include <starpu_opencl.h>

#define USE_STARPU

#ifndef USE_STARPU
#define PEPPHER_INITIALIZED
#else
#define PEPPHER_INITIALIZED

#include "starpu_init.h"

PEPPHER_INITIALIZED
else

PEPPHER_INITIALIZED

#endif
#endif

#endif
```

Macros PEPPHER_INITIALIZIZE and PEPPHER_SHUTDOWN are defined to initialize and shutdown the
StarPU run-time system. The main header file also includes all wrapper header files generated, in this case only one wrapper header file, namely “vector_scal_wrapper.h”.

The wrapper header file contains a more complex structure, here we discuss it in segments. First, the variable definition part of the wrapper header file is shown in Figure 4.4:

```c
struct readOnlyArgs_vector_scal
{
    float factor;
};

typedef struct readOnlyArgs_vector_scal ROA_vector_scal;
starpu_codelet cl_vector_scal;
ROA_vector_scal arg;
```

Figure 4.4 The variable definition part of the wrapper header file example

Structure readOnlyArgs_vector skal encapsulates all read-only parameters of the vector skal interface. Read-only parameters are registered to a StarPU task so that StarPU will not write those parameters back from the device where implementation executes on, to the device where implementation is called when implementation finishes execution.

Second, the main wrapper code is shown in Figure 4.5:

```c
void vector skal ( float * arr, unsigned size, float factor )
{
    arg.factor=factor;

d_vector skal where =0(STARPU_CPU|STARPU_CUDA|STARPU_OpenCL;
cl_vector skal cpu_func=scalar_cpu_func_WRAPPER;
cl_vector skal cuda_func=scalar_cuda_func_wrapper;
cl_vector skal opencl_func=scalar_opencl_func_wrapper;
cl_vector skal n_buffers = 1;

int i = starpu_opencl_load_opendcl_from_file( 
    "vector skal\vector skal opencl\codelet_vector skal.c", 
&codelet_vector skal, ");
starpu_data_handle arr_handle;
starpu_data_data_register( & arr_handle, 0, (uintptr_t)arr, size, sizeof(arr[0]));

struct starpu_task *task = starpu_task_create();
task->synchnous = 1;
task->cl = &cl_vector skal;
task->buffers[0].handle =arr_handle;task->buffers[0].mode = STARPU_RW;
task->cl_arg = &arg;
task->cl_arg_size = sizeof( ROA_vector skal );

int ret = starpu_task_submit(task);

if (STARPU_UNLIKELY(ret == -EODEV))
{
    fprintf(stderr, "No worker may execute this task\n");
    return;
}

starpu_data_unregister(arr_handle);
starpu_opencl_unload_opendcl(&codelet_vector skal);
}
```

Figure 4.5 The main wrapper code example
The main wrapper first initializes a codelet structure (cl_vector skal) which encapsulates information about all default implementations for different platforms, then creates a StarPU task and binds it with the codelet. It also registers parameters on the task. So far we have a StarPU task which knows all implementation variants and the way to transfer parameters if one of these implementations is called. In other word, this task encapsulates all necessary information for dynamic composition and scheduling by StarPU, and then we submit it to StarPU and let it perform these run time services. When a chosen implementation finishes execution, StarPU will automatically transfer parameters that are not read-only back to the processor where the caller is located. Last, we unregister parameters from that StarPU task.

Third, the CPU wrapper code is shown in Figure 4.6:

```c
extern void scal_cpu_func(float * arr, unsigned size, float factor);

void scal_cpu_func_wrapper(void *buffers[], void *args)
{
    scal_cpu_func(
        (float *)STARPU_VECTOR_GET_PTR((starpu_vector_interface_1 *)buffers[0]),
        STARPU_VECTOR_GET_NX((starpu_vector_interface_1 *)buffers[0]),
        ((ROA_vector skal *)_args)->factor
    );
}
```

Figure 4.6 The CPU wrapper code example

CPU wrapper scal_cpu_func_wrapper () converts parameters managed by StarPU to normal C/C++ parameters, and passes them to the CPU default implementation which is a normal C/C++ function. StarPU is responsible for parameter transfer between different processing units, so converting parameters from StarPU format to normal C/C++ format involves interacting with the StarPU data interface. Parameters are converted in 2 ways: read-only parameters can be extracted directly from the structure ROA_vector skal (Redefinition of readOnlyArgs_vector skal), and other parameters are extracted from the StarPU data interface by StarPU APIs.

Fourth, the CUDA wrapper, which is similar to CPU wrapper, is shown in Figure 4.7:

```c
extern void scal_cuda_func(float * arr, unsigned size, float factor);

void scal_cuda_func_wrapper(void *buffers[], void *args)
{
    scal_cuda_func(
        (float *)STARPU_VECTOR_GET_PTR((starpu_vector_interface_1 *)buffers[0]),
        STARPU_VECTOR_GET_NX((starpu_vector_interface_1 *)buffers[0]),
        ((ROA_vector skal *)_args)->factor
    );
}
```

Figure 4.7 The CUDA wrapper example

Last, the OpenCL wrapper is shown in Figure 4.8:
Besides functionality similar to the CPU wrapper, the OpenCL wrapper performs extra works required by the OpenCL execution model, such as loading the OpenCL codelet file, since StarPU requires an OpenCL computation kernel to be written in a separate file of codelet type.

### 4.2.2 Compiling and linking the application with the interlayer

Makefile generation enables the application to directly compile with the StarPU run-time system. The generated makefile for the vector scale application is shown in Figure 4.9:

```plaintext
CFLAGS += $(shell pkg-config --cflags libstarpu)
LDFLAGS += $(shell pkg-config --libs libstarpu)
CC = g++

main: main.o vector_scale_cpu.o vector_scale_cuda.o vector_scale_OpenCL.o
g++ main.o vector_scale_cpu.o vector_scale_cuda.o vector_scale_OpenCL.o
$(CFLAGS) $(LDFLAGS) -o main
main.o: main.c $(CFLAGS) -c -o main.o
vector_scale_cpu.o: vector_scale VECTOR_scale_cpu.c
$(CFLAGS) -c -o vector_scale_cpu.o
vector_scale_cuda.o: vector_scale VECTOR_scale_cuda.c
$(CFLAGS) -c -o vector_scale_cuda.o
vector_scale_OpenCL.o: vector_scale VECTOR_scale_OpenCL.c
$(CFLAGS) -c -o vector_scale_OpenCL.o

%.o: %.c
  $(CC) -c $(CFLAGS) $< -o $@

vector_scale.o: vector_scale VECTOR_scale.c
  $(CC) -c $(CFLAGS) $< -o vector_scale.o

vector_scale_CUDA.o: vector_scale VECTOR_scale_CUDA.c
  $(CC) -c $(CFLAGS) $< -o vector_scale_CUDA.o

vector_scale_OpenCL.o: vector_scale VECTOR_scale_OpenCL.c
  $(CC) -c $(CFLAGS) $< -o vector_scale_OpenCL.o

clean:
  rm -f main *.o
```

Figure 4.9 The makefile example

The second rule is for compiling the entry component to an object file. And the rules below it compile each default implementation to an object file. The first rule links all object files together to an executable.

After modification of our test application discussed in Section 4.1 and running this composition tool, this application can be compiled and linked using "make". The compilation process is shown in Figure 4.10:
First the entry component is compiled to an object file, and then the default CPU implementation, the default CUDA implementation, and the default OpenCL implementation are compiled to object files. Last these object files are linked to an executable. Note that the StarPU library is used in the compilation.

### 4.2.3 Testing the application on the StarPU platform

Last we can test if the executable succeeds in running on the StarPU run-time system. Test results are shown below in segments.

A test for a specific platform is performed by specifying the only available platform for StarPU scheduler to be the platform, thus StarPU is forced to schedule code portable implementation on the platform. The test result for CPU is shown in Figure 4.11:

![Figure 4.11 Test result for the vector scale application running on CPU platform](image)

Before vector scale component execution, all elements in a vector is 1.0 and after, all elements is scaled to 3.5 which shows StarPU successfully transfers data between execution units, perform dynamic composition and scheduling on CPU platform. Similar tests are applied on CUDA and OpenCL. Test result for CUDA and OpenCL is shown respectively in Figure 4.12 and 4.13:

![Figure 4.12 Test result for the vector scale application running on CUDA platform](image)
Figure 4.13 Test result for the vector scale application running on OpenCL platform

```
BEFORE : First element was 1.000000
        Last element is 1.000000

```

```
*********** Hello in vector-scale OPENCL call ***********

AFTER First element is 3.500000
     Last element is 3.500000
```
4.3 Comparison between generated code and hand-written code

Comparing generated code and hand-written code is valuable in that it shows the advantages and disadvantages of automatic code generation by this composition tool.

Generated code and hand-written code for adapting C/C++ application to StarPU are the same except for the case that a component is called inside a loop. An example for comparing such difference is displayed in Figure 4.14:

If in an application one component is called in a loop, in generated code, the StarPU task in the main wrapper is initialized and destroyed multiple times unnecessarily; in hand-written code this can be done in a smart way, by initializing task before the loop and destroying it after the loop. Thus, in cases of a loop, hand-written code gains more performance than generated code.

However, hand-written code takes much higher porting cost.

4.4 Programming effort

With this composition tool support, we expect to decrease programming effort for adapting normal C/C++ application to the StarPU platform.

Writing XML files is simple especially when well-tested examples of XML annotation files can be reused. However, if this task is performed manually, it might be a complicated task. It requires the user to learn execution model and programming interface of StarPU, design and implement wrapper correctly. The learning process might be time-consuming. Furthermore, if a run-time error is found in wrapper code, it may take hours to fix since debugging code in a heterogeneous architecture is far more complex than a single-CPU platform where XML file debugging takes place. With verbose mode, XML debugging may be accelerated. Last, writing the makefile is error-prone.
because paths and library are tricky to handle. Writing XML files might decrease the effort of editing makefile because libraries and paths are handled automatically.

### 4.5 Performance portability

If we succeed in adapting normal C/C++ applications to the StarPU platform, it is safe to conclude that performance portability is increased because StarPU can perform dynamic scheduling: if the application is ported to a new heterogeneous architecture, StarPU can automatically analyze for available resources and schedule the application tasks to most suitable processors or accelerators. Although run-time composition and scheduling take some overheads, StarPU has shown that if given a proper scheduling, obtaining a consistent superlinear speedup by StarPU is possible [3]. Thus, near-optimal performance might be achieved by StarPU’s run-time services whatever the hardware architecture is.

Besides the composition carried out at run-time, static composition can also be applied to further improve performance portability. For example, when porting an application to new architectures, valuable hints given by a programmer with internal knowledge of both application and architecture or by a performance model can help to specify or remove certain composition choices. If the best implementation is chosen at this stage, extra performance is gained [7]. Furthermore, if statically we can choose the best implementation, run-time overhead for implementation selection and scheduling can be removed. Ideally, in this case, creating a new task for StarPU is unnecessary, so computation can be directly invoked on a chosen platform. In practice, this is not possible yet. Even if the best implementation couldn’t be decided statically, removing some composition choices will decrease run-time implementation selection time since fewer entries should be inspected. Static selection can be done easily by composition tool support. So porting cost is low and extra performance is gained with respect to a new architecture by static composition.

It is important to notice that StarPU does not support multiple implementations for a single platform type which would be necessary to support implementation selection in a more general sense. This composition tool is designed to support this limitation of StarPU, thus it is a necessary supplement for StarPU.

### 4.6 Design choice motivation

This composition tool decouples a call from a specific implementation. This choice facilitates both static and dynamic selection.

This composition tool decouples XML annotations with adaptation code generation by inserting an IR between the two. This choice facilitates static composition. Furthermore, if a new processor or accelerator besides CPU, CUDA and OpenCL is supported by StarPU in the future, for the composition tool, only code generation module is required to be extended and other modules, such as XML parsing, IR construction, and static composition can be reused.

Xerces-C++ (see Section 2.3) is chosen for parsing and validating XML files, because of its high usability and extensibility. Another reason for this choice is that we have experiences of Xerces-C++ in previous projects. Several alternatives exist such as libxml2.

The entry component serves as a Façade in the intermediate representation. This allows easy access to a complex component dependency graph.

Code generation involves calculating strings. String calculation classes we designed for this purpose are initialized by the Factory method, so it is easy to extend this composition tool for code generation. When some new functionality needs to be generated for the wrappers, new classes for string calculation should be added which
requires that only the Factory class should be modified. One complicated problem in the implementation of this composition tool is string calculation based on XML information. For some strings, it requires a complex calculation process. The Composite pattern is used for the design of complex string calculation classes, so one can call another, and so on, to reduce complexity recursively or step by step.

It is easy to apply static composition, and even to manage these compositions. Each composition is uniform as a pass towards the intermediate representation.
5 Related works

Heterogeneous architecture has gained much attention in recent years, and a lot of research and work has been made around this topic. This composition tool is involved in its three sub-problems: programming model, implementation selection, and translation. Comparisons are discussed around these sub-problems.

5.1 Programming model

Presently several programming approaches has been developed to ease utilization of heterogeneous computational resources.

CUDA [9] provides APIs for transferring data and code between CPUs and Nvidia GPUs, thus programming on such devices becomes easier. CnC-CUDA [5] suggests a high level declarative and implicit parallel coordination language. Programmers write CPU steps and GPU steps, leaving CnC-CUDA framework for task management and scheduling over heterogeneous computational resources.

OpenCL [10] offers a new language for writing kernels which are functions executing on OpenCL devices and a set of APIs to define and control these platforms. OpenCL can also enable non-graphical computing to execute on graphics processing unit.

PetaBricks [2] suggests a new implicit parallel language which includes constructs to specify multiple algorithmic choices for the same functionality. Programmers write transforms (analogous to functions) and rules (how to make progress for a given task). PetaBricks compiler and autotuner transform the application to C++ and eliminate unused choices.


Kessler and Löwe [6] suggest a framework for writing performance-aware components. Component provider, who knows implementation details best, is required to extend traditional components by writing performance-related meta-data and -code, for example, a time function for performance prediction.

The composition tool presented in this thesis requires component provider to write meta data in XML format. Implementation might be written in CUDA, or OpenCL. The application needs several trivial modifications.

CUDA and CnC-CUDA addresses only CUDA devices, so porting cost to other kinds of processors might be high. OpenCL provides a unified framework for a variety of processors and accelerators which addresses code portability, leaving performance portability a complicated challenge. Comparing with PetaBricks and performance aware components, the approach of the composition tool presented in this thesis minimizes modification of the source code of the application and requires no modification of components which facilitates reusing existing components, and its black-box composition shows applicability if only binaries of components are available, furthermore, meta data in XML format is easy for extension. Elastic computing allows reuse of existing applications, but elastic components are restricted to device experts which limits its flexibility and extensibility.

5.2 Implementation selection

In the PetaBricks framework, a compiler performs static analysis of source code in the PetaBricks language, and
encodes algorithmic choices and tunable parameters in the output code. An autotuning system and choice framework must find optimal choices and parameters upon output of PetaBricks compiler at compile time or installation time and generate a configuration file. This configuration file can be tweaked manually to force composition choices. Finally, a run-time system schedules tasks across processors.

In the elastic computing framework, elastic functions specify multiple implementation choices of the same functionality. By implementation planning, thousands of new implementations and their performance prediction data are generated for different situations. The elastic computing system serves as run-time system and performs implementation selection based on performance prediction data obtained by an analytical model.

In the performance-aware component framework, performance prediction data are computed off-line based on meta-code and -data provided by the component provider. A composition tool then injects into each component brokering code which looks up performance prediction data at run-time and performs implementation selection.

The composition tool presented in this thesis performs static composition by hint. This approach shares similarity with PetaBricks compiler in that the configuration file can be tweaked by hand statically. The implementation selection of elastic computing depends on an analytical model for performance prediction which requires less overhead for calibration, but the performance prediction might not be as accurate as those based on empirical models in accuracy since run-time factors may show a significant influence. The approach of performance-aware components gives component providers flexible ways for specifying performance prediction calculation, either by an analytical model or an empirical model.

5.3 Transformation

Transformation here is defined as how an application changes after a certain processing.

PetaBricks compiler invokes a source-to-source translation on an application from PetaBricks language to C++, and then an autotuning system and choice framework makes algorithmic choices and sets autotuning parameters by generating a configuration file which can be fed back to the PetaBricks compiler for elimination of unused choices. Finally an application with multiple implementation choices is transformed to one with only optimal implementation choices.

Elastic computing requires no modification of the user application, so no transformation is needed.

Performance-aware components are processed and injected brokering code by a composition tool at compile time.

CORBA [1] provides a component model that enable language transparency, it requires an interface described by Interface Definition Language, from which glue code such as stubs and skeletons are generated. The glue code glues non-fitting components together for language adaptation.

The composition tool presented in this thesis transforms an application with XML annotations statically by generating an interlayer for each invocation of a component, and the application requires some modification of application and keeps components unchanged. This approach shares similarity with CORBA, interface annotations serves as IDL and the interlayer generated acts as a stub for a client application.
6 Limitations and future work

6.1 Limitations

Currently, the depth of the component dependency graph should not exceed one which also implies that it is a flat tree. This limitation is primarily caused by StarPU because currently it does not support to submit a task within a task.

One interface can not contain multiple methods. This limitation is primarily caused by the current schema specification in which the interface name is the method name. One interface can only have one name which means only one method is allowed in one interface. However, the composition tool supports multiple methods in one interface, so in the future, if the schema is modified to support the limitation, the composition tool takes only limited effort to change.

When a template is used in a CUDA implementation of a component, an OpenCL implementation can not co-exist in that component. This limitation is caused by compiler conflicts. When a template is used in a CUDA implementation, the caller must include the implementation. Then the caller has to be renamed to end with ‘.cu’ because it contains CUDA code. Also the caller has to be compiled with the nvcc compiler. However, nvcc can not compile OpenCL code, which causes compiler conflicts, so a CUDA implementation with template and an OpenCL implementation are exclusive to each other. This limitation is accepted.

6.2 Future work

The composition tool presented in this thesis requires XML annotation files for the application. Although writing XML takes limited effort especially when well-tested XML files can be reused, it might be possible to automatically generate such XML files by static analysis of the application or other techniques. This feature, if successfully implemented, will further reduce programming effort.

The error handling mechanism can not continue to detect errors when one is found. This makes it not convenient enough to debug XML files. Techniques from compiler technology might be used to improve error handling.

In section 4.4, we discuss the difference between manually written code and generated code. Manually written code is better when a component is called in a loop. This can be improved by storing the task structure in a global structure so a task does not need to be initialized and unregistered multiple times during loops.

Static composition by the composition tool can be combined with a performance model. In StarPU there are built-in performance models, however, StarPU does not support multiple implementations of the same platform type which makes StarPU not general enough. In order to aid implementation selection in a more general way, an independent module for building and looking up a performance model based on training execution can be added to the composition tool so static composition can get better precision.

Required modification of application discussed in Section 4.1 may be automated by invasive software techniques.

A quantitative study for comparing programming effort between writing generated code by this composition tool manually and writing XML annotations for an application and its required components can be performed to verify that programming efforts decreases by employing this composition tool.
7 Conclusions

The StarPU run-time system successfully decreases programming effort while increasing performance portability of applications on heterogeneous architectures. We present a composition tool for automatic adaptation for normal C/C++ applications with XML annotations to exploit StarPU with limited effort. It also provides facilities for static composition.

Automatic adaptation might decrease programming effort to adapt normal C/C++ applications to StarPU, which means possibility for porting a tremendous amount of existing applications to heterogeneous architectures, so the performance of these applications may gain a remarkable performance improvement. This composition tool not only provides tool support for PEPPHER components within the PEPPHER framework, but also gives a necessary supplement for StarPU by static composition in a more general way.

As future work, we plan to implement a history-based performance model which guides automatic static composition. The history-based performance model calculates performance data based on off-line training execution.
Bibliography


Glossary

**Code portability** – An indicator of how easy it is to port an application to different platforms.

**Component** – A modular and replaceable program module.

**Component configuration** – A structure that shows which implementation should be used in order to provide the functionality that the component’s interface declares.

**Component model** – A specification for how to write components.

**Composition** – A binding of a call to a callee.

**Composition tool** – A pre-compiler that facilitates to make composition choices and performs some necessary transformations.

**CPU** – A general purpose processing unit that is most commonly used in desktop computers.

**CPU wrapper** – A function generated by the composition tool presented in this thesis, used for adapting parameters from StarPU format to that of normal C/C++ code executing on CPU.

**CUDA** – A framework developed by Nvidia to facilitate access to Nvidia GPUs.

**CUDA wrapper** – A function generated by the composition tool presented in this thesis, used for adapting parameters from StarPU format to that of CUDA code executing on CUDA devices.

**Default implementation** – The implementation that is picked among a set of implementation variants for execution on a specific platform.

**Dynamic composition** – The behavior that determines composition choice at run time.

**Entry component** – The application’s entry function, normally the “main” function for a C/C++ application.

**GPGPU** – The technique that ports general purpose computation to GPUs.

**GPU** - Graphics processing unit, optimized for processing computer graphics.

**Heterogeneous architecture** – A multi-core system that consists processors of different type.

**Homogeneous architecture** – A multi-core system that consists processors of the same type.

**Implementation** – A function body that implements an interface.

**Interface** – A function declaration, either explicit or implicit.

**Intermediate representation (IR)** – A data structure commonly used in compiler technology, which is constructed from compiler input and in turn determines compiler output.

**Linking statement** – The compile command in a makefile that links all object files to an executable.

**Main header file** – One of the files generated by the composition tool presented in this thesis, which provides simple access to other generated files.

**Main wrapper** – One of functions generated by the composition tool presented in this thesis, used for initialization, submission and destroy of a StarPU task.
OpenCL – A language framework that is designed to support different types of processors on heterogeneous architectures.

OpenCL wrapper – A function generated by the composition tool presented in this thesis, used for adapting parameters from StarPU format to that of OpenCL code executing on OpenCL devices.

Pass – A piece of code that traverses the input program on its intermediate representation for one time, often used in compiler technology.

PEPPHER – An EU FP7 project which suggests a framework for developing applications on architecturally diverse, heterogeneous many-core processors to ensure performance portability.

Performance portability – An indicator of how well an application performs when porting it to different platforms.

StarPU – A run time system offers dynamic composition and scheduling for applications that executes on GPU-based heterogeneous architectures.

StarPU task – The unified and schedulable unit in the StarPU run time system.

Static composition – The behavior that determine composition choice at compile time.

Wrapper header file – One of the files generated by the composition tool presented in this thesis, which consists of the main wrapper, the CPU wrapper, the CUDA wrapper and the OpenCL wrapper.

Xerces-C++ – A validating XML parser which makes it easy to give external applications the ability to read XML documents.

XML abstract syntax tree – A tree representation of the abstract syntactic structure of source code written in XML language.

XML annotation – Meta data in XML format that describes important properties of an application.

XML schema – An XML specification which defines valid XML instantiations.