Design and Implementation of a Compiler for an XML-based Hardware Description Language to Support Energy Optimization

Design och implementering av en kompilator för ett XML-baserat hårdvarubeskrivande språk med support för energiöptimering

Ming-Jie Yang

Supervisor: Lu Li
Examiner: Christoph Kessler
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting 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/](http://www.ep.liu.se/)

© Ming-Jie Yang
Abstract

GPU-based heterogeneous system architectures are popular as they combine the advantages of CPU with the benefits of GPU. Development of high-performance and power-efficient software for heterogeneous system architecture needs to take both hardware and software specifications into consideration, which leads the software development process to be more complicated. To simplify the software development process, Architecture Description Languages (ADLs) came out. By modeling the target architecture components into structural formats, programmers can adapt their software to the platforms which they used.

XPDL is a modular and extensible XML-based platform description language which is mainly designed to support optimization. The purposes of this thesis are to design the query API (Application Programming Interface) and develop a compiler which translates the XPDL descriptors to libraries that implement the API to support programmers for the development of adaptive high-performance and energy-optimized software.

In this thesis, we design and develop a compiler to generate the API according to the XPDL descriptors. The main workflow of the designed compiler is following: first, the toolchain validates the XPDL descriptors against XSDs. Second, it parses the descriptors into DOM trees and transforms them into XPDL model trees. Next, the compiler links all XPDL model trees together, which results in the intermediate representation (IR). Then, any unspecified node values which mean the unknown attributes, are handled by microbenchmark generator and executor. In the end, the code generator generates the libraries which expose the API according to the information in the IR. Finally, a few example codes are discussed to show how the API can be used to develop performance adaptive applications on heterogeneous systems.
Acknowledgments

My deepest respect and gratitude go to my supervisor Lu Li and examiner Christoph Kessler for their patience and advice. I thank Therése Luong, Zhili Liu, Stefan Persson and Hans Tchou, for their support and encouragement. A special thanks to Jianxing Dai for his suggestion through the revising process. Finally, I would like to thank my friends Tun-Wei Hsu, Ze-Yu Huang, Chun-Fu Kuei and Chang-Chun Wu for the wonderful memories in Linköping.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables viii

1 Introduction 1
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Research questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.4 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.5 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Background 3
2.1 Architecture Description Language . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Platform Description Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.3 Extensible Markup Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.3.1 XML Schema Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.3.2 Xerces-C++ XML Parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.4 GPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.4.1 CUDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.5 Intermediate Representation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.6 MeterPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.7 Metaprogramming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.8 Boost C++ Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.9 DOT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.10 XPDL Language Design as Source Language . . . . . . . . . . . . . . . . . . . . 7
2.10.1 XPDL Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.10.2 Name-Type Relation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.10.3 CPU Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.10.4 GPU Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.10.5 Memory Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.10.6 Interconnect Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.10.7 Instruction Power Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.10.8 Software Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.10.9 Microbenchmark Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.10.10 System Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3 XPDL Query API Design 13
List of Figures

2.1 Result of executing the instruction energy cost with MeterPU .................. 6
2.2 An example of using DOT to describe a graph ........................................ 7
2.3 A simple CPU structure ............................................................................ 8

4.1 Structure of XPDL compiler ................................................................. 15
4.2 Workflow of XPDL parser ..................................................................... 16
4.3 a) An example XML code b) XML code parsed into DOM tree ............ 17
4.4 UML class diagram for XPDL node ....................................................... 17
4.5 Principle of the model linker .................................................................. 18
4.6 An example file which contains the information of the IR .................. 20
4.7 Structure of IR Visualizer ....................................................................... 20

5.1 Pseudo code of tree transformation ....................................................... 24
5.2 Pseudo code of getting attributes .......................................................... 24
5.3 Pseudo code of mirobenchmark generator .............................................. 25
5.4 Pseudo code of mirobenchmark executor .............................................. 25
5.5 The IR before and after microbenchmark executor ............................... 25
5.6 Pseudo code for code generator .............................................................. 26
5.7 Example pseudo code for code generation ............................................. 26

6.1 Result for example program 1 ................................................................. 30
6.2 Result for example program 2 ................................................................. 30
6.3 Result for example program 3 ................................................................. 31
List of Tables

7.1 Features comparison between XPDL, PDL, HPP-DL, ALMA-DL and hwloc . . . 34
<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Example of an XML document</td>
<td>4</td>
</tr>
<tr>
<td>2.2 This example XSD file corresponds to Listing 2.1</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Measure the instruction energy cost with MeterPU</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Example of the name-type relation</td>
<td>7</td>
</tr>
<tr>
<td>2.5 Descriptor for a simple multi-core CPU</td>
<td>8</td>
</tr>
<tr>
<td>2.6 Descriptor for L1 cache</td>
<td>8</td>
</tr>
<tr>
<td>2.7 Descriptor for L2 cache</td>
<td>9</td>
</tr>
<tr>
<td>2.8 Descriptor for L3 cache</td>
<td>9</td>
</tr>
<tr>
<td>2.9 Descriptor for Nvidia K20c</td>
<td>9</td>
</tr>
<tr>
<td>2.10 Descriptor for DDR3 8G</td>
<td>9</td>
</tr>
<tr>
<td>2.11 Descriptor for DDR3 16G</td>
<td>10</td>
</tr>
<tr>
<td>2.12 Descriptor for KHX16C9T3K2</td>
<td>10</td>
</tr>
<tr>
<td>2.13 Descriptor for PCIe3</td>
<td>10</td>
</tr>
<tr>
<td>2.14 Descriptor for the instruction power modeling</td>
<td>10</td>
</tr>
<tr>
<td>2.15 Descriptors for software modeling</td>
<td>11</td>
</tr>
<tr>
<td>2.16 Descriptors for instruction power modeling of subtraction</td>
<td>11</td>
</tr>
<tr>
<td>2.17 Descriptor for microbenchmark</td>
<td>11</td>
</tr>
<tr>
<td>2.18 Descriptor for whole system</td>
<td>12</td>
</tr>
<tr>
<td>4.1 User-defined source</td>
<td>18</td>
</tr>
<tr>
<td>4.2 The source code generated by microbenchmark generator</td>
<td>19</td>
</tr>
<tr>
<td>4.3 An example metafunction of a CPU</td>
<td>21</td>
</tr>
<tr>
<td>4.4 An example metafunction of the system</td>
<td>21</td>
</tr>
<tr>
<td>4.5 An example metafunction of the software component</td>
<td>22</td>
</tr>
<tr>
<td>6.1 Example program 1</td>
<td>29</td>
</tr>
<tr>
<td>6.2 Example program 2</td>
<td>30</td>
</tr>
<tr>
<td>6.3 Example program 3</td>
<td>31</td>
</tr>
</tbody>
</table>
1 Introduction

In this chapter, we provide the introduction to this thesis. First, we explain the motivation in Section 1.1. Then we discuss the aim, research questions, and delimitations of this thesis in Section 1.2, Section 1.3, and Section 1.4 respectively. Finally, the outline of the thesis is given in Section 1.5.

1.1 Motivation

In recent years, new research studies have appeared which tackle the issue of high-performance software. With the increasing of the software complexity, traditional single-core CPU (Central Processing Unit) systems can not meet the performance requirements of the modern software. The clock speed of single-core CPU architectures is bounded by the physical limitations such as transmission time, switching delay and power consumption. To deal with the limitations, multi-core CPU architectures were developed. In general, multi-core CPU architectures have higher performance against single-core CPU architectures. Nevertheless, some of the software applications perform better on GPU (Graphics Processing Unit), e.g., data-parallel computational application. On the other hand, GPU will not perform efficiently if there are only a few data points to compute. Therefore, heterogeneous system architecture has been widely used due to combining the advantages of both CPU and GPU. Development of high-performance and power-efficient software for heterogeneous system architecture needs to take both hardware and software specifications into consideration, which leads the software development process to be more complicated.

Developing high-performance and energy-optimized software needs knowledge of hardware and software information (e.g., the number of threads, memory bandwidth, compatibility of GPU, the energy cost of particular instructions), so Architecture Description Languages (ADLs) come out. Each kind of ADL has its purposes and limitations, such as PDL [12] (PEPHER platform description language) is an XML-based platform description language which can support programmers and tools developing high-performance software. However, PDL is mainly for single node heterogeneous system architecture and lacks modularity and extensibility.

XPDL [6] is an XML-based platform description language which supports energy modeling. It is designed to overcome the limitations of PDL and make the language more elegant.
In XPDL, hardware and software information are structurally modeled separately. Moreover, XPDL also improves reusability. For more details, see Section 2.10.

1.2 Aim

The aims of this thesis are to design the query API (Application Programming Interface) and develop a compiler which translates the XPDL descriptors to libraries which implement the API. As mentioned in Section 1.1, we need detailed knowledge of hardware and software information to develop high-performance and energy-optimized software; hence the query API should provide an interface to get the information of the system.

Although XPDL provides a solution to characterize the hardware and software in the target platform, we need a compiler to transform the information for further use. The compiler should be able to read the XPDL descriptors from the indicated path, retrieve the information described in the descriptors, generate and execute the microbenchmarks for unknown attributes and finally, generate query-enabled libraries.

1.3 Research questions

The following research questions are proposed to help readers further understand the objectives of this thesis.

1. Compiler-generated API
   How to design an expressive and low-overhead API which provides an interface for querying the information about the target platform?

2. Compiler
   How to design such a compiler which translates the XPDL descriptors to libraries that implement the API?

1.4 Delimitations

With the rapid development of computer technology, new architectures are proposed from time to time. Although we try to make the language and compiler generic, there might exist some architectures whose specifications can not be met. Those unique platforms will not be discussed.

The libraries generated by the compiler depend on the Boost C++ library. Users who use this API are assumed to have the environment to support them. Support for other languages is not considered in this current stage of this project.

1.5 Thesis Outline

This thesis contains eight chapters. Chapter 1 gives an introduction to the project. Chapter 2 provides background which is used in this thesis. Chapter 3 introduces the design of the API. Chapter 4 and Chapter 5 present the design and implementation of the compiler respectively. Chapter 6 shows the result and the discussion. Chapter 7 discusses the related work. Chapter 8 covers the conclusion and future work.
2 Background

From Section 2.1 to Section 2.9, we cover the technical background used in this thesis. Section 2.10 presents the basics of XPDL, including the overview of XPDL, name-type relation, hardware modeling, software modeling and system modeling.

2.1 Architecture Description Language

Architecture Description Languages (ADLs) are used to characterize the software or hardware architecture of a computer system. Software ADLs, see e.g. [3, 13, 5, 8, 9, 10], describe the software information such as structure, processes, threads, data, function relations, etc. Hardware ADLs, see e.g. [4, 14, 11] typically describe the hardware components in a system such as processing units, memory, interconnect, and so on. ADLs are usually structurally formed to represent the target architectures. Also, they are both machine and human readable. In this thesis, the term ADL indicates the hardware ADL.

2.2 Platform Description Language

A platform consists of hardware components and software installed in the system. Thus, platform description languages (PDLs) refer to the computer languages which describe not only the hardware but also the installed software in the systems, see e.g. [12, 6].

2.3 Extensible Markup Language

Extensible Markup Language (XML) is a markup language which aims to help interpret the data and provides a mechanism to characterize the data structurally. Unlike other markup languages, tags in XML are not fixed, which makes XML extensible. On account of the extensibility, we can design another markup language by XML. An XML document mainly consists of three parts, prolog, root element, and elements. Prolog indicates that this is an XML document and has two attributes to declare the version and the text encoding method. Every XML document must have a root element which is the ancestor of other elements in the document. In the elements, there might be some attributes describing their characteristics. All of the attribute values must be quoted. In Listing 2.1 it presents a simple
2.3. Extensible Markup Language

XML document. The `<country_list>` tag indicates the root element of this XML document and `<country>` tags with attributes name, capital, population, and currency are the element of the XML document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<country_list>
  <country name="Sweden" capital="Stockholm" population="10053061"
    currency="SEK" />
  <country name="Japan" capital="Tokyo" population="126740000"
    currency="JPY"/>
</country_list>
```

Listing 2.1: Example of an XML document

In the following sections, we introduce XML Schema Definition in Section 2.3.1 and Xerces-C++ XML Parser in Section 2.3.2.

2.3.1 XML Schema Definition

XML Schema Definition (XSD), which uses the XML syntax, defines the structure (or rules) of the XML document as in Listing 2.2 and its main purposes are [15]:

1. Limit the appearance of elements and attributes.
2. Define the number of elements.
3. Declare the data type of elements and attributes.
4. Set the default value for elements and attributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema>
  <element name="country_list" type="country_list_type" />
  <complexType name="country_list_type">
    <sequence>
      <element name="country" type="country_type" maxOccurs="unbounded" minOccurs="1" />
    </sequence>
  </complexType>
  <complexType name="country_type">
    <attribute name="name" type="string"/>
    <attribute name="capital" type="string"/>
    <attribute name="population" type="int"/>
    <attribute name="currency" type="string"/>
  </complexType>
</schema>
```

Listing 2.2: This example XSD file corresponds to Listing 2.1

2.3.2 Xerces-C++ XML Parser

Xerces-C++ XML Parser is an open-source XML parser developed by Apache Software Foundation. It provides an API to read and write XML files and validate the XML against the predefined XSD.

2.4 GPU

A graphics processing unit (GPU) is a processor originally used to increase the performance of graphic and video processing. Recently, GPU is also used to accelerate applications with large amounts of data to compute. Compare to CPU, GPU contains hundreds of cores which can handle thousands of threads at the same time. The compute capability refers the specification and available features on particular Nvidia GPUs.

2.4.1 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform created by Nvidia. In CUDA, a grid contains blocks which are isolated from other grids and a block contains multiple threads. Threads within the same block communicate with others by shared memory.

2.5 Intermediate Representation

Usually, an Intermediate Representation (IR) is constructed from the input by compiler, and it is typically represented in the tree structure or linear representation. It should be easy to produce and be translated into target machine code. The design of the IR will affect the efficiency of the compiler. The advantage of using an IR in this project is to reduce the difficulty of code generation. Instead of generating the code directly from the complicated source, the code can be generated from a structured IR.

2.6 MeterPU

MeterPU is a C++ template-based generic API for measuring energy and time cost of hardware components such as CPU, GPU, and DRAM. In this thesis, we adapt MeterPU to be used with the XPDL compiler for benchmarking the energy consumption of instructions. In MeterPU, the measurement of energy consumption of CPU is implemented by Intel Performance Counter Monitor (PCM). MeterPU integrates various functions into simple interfaces which simplify the measurement. In Listing 2.3, it shows the example of measuring the instruction power consumption of the CPU with MeterPU, and Figure 2.1 shows the execution result.

---

2http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
2.7 Metaprogramming

A metaprogram is a program that generates or modifies other programs \[1\]. The process of writing metaprogram is called metaprogramming.

In C++, metaprogramming can be achieved by the template construct. When providing arguments to a template, it will be instantiated with those arguments. Moreover, the template instantiation performs at compile time when all of the arguments of a template are known. This kind of programming technique is called C++ template metaprogramming.

The constexpr specifier (since C++11) declares that values of functions or variables can be computed at compile time. It is another programming construct of metaprogramming.

2.8 Boost C++ Libraries

Boost C++ Libraries\(^4\) are a collection of open-source C++ libraries. They provide functionality which is missing in the C++ standard template library (STL), in another word, they are a complement of C++ STL.

2.9 DOT

DOT\(^5\) is an open-source graph description language developed by AT&T. Figure 2.2 is an example of using DOT to describe a graph.

\(^4\)http://www.boost.org
\(^5\)http://www.graphviz.org/content/dot-language
2.10 XPDL Language Design as Source Language

In this section, the design of the XPDL language is presented, starting with an overview of XPDL. Then the name-type reference relation is introduced. From Section 2.10.3 to Section 2.10.8, we describe the ways of modeling the variety of hardware and software components. Finally, the microbenchmark modeling and system modeling will be covered.

2.10.1 XPDL Overview

XPDL is a Platform Description Language which characterizes hardware and software components in a computer system. Different types of components, such as CPUs, GPUs, memories, interconnects, instruction power, and installed software are described in XPDL models. Models are organized in separate descriptors. XPDL models are reusable. By assigning a name to them, others can reference the models by their name.

The \textbf{group} element is used to group elements. The attribute \texttt{quantity} used in the group indicates that the elements in the group are homogeneous. With the attributes \texttt{prefix} and \texttt{quantity} together, we can assign IDs to the elements in the group automatically.

The \textbf{microbenchmark} element is used to describe the microbenchmark information such as the source name, path, compiler and compile information. For more details, refer to Section 2.10.9.

The \textbf{system} element is used to model the whole system. It characterizes the number of each hardware components and software components installed in the system. See Section 2.10.10 for more details.

2.10.2 Name-Type Relation

The attributes \texttt{name} and \texttt{type} indicate the reference relation between two models. If there is an attribute called \texttt{type} with value and an attribute \texttt{name} with the same value as \texttt{type} in another descriptor, the name-type relation is established. It means that the model with the attribute \texttt{type} inheritances the model with the attribute \texttt{name}. In Listing 2.4, we can observe that model A inheritances model B. This mechanism makes the descriptors reusable.

\begin{verbatim}
<!---A.xml -->
<xpd1:A type="b"/>
</!---A.xml -->
\end{verbatim}

\begin{verbatim}
4 <xpdl:B name="b" attr_0="b0" attr_1="b1" attr_2="b2"/>
\end{verbatim}

Listing 2.4: Example of the name-type relation
2.10.3 CPU Modeling

A general multi-core CPU architecture comprises cores and caches. In the example in Figure 2.3, each core contains its own L1 cache; L2 cache is often shared by two cores; L3 cache is shared among all cores on the chip. A CPU model usually contains the following information: frequency, frequency unit, number of cores, cache names, cache sizes and cache size units.

Listing 2.5 shows the descriptor which corresponds to Figure 2.3. The <group> tag indicates the elements in the scope are homogeneous. The attribute quantity specifies the number of the elements in the group. With the attribute quantity following prefix, ID are assigned to elements in the group automatically. We take Listing 2.5 as an example, with prefix=core_group and quantity=2, the IDs of the elements are assigned as core_group 0 and core_group 1.

To increase the reusability, we characterize the cache information into other descriptors instead of enclosing it into the descriptor of CPU. The name-type pair represents the reference relation as mentioned in Section 2.10.2. The cache elements in Listing 2.5 reference the attributes size and unit from the descriptors in Listing 2.6, 2.7 and 2.8.

```xml
1 <!---descriptor for the simple CPU -->
<xpdl:cpu name="Intel_Xeon_E5_2630L" num_of_cores="4" num_of_threads="8">
  <xpdl:group prefix="core_group" quantity="2">
    <xpdl:group prefix="core" quantity="2">
      <xpdl:core frequency="2" unit="GHz"/>
    </xpdl:group>
    <xpdl:cache type="L1"/>
  </xpdl:group>
  <xpdl:cache type="L2"/>
  <xpdl:cache type="L3"/>
</xpdl:cpu>
```

Listing 2.5: Descriptor for a simple multi-core CPU

```xml
<!--descriptor for L1 cache -->
<xpdl:cache name="L1" size="32" unit="KiB"/>
```

Listing 2.6: Descriptor for L1 cache
2.10. XPDL Language Design as Source Language

2.10.4 GPU Modeling

The `<gpu>` tag is used to characterize different kinds of GPUs. A GPU model consists of the following information: maximum dimension size of a grid (x,y,z), maximum dimension size of a thread block (x,y,z), compute capability, core frequency with unit, memory size with unit and cache size with unit. Listing 2.9 shows an example descriptor for a Nvidia K20c GPU.

```xml
<!--descriptor for Nvidia_K20c -->
<xpdl:gpu name="Nvidia_K20c" compute_capability="3.52"
max_dim_grid_size_x="2147483647" max_dim_grid_size_y="65535"
max_dim_grid_size_z="65535" max_dim_thread_block_x="1024"
max_dim_thread_block_y="1024" max_dim_thread_block_z="64"
max_threads_per_block="1024" type="nvidia_gpu" num_of_cores="2496">
  <xpdl:group name="SMs" quantity="13">
    <xpdl:group name="SM" quantity="192">
      <xpdl:core frequency="0.706" frequency_unit="GHz"/>
      <xpdl:memory name="Shared_per_block" size="49152" unit="bytes"/>
    </xpdl:group>
    <xpdl:cache name="L1_per_SMs" size="64" unit="KiB"/>
  </xpdl:group>
  <xpdl:memory name="Global" size="5120" unit="MiB"/>
  <xpdl:cache name="L2_shared" size="768" unit="KiB"/>
</xpdl:gpu>
```

Listing 2.9: Descriptor for Nvidia K20c

2.10.5 Memory Modeling

The `<memory>` tag is used to characterize the different kinds of memory modules. In a memory module, it contains the information of size, size unit, bandwidth, bandwidth unit, static power and static power unit.

Listing 2.10, 2.11 and 2.12 show example models of memory modules. The examples indicate that descriptors of DDR3_8G and DDR3_16G reference the same descriptor, KHX16C9T3K2, which means that the attributes `static_power` and `static_power_unit` are inherited by DDR3_8G and DDR3_16G.

```xml
<!--descriptor for DDR3_8G -->
<xpdl:memory name="DDR3_8G" type="KHX16C9T3K2" size="8" unit="GiB"
bandwidth="?"/>
```

Listing 2.10: Descriptor for DDR3 8G
2.10. XPDL Language Design as Source Language

```
<!--descriptor for DDR3_16G -->
<xpdl:memory name="DDR3_16G" type="KHX16C9T3K2" size="16" unit="GiB"
  " bandwidth="?"/>

Listing 2.11: Descriptor for DDR3 16G
```

```
<!--descriptor for KHX16C9T3K2 -->
<xpdl:memory name="KHX16C9T3K2" static_power="2.460"
  static_power_unit="W" />

Listing 2.12: Descriptor for KHX16C9T3K2
```

2.10.6 Interconnect Modeling

The `<interconnect>` tag is used to characterize a variety of interconnects. Listing 2.13 shows an example descriptor for PCIe3. The `<channel>` tags indicate that this interconnect component comprises different kinds of channels. Since each channel has different bandwidth and energy cost, we model them separately.

```
<!--descriptor for PCIE3 -->
<xpdl:interconnect name="pcie3">
  <xpdl:channel name="up_link" max_bandwith="2"
    max_bandwith_unit="GiB" energy_per_byte="35"
    energy_per_byte_unit="nJ"/>
  <xpdl:channel name="down_link" max_bandwith="4.5"
    max_bandwith_unit="GiB" energy_per_byte="15"
    energy_per_byte_unit="nJ"/>
</xpdl:interconnect>

Listing 2.13: Descriptor for PCIe3
```

2.10.7 Instruction Power Modeling

The `<instruction>` tag is used to model the instruction energy cost. Listing 2.14 is an example of instruction power modeling. Each model contains the information of energy cost and its unit.

```
<!--descriptor for instruction "add" -->
<xpdl:instruction name="add" energy_cost="7.33e-06"
  energy_cost_unit="mJ"/>

<!--descriptor for instruction "sub" -->
<xpdl:instruction name="sub" energy_cost="7.05e-06"
  energy_cost_unit="mJ"/>

<!--descriptor for instruction "mul" -->
<xpdl:instruction name="mul" energy_cost="8.10e-06"
  energy_cost_unit="mJ"/>

Listing 2.14: Descriptor for the instuction power modeling
```

2.10.8 Software Modeling

The tags `<library>`, `<os>` and `<software>` represent the library, operating system, and software which are installed in the system respectively. In Listing 2.15 it shows the
example descriptors of cuBlas, ubuntu, and Matlab.

```xml
<!--descriptor for cuBlas -->
<xpdl:library name="cublas" path="../cublas_path" version="5.0"/>
<!--descriptor for ubuntu -->
<xpdl:os name="ubuntu" path="../os_path" version="12.3"/>
<!--descriptor for matlab -->
<xpdl:software name="matlab" path="../matlab_path" version="7.0.2"/>
```

Listing 2.15: Descriptors for software modeling

### 2.10.9 Microbenchmark Modeling

Listing 2.16 shows an example descriptor which has a question mark as attribute value of energy_cost. The question mark indicates that the attribute value is unknown and needs a microbenchmark to measure it. The XPDL compiler provides a mechanism to generate and execute the microbenchmark (see Section 4.2 and 4.3). However, it is necessary to provide the information to the compiler so that the compiler knows what to generate and execute. Listing 2.17 is an example descriptor for a microbenchmark. Information used to model the microbenchmark including:

1. name to indicate the microbenchmark model is referenced by the other model (Listing 2.16 in this case).
2. src and path to describe the name and path of the user-defined source respectively.
3. compiler, cflags, lflags, objs, and libs to express the information which are needed to compile the microbenchmark code.

```xml
<!--descriptor for instruction "sub" -->
<xpdl:instruction name="ins_sub" energy_cost="?" energy_cost_unit="mJ">
    <microbenchmark type="mb_ins_sub"/>
</xpdl:instruction>
```

Listing 2.16: Descriptors for instruction power modeling of subtraction

```xml
<!--descriptor for microbenchmark of instruction energy cost -->
<xpdl:microbenchmark name="mb_ins_sub" src="energy_cost.cpp" path="/microbenchmark/01_Instruction_Power/01_Sub" compiler="g++" cflags="-I $XPDL_HOME/meterpu -g3 -std=c++11 -DENABLE_PCM -DENABLE_PTHREAD -I$XPDL_HOME/IntelPerformanceCounterMonitorV2.8 -I . -O3" lflags="-lpthread" objs="$XPDL_HOME/IntelPerformanceCounterMonitorV2.8/cpucounters.o $XPDL_HOME/IntelPerformanceCounterMonitorV2.8/msr.o $XPDL_HOME/IntelPerformanceCounterMonitorV2.8/pci.o $XPDL_HOME/IntelPerformanceCounterMonitorV2.8/client_bw.o" libs=""/>
```

Listing 2.17: Descriptor for microbenchmark

### 2.10.10 System Modeling

We model the whole system by using the tag <system>. The system model describes what kind of hardware components and software components are in the system. Listing 2.18 is an example system model that contains:
1. A CPU with id=xpdl_cpu which references the descriptor named Intel_Xeon_E5_2630L.

2. A GPU with id=xpdl_gpu which references the descriptor named Nvidia_K20c.

3. Two memories with id=xpdl_memory_1 and id=xpdl_memory_2 which reference the descriptors DDR3_16G and DDR3_8G, respectively.

4. An interconnect with id=xpdl_interconnect which references the descriptor pcie3.

5. Two installed libraries with id=xpdl_blas and xpdl_cublas which reference the descriptor blas and cublas respectively.

6. An installed software with id=xpdl_matlab which references the descriptor matlab.

7. An installed operating system with id=xpdl_os which references the descriptor ubuntu.

8. An instruction energy cost model with id=xpdl_ins_add which references the descriptor ins_add

```xml
<!--descriptor for whole system -->
<xpdl:system>
  <xpdl:cpu id="xpdl_cpu" type="Intel_Xeon_E5_2630L"/>
  <xpdl:gpu id="xpdl_gpu" type="Nvidia_K20c"/>
  <xpdl:memory id="xpdl_memory_2" type="DDR3_8G"/>
  <xpdl:memory id="xpdl_memory_1" type="DDR3_16G"/>
  <xpdl:software id="xpdl_matlab" type="matlab"/>
  <xpdl:interconnect id="xpdl_interconnect" type="pcie3"/>
  <xpdl:library id="xpdl_blas" type="blas"/>
  <xpdl:library id="xpdl_cublas" type="cublas"/>
  <xpdl:os id="xpdl_os" type="ubuntu"/>
  <xpdl:instruction id="xpdl_ins_add" type="ins_add"/>
</xpdl:system>
```

Listing 2.18: Descriptor for whole system
In this chapter, we present the design of the query API. We start from addressing the requirements. Following is the design of it. In the end, we provide some examples of each category of API.

3.1 Requirements of XPDL Query API

The XPDL Query API should include the functions which can:

1. Access the element attributes corresponding to the XPDL descriptors.
2. Get the aggregate quantities of a certain type of components in the system.
3. Check whether a particular software component is installed in the system or not.

3.2 Design

We map each of the components in the system into a corresponding C/C++ struct, which can encapsulate small groups of related properties or the characteristics of an element in inventory. Furthermore, the names of the struct for hardware and software components are defined as the component type with index and software name with version following (e.g., xpdl_cublas_5_0), respectively. The index is depending on the number of certain type of hardware components, e.g., if there are three CPUs in the system, the name of structs will be cpu_1, cpu_2, and cpu_3. The attributes of a component are incorporated in properties name and value in its own struct block. The number of instances of each component type are summed up and be represented as variable names and values in an additional struct which is named as system. Metaprogramming is used in the generated API. With meta programming, we have no cost of calling the API during runtime. Since the compiler knows all of the variable values at compile-time, we can get them directly. We also introduce getter functions to get the attributes of a particular component. Moreover, we also provide functions which can verify if a software component exists in the system.
3.3 API Examples

Currently, we have seven categories of query API. They are CPU, GPU, memory, interconnect, instruction energy cost, software, and system. In this section, we present examples of each category of API.

1. CPU
   
   `xpdl::cpu_1::num_of_cores;
   xpdl::get<xpdl::system::cpus,0>::num_of_cores;
   xpdl::cpu_1::num_of_hardware_threads;
   xpdl::get<xpdl::system::cpus,0>::num_of_hardware_threads;
   xpdl::cpu_1::l1_cache_size;
   xpdl::get<xpdl::system::cpus,0>::l1_cache_size;

2. GPU
   
   `xpdl::gpu_1::num_of_cores;
   xpdl::get<xpdl::system::gpus,0>::num_of_cores;
   xpdl::gpu_1::compute_capability;
   xpdl::get<xpdl::system::gpus,0>::compute_capability;
   xpdl::gpu_1::core_frequency;
   xpdl::get<xpdl::system::gpus,0>::core_frequency;

3. Memory
   
   `xpdl::memory_1::size;
   xpdl::get<xpdl::system::memories,0>::size;
   xpdl::memory_1::bandwidth;
   xpdl::get<xpdl::system::memories,0>::bandwidth;
   xpdl::memory_1::static_power;
   xpdl::get<xpdl::system::memories,0>::static_power;

4. Interconnect
   
   `xpdl::interconnect_1::up_link_max_bw;
   xpdl::get<xpdl::system::interconnects,0>::up_link_max_bw;
   xpdl::interconnect_1::up_link_energy_per_byte;
   xpdl::get<xpdl::system::interconnects,0>::up_link_energy_per_byte;
   xpdl::interconnect_1::up_link_energy_per_byte_unit;
   xpdl::get<xpdl::system::interconnects,0>::up_link_energy_per_byte_unit;

5. Instruction Energy Cost
   
   `xpdl::instruction_1::energy_cost;
   xpdl::get<xpdl::system::instructions,0>::energy_cost;
   xpdl::instruction_1::energy_cost_unit;
   xpdl::get<xpdl::system::instructions,0>::energy_cost_unit;

6. Software
   
   `is_installed<xpdl::system::libraries,xpdl_xblas>::value
   xpdl::get<xpdl::system::libraries,xpdl_xblas>::path;
   xpdl_blas_5_0::path

7. System
   
   `xpdl::system::num_of_cpus;
   xpdl::system::num_of_gpus;
   xpdl::system::num_of_memories;
Chapter 4 presents the design of the XPDL compiler. The XPDL compiler is mainly composed of seven parts as shown in Figure 4.1: the parser, microbenchmark generator, microbenchmark executor, IR loader, IR saver, IR visualizer and code generator. The parser validates the XPDL descriptors against the XSD and links the sub-model trees to a single model tree recursively. The IR loader loads back the IR from the file. The microbenchmark generator generates the microbenchmark driver. The microbenchmark executor executes the driver which merges with user-defined code under measurement and writes the result back to the IR. When the IR is ready, the IR saver saves the IR into file, the IR visualizer generates the graph of IR and the code generator produces the API according to the IR.

Figure 4.1: Structure of XPDL compiler
4.1 Parser

There are three phases in the parser as shown in Figure 4.2, which are XML validation, tree transformation and model linker. In this section, we introduce the design of XPDL parser which is based on Xerces-C++ XML Parser.

4.1.1 XML Validation

First, the XPDL parser reads XML and XSD files as inputs and then validates the XML file against its corresponding XSD file by using Xerces-C++ XML Parser. If there is any syntax error in the XML files, the error messages will pop up in the terminal indicating the position where the error occurs, and the program will be terminated. Once the XML passes the validation, the parser parses it into DOM tree.

4.1.2 Tree Transformation

Although Xerces C++ XML Parser produces a tree structure, the unnecessary nodes and the limitation to reference other nodes make the tree inflexible. When an XML file be parsed into a DOM tree, there exist nodes with #Text as names and the empty string as node values, see Figure 4.3. Since these redundant nodes will not be used in the future stages, we decide to abandon them. A variety of the DOM node types increase the complexity of tree operation. Accessing the nodes in the tree needs to check the type before calling the corresponding functions.

Due to the reasons above, we introduce our tree structure. As shown in Figure 4.4, the XPDL node is designed as name-value pair with a vector of pointers storing their children and an ID as its identifier. By providing the set() and get() functions, we can easily modify and retrieve the node properties. We recursively visit each node in the DOM tree and fetch the information to generate the XPDL node. After the process, the XPDL tree is formed, and the DOM trees are eliminated.

---

1See https://www.w3schools.com/xml/dom_nodetype.asp
4.1. Parser

Figure 4.3: a) An example XML code b) XML code parsed into DOM tree

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td># name: string</td>
</tr>
<tr>
<td># value: string</td>
</tr>
<tr>
<td># ID: int</td>
</tr>
<tr>
<td># parent: Node*</td>
</tr>
<tr>
<td># children: vector&lt; Node* &gt;</td>
</tr>
<tr>
<td>+ getChildren(): vector&lt; Node* &gt;</td>
</tr>
<tr>
<td>+ getNodeName(): string</td>
</tr>
<tr>
<td>+ getNodeValue(): string</td>
</tr>
<tr>
<td>+ setNodeValue(): void</td>
</tr>
<tr>
<td>+ setNodeName(): void</td>
</tr>
</tbody>
</table>

Figure 4.4: UML class diagram for XPDL node

4.1.3 Model Linker

The model linker is used to link the sub-model trees into a concrete model tree. *name* and *type* attributes in XPDL refer to the reference relation. First, XPDL parser loads system.xml and transfers it into an XPDL tree. Then, we recursively search the type node in the tree and search the corresponding XML file, which presents as the value of type node, in the directory. Finally, the corresponding XML file will be parsed into trees and linked to the mother tree. If there is any type node in the currently loaded tree, the parser will do the same process as mentioned above. When there is no reference relation in the current level, the linking process moves to next type node in the tree, which is generated from the descriptor of *system*. 
4.2 Microbenchmark Generation

If there is a name-type pair established, the tree with type node inheritances the tree with name node as shown in Figure 4.5. After all the sub-model trees are linked together, the IR is formed.

4.2 Microbenchmark Generation

If there is a question mark as an attribute value in XPDL descriptors and users have already modeled the microbenchmark (see Section 2.10.9), it will trigger the microbenchmark generator. Microbenchmark generator produces the microbenchmark code by specified users-defined source. For example, we define a piece of C++ code in Listing 4.1, and then the microbenchmark generator will generate the complete microbenchmark code in Listing 4.2.

4.3 Microbenchmark Execution

After generating the microbenchmark, the executor compile and run it. First, the executor fetches the compile information such as compiler, cflags, lflags and location of files from XPDL descriptors. Then, it creates the command to compile and execute the microbenchmark code. Finally, the result which is displayed on the terminal will be fetched and written back to the IR. Since the code should be compiled and executed at the runtime, we use a mechanism which can run the Linux bash command when main program is still executing. When all of the question marks in the IR are handled, the completed IR is formed.

```
c=a+b;
```

Listing 4.1: User-defined source
#include <MeterPU.h>
int main()
{
    using namespace MeterPU;
    int a = 1;
    int b = 2;
    int c;
    Meter<PCM_Energy> meter;
    meter.start();
    for (int i = 0; i < 1000; i++)
    {
        // Unrolled to mitigate jump instruction noise
        c = a + b;
        c = a + b;
        c = a + b;
        c = a + b;
    }
    meter.stop();
    meter.calc();
    // print the result
    meter.get_value();
}

Listing 4.2: The source code generated by microbenchmark generator

4.4 IR Saver

IR saver stores the IR into a file. We define ID as a unique identifier to a node. A counter is used to record the number of created nodes. When a node is created, the counter increases and is assigned as ID to the node. After getting an ID, a node is written into the file which is IR.txt. Each of the nodes in the IR is represented as <ID;parentID;name;value> in the file. ID and parentID are the identifiers to a node and its parent respectively. name and value indicate the name and value of a node. Figure 4.6 shows an example file which contains a part of information of the IR.

4.5 IR Loader

IR loader loads back the IR from IR.txt. IR loader sequentially reads the node information. When a line of information is read, IR loader creates the node and links it to its parent according to the loaded information. When all the nodes represented in the IR.txt are handled by IR loader, the IR is formed.
4.6 IR Visualizer

IR visualizer produces the graph of the IR. It is composed of two parts which are DOT code generator and DOT code executor, see Figure 4.7.

DOT code generator recursively generates the DOT code according to the relation between nodes. DOT code executor runs the DOT file which is generated by the generator and produces the graph of the IR.

4.7 Code Generation

First, the generator generates the headers which are necessary to the libraries. Then, it represents each component into a metafunction as shown in Listing 4.3. All of the XPDL nodes in the tree are denoted as attribute names and values in the struct. To be able to access the attributes, we introduce the get function in line 18-19.

When an XML file is parsed into a tree, a counter of the corresponding component type which is stored in a std::map increases. The counter determines the number of metafunctions for corresponding components. For example, if the counter of CPU is two, then we generate two metafunctions for CPU. Also, the counters of components are also used to generate the number of components in the metafunction of system, see Listing 4.4.

Listing 4.5 is the example code generated from the software component. As hardware components, the generator interprets it into metafunction. The is_installed() functions is generated to check whether a software is installed in the system or not. Before calling the is_installed function, we need to define the macro of the type for the software components.
constexpr specifier supports for compile-time computing. Since all attribute values in the metafunctions are assigned at compile-time, it is almost no cost to get the attribute values during runtime.

```cpp
struct cpu_1
{
    static constexpr char* xpdl_id=(char*)"xpdl_cpu";
    static constexpr int num_of_cores=4;
    static constexpr int num_of_hardware_threads=8;
    static constexpr int core_group_num=2;
    static constexpr int core_in_group_num=2;
    static constexpr double core_frequency=2;
    static constexpr char* core_frequency_unit=(char*)"GHz";
    static constexpr int l1_cache_size=32;
    static constexpr int l2_cache_size=256;
    static constexpr int l3_cache_size=15;
    static constexpr char* l1_cache_size_unit=(char*)"KiB";
    static constexpr char* l2_cache_size_unit=(char*)"KiB";
    static constexpr char* l3_cache_size_unit=(char*)"MiB";
};
```

Listing 4.3: An example metafunction of a CPU

```cpp
struct system
{
    static constexpr int num_of_cpus=1;
    static constexpr int num_of_gpus=1;
    static constexpr int num_of_memorys=2;
    static constexpr int num_of_interconnections=1;
    static constexpr int num_of_instructions=2;

    typedef mpl::vector<cpu_1> cpus;
    typedef mpl::vector<gpu_1> gpus;
    typedef mpl::vector<instruction_1, instruction_2> instructions;
    typedef mpl::vector<interconnect_1> interconnects;
    typedef mpl::vector<memory_1, memory_2> memories;
    typedef mpl::set<xpdl_blas_5_0, xpdl_cublas_5_0> libraries;
    typedef mpl::set<xpdl_matlab_7_0_2> softwares;
    typedef mpl::set<xpdl_os_12_3> oss;
};
```

Listing 4.4: An example metafunction of the system
struct xpdl_blas_5_0
{
    static constexpr char* path=(char*)"../src/cublas";

    static constexpr char* xpdl_id=(char*)"xpdl_blas_5_0";

    static constexpr char* xpdl_type=(char*)"blas";
};

#define XDPL_PREPARE_SW_QUERY(sw_name) struct sw_name

template <class Software_Set, class A_Software>
struct is_installed : mpl::has_key
    <Software_Set, A_Software>::type {};
5 Compiler Implementation

We have already introduced the design of the XPDL compiler in Chapter 4. In this chapter, we present the key building blocks of the compiler. We start from showing how to transfer the DOM tree to XPDL tree. Then we present the design of microbenchmark generator and executor. In the end, we describe the way of implementing code generator.

5.1 Tree Transformation

After a descriptor passes the validation phase in the Xerces parser, a DOM tree is generated. We traverse through all nodes in the tree. If the name of a node is not #Text, we create an XPDL node and copy the name and value from DOM node into it. Then the new XPDL node will be added to its parent. After the new XPDL node is created, we search and get the attribute nodes in DOM tree. Then the attributes are represented in XPDL nodes as children of it. Figure 5.1 shows the pseudo code of the tree transformation function. The function TREE_TRANSFORMATION() is a recursive function which traverses through all nodes in the tree and creates XPDL nodes copied from DOM node. Figure 5.2 shows the pseudo code for getting attribute nodes from DOM tree. The function GET_ATTRIBUTES() looks up attributes which exist in a DOM node. According to the number of attributes in a DOM node, it creates the corresponding amount of XPDL nodes and copies the attributes name and value from the DOM.
5.2 Microbenchmark Generation and Execution

Figures 5.3 and 5.4 show the pseudo codes for microbenchmark generator and executor respectively. First, the generator generates the headers and the main body of the microbenchmark code. Then, it searches and fetches the user-defined code in the specific directory. The user-defined code is written into the benchmark file. After microbenchmark code is generated, the executor searches for the compiling and executing information in the IR. When the information is ready, the executor compiles and executes microbenchmark code by running compiling and executing commands. Finally, microbenchmark executor returns the result and writes back to IR. Figure 5.5 shows an example of the IR before and after the execution of the microbenchmark executor.
5.3 Code Generation

The implementation of the API code generator starts from getting all of the first level nodes of IR and storing them into a vector, since the first level of nodes in the IR consists of component types which decides which category of code should be produced. Then, we loop through the vector and generate the code according to the type of component that we meet in the current iteration (for example, if we meet xpdl:cpu, we generate code for the CPU component). Figure 5.6 shows the pseudo code of the code generator which handles the code generation.
First, code generator searches for the type of node which is indicated as the parameter component_type of the function and stores it/them into a vector by the function GET_ALL_TARGET_PARENT(). Then, it loops through the vector and searches for the target child node in each iteration and generates the code by WRITE_TO_FILE() according to the value we get from the target node. Figure 5.7 shows an example to generate the code for CPU core components. Since core components are children of CPU components, the core components code will be generated if we meet xpdl:cpu node.

```
1: function CODE_GENERATOR(component_type, target_node, var_type, var_name)
2:  node_list ← GET_ALL_TARGET_PARENT(component_type)
3:  for i = 0 to node_list.length - 1 do
4:     node ← GET_TARGET_NODE(target_node)
5:     var_value ← node.value
6:     WRITE_TO_FILE(var_type, var_name, var_value)
```

Figure 5.6: Pseudo code for code generator

```
1: CODE_GENERATOR(xpdl: core, frequency, double, core_frequency)
2: CODE_GENERATOR(xpdl: core, unit, string, core_frequency_unit)
```

Figure 5.7: Example pseudo code for code generation
6 Result and Discussion

We have presented the design (Chapter 4) and implementation (Chapter 5) of the compiler in previous chapters. In this chapter, we show the result by providing three example programs with executing result in Section 6.2 and then discuss this project in Section 6.3.

6.1 Experiment Environment

The experiment environment is as follows:

1. Hardware:
   
   a) Intel Xeon CPU E5-2630L v2 2.40 GHz
   
      • Number of cores: 24
      • Number of threads on each core: 2
      • L1 cache size: 32K
      • L2 cache size: 256K
      • L3 cache size: 15360K

2. Software:
   
   a) Ubuntu 14.04.4 LTS
   
   b) GCC 5.4.1
   
   c) MeterPU version v0.81
   
   d) Intel Performance Counter Monitor v2.8

6.2 Result

The following are three example programs to demonstrate the use of the API generated by the compiler. Example program 1 is a pthread application to demonstrate how to call the API to get the number of threads in the system. Example program 2 is to show how to get the instruction energy cost which are defined in the system. Example program 3 shows the example of using API to check if the software is installed in the system.
6.2. Result

6.2.1 Example Program 1

In Listing 6.1 we demonstrate a multi-threaded application with the pthread library on a system with 48 hardware threads. We get the total numbers of available threads from the API by calling \texttt{xpdlib::cpu::num\_of\_hardware\_threads}. We run simple compute-intensive functions on each thread, see Listing 6.1. The application is compiled with \texttt{g++ -O0} flag and the execution time is measured by MeterPU. In general, running a compute-intensive program with maximum thread number can get the full usage of CPU.
#include <iostream>
#include <iomanip>
#include <MeterPU.h>
#include <pthread.h>
#include "xpdl.hpp"
using namespace std;

void *thread_func(void *thread_id)
{
    /*some compute-intensive operation*/
    int a=10;
    int b=2;
    int c;
    for(int i=0;i<1000;i++)
    {
        for(int j=0;j<1000;j++)
        {
            for(int k=0;k<1000;k++)
            {
                c=a+b;
            }
        }
    }
}

int main()
{
    using namespace MeterPU;
    Meter<CPU_Time> meter;
    int rc;
    constexpr int num_threads = xpdl::cpu::num_of_hardware_threads;
    cout<<"Number of threads: "<<num_threads<<endl;
    pthread_t threads[num_threads];
    meter.start();
    for(long i=0;i<num_threads;i++)
    {
        rc = pthread_create(&threads[i], NULL, thread_func, (void*)i);
    }
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
    meter.stop();
    meter.calc();
    std::cout<<"[CPU Time Meter] Time consumed is: "
              <<meter.get_value()<<" micro seconds."<<std::endl;
}

Listing 6.1: Example program 1
6.2.2 Example Program 2

Listing 6.2 shows an example program getting the instruction power in the system and Figure 6.2 shows the execution result of it. It shows that we can help programmers to predict or estimate the total energy cost when they run a certain program.

```cpp
#include <iostream>
#include "xpdl.hpp"

/*
int add(int a, int b)
{
    return a+b;
}
int subtract(int a, int b)
{
    return a-b;
}
int multiply(int a, int b)
{
    return a*b;
}
*/

int main()
{
    constexpr double sum_of_energy_cost;
    //instruction_1 : add
    //instruction_2 : subtract
    //instruction_3 : multiply
    sum_of_energy_cost = xpdl::instruction_1::energy_cost +
                        xpdl::instruction_2::energy_cost +
                        xpdl::instruction_3::energy_cost;
    cout<<"Sum of energy cost is: "<<sum_of_energy_cost<<" mJ"<<endl;
}
```

Listing 6.2: Example program 2

Figure 6.1: Result for example program 1

Figure 6.2: Result for example program 2
6.2.3 Example Program 3

Listing 6.3 shows an example program checking if a software is installed in the system and how to get the installed path. Figure 6.3 shows the execution result of it. Since the API is portable, it might be used in other systems which contain the same hardware components. However, the software components might be different. Users need to check the existence of software components before calling them.

```cpp
#include <iostream>
#include <string>
#include "xpdl.hpp"

XDPL_PREPARE_SW_QUERY(xpdl_blas_5_0);

int main()
{
    string installed_path;
    if(xpdl::is_installed<xpdl::system::libraries,
xpdl_blas_5_0>::value)
    {
        cout<<"Installed software is found, path is "
<<xpdl_blas_5_0::path<<endl;
        //RUN_COMPONENT_DEPENDS_ON_BLAS_5_0();
    }
    else
    {
        // SEARCH_OTHER_SOFTWARE();
    }
}
```

Listing 6.3: Example program 3

Figure 6.3: Result for example program 3

6.3 Discussion

We implement the parser of the compiler through Xerces XML Parser. Although it is convenient for us to develop this project, there are still some problems with it. As mentioned in Section 4.1.2, the DOM trees generated by it have redundant text nodes as their children. The other problem is the data types of the DOM nodes. The constraints cause the inconvenience of accessing information in the DOM trees. Thus, we define the simple XPDL node type which contains the name, value, and its children. Despite the fact that it takes one more step to convert the DOM nodes into XPDL nodes, it makes data accessing easier.
An increasing number of recent publications and empirical studies have released the positive contribution that architecture description language (ADL) can make to heterogeneous architectures, see, e.g. [12, 11, 9, 4]. However, they all have their advantages and limitations. In the following sections, we will discuss their main features and make comparisons with XPDL.

7.1 PDL

PDL [12] is an XML-based platform description language for heterogeneous multi-core systems. It is used to generate the platform information to programmers, tools, compilers, etc. In PDL, the hardware properties are presented as key-value pairs in XML and the naming of the attributes are depending on the users’ demands. Although it provides a flexible way to create key-value pairs, it might lead to inconsistencies of presenting the platform properties. To avoid inconsistencies, we normalize the way of interpreting the platform properties in XPDL. In Listing 7.2, we express the same hardware as in Listing 7.1 in XPDL description. The other issue is that PDL tries to interpret the platform from a single huge descriptor which makes the descriptors non-reusable. In contrast, components in XPDL are organized as small models. Each model can be re-referenced by other models if necessary.
7.2. HPP-DL

HPP-DL [11] is a JSON-based hardware platform description language. Its purpose is to support developers and tools to optimize the performance and energy efficiency of heterogeneous platforms. It characterizes the hardware information in a hierarchical manner and also provides an ID attribute to express the relationship between components which is similar to name and type in XPDL.

HPP-DL does not provide a mechanism to group the homogeneous elements like the group in XPDL. In HPP-DL, the CPU in Figure 4.1 will be modeled as one processor block with id=0, four core blocks with id=0 to 3 and five cache blocks with id=0 to 4 which increases the size of the descriptor. Moreover, it does not support power modeling, software modeling, and microbenchmarking.

7.3 hwloc

Hardware Locality (hwloc) [4] is a software tool that fetches the hardware specification from the user operating system and represents the hardware information in a hierarchical manner. However, hwloc does not support all operating systems and only supports specific CPUs and GPUs. In the current stage, XPDL does not support getting hardware information from the operating system.

7.4 ALMA-ADL

ALMA-ADL [14] is an architecture description language to generate the hardware information for Multi-processor System-on-chip (MPSoCs). Its purposes are to support parallelization and simulation. Compared to XPDL [6], PDL [12] and HPP-DL [11], it is based on its own markup language. ALMA-ADL also provides if and for to model the MPSoCs which is similar to group in XPDL. In addition, the descriptors can be converted to XML and
JSON representation. However, it does not support power modeling, software modeling, and microbenchmarking.

### 7.5 Comparison

In Table 7.1 we make a comparison between XPDL, PDL, HPP-DL, ALMA-DL, and hwloc. We can observe that XPDL supports all of the features except fetching information from the operating system.

<table>
<thead>
<tr>
<th></th>
<th>XPDL</th>
<th>PDL</th>
<th>HPP-DL</th>
<th>ALMA-DL</th>
<th>hwloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware modeling</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Software modeling</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Power modeling</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch info. from OS</td>
<td>O</td>
<td></td>
<td></td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Group</td>
<td>O</td>
<td></td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microbenchmark</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Features comparison between XPDL, PDL, HPP-DL, ALMA-DL and hwloc
In this chapter, we conclude the thesis and present the future work.

We have presented the XPDL compiler. The XPDL compiler generates the libraries which implemented the query API. The query API helps users to search and get the hardware and software information in a computer system. Moreover, we introduce metaprogramming in our API, which means there is no overhead to call the API. The XPDL compiler also provides mechanisms to save and load the IR. In addition, the compiler can visualize the IR which help users to understand the IR if necessary.

Despite microbenchmarking provides a good solution to deal with the unknown values in the descriptors, we only implemented a few because of time constraints. In the future, we will design more metric programs to support microbenchmarking.

The variation of the instruction energy costs is high. For managing it, the number of runs should be a parameter, and the microbenchmark executor will return the median value of the results in the future.

In the current stage, the compiler does not support fetching the hardware information from the operating system like hwloc [4]. In the future, we might improve the compiler to support fetching information from the operating system and writing back into the IR.
Bibliography


