Graphical visualization of software pipelined code execution on pipelined and clustered VLIW DSP processors

by

Andrei Mamon

LIU-IDA/LITH-EX-6--10/016--SE

2010-05-01
Final Thesis

Graphical visualization of software pipelined code execution on pipelined and clustered VLIW DSP processors

by

Andrei Mamon

LIU-IDA/LITH-EX-6--10/016--SE

2010-05-01

Supervisor: Mattias Eriksson
Examiner: Christoph Kessler
Acknowledgements

This report describes the structure and implementation details of the visualization framework Optvis which was programmed by Lukas Kemmer as a goal of earlier unfinished thesis project. Lukas has finished with programming of Optvis, but has not written the thesis document describing its implementation and guidelines on how to use the product.

It turned out that Lukas’ code already supports visualizing software pipelined code making the coding part of this project futile. Instead this document will cover both the implementation of software pipelined module and the rest of Optvis to provide a complete document describing the visualizer. Additionally some cosmetic tweaks were done on the existing code to improve the quality of produced visualization.
Abstract

This report follows the development, testing and evaluation of a retargetable compiler visualization framework Optvis which can be used to visualize compiler generated code for different VLIW architectures. Optvis is implemented in C++ programming language and functions as both standalone application and as plug-in for the retargetable compiler framework OPTIMIST.

The purpose of this thesis work is to present the Optvis framework, to give a detailed view over its internal structure and interaction with an end user. Additionally Optvis-OPTIMIST integration is discussed. Testing and evaluation round up the thesis providing a solid ground for the future work.
Contents
1 Introduction ............................................................................................................................. 1
  1.1 Compiler Visualization ...................................................................................................... 1
  1.2 Background ....................................................................................................................... 2
  1.3 Thesis question ................................................................................................................. 2
  1.4 Method and Sources ......................................................................................................... 2
  1.5 Limitations ........................................................................................................................ 3
  1.6 Audience ........................................................................................................................... 3
  1.7 Report Structure ............................................................................................................... 3
2 Theoretical Background ........................................................................................................... 5
  2.1 Compilers .......................................................................................................................... 5
  2.2 VLIW Architecture ............................................................................................................. 6
  2.3 OPTIMIST Framework ....................................................................................................... 8
  2.4 xADML ............................................................................................................................... 9
    2.4.1. xADML Document Structure ..................................................................................... 9
  2.5 Software Pipelining ......................................................................................................... 10
  2.6 CPLEX, AMPL ................................................................................................................... 11
3 Graphical Visualizer ............................................................................................................... 12
  3.1 Input Formats ................................................................................................................. 12
  3.2 Output Formats .............................................................................................................. 13
  3.3 Output Format Modifiers ............................................................................................... 16
  3.4 OPTIMIST – Optvis Interaction ....................................................................................... 17
  3.5 Software Pipelining ......................................................................................................... 19
4 Implementation ..................................................................................................................... 24
  4.1 Visualization Data Classes ............................................................................................... 24
  4.2 Color String Class ............................................................................................................ 24
  4.3 Table Class ...................................................................................................................... 24
  4.4 Visualization Data Transformation ................................................................................. 25
  4.5 XERCES Interaction ......................................................................................................... 26
    4.5.1. The SimpleDomBuilder Class .................................................................................. 26
    4.5.2. The XercesStringWrapper Class .............................................................................. 26
    4.5.3. The DomFromIStream Function ............................................................................. 26
5 Testing .................................................................................................................................... 27
6 Conclusion .............................................................................................................................. 29
7 References ............................................................................................................................. 30


1 Introduction

This thesis report follows the process of extending a visualizer for the retargetable code generator OPTIMIST which produces optimized code for different types of VLIW and DSP processors. In earlier thesis projects it has been made possible for OPTIMIST [3] to produce optimized code for loops. Therefore visualizer will be extended so that it will be able to present a graphical output which shows software pipelined code.

Ever since the Stone Age people were trying to present their thoughts in a way that could be understood by others without having them to know the subject in depth. Cave paintings, Egyptian hieroglyphs, geometry and even Leonardo da Vinci’s drawings were all forms of visualisation of their thoughts and ideas. Visualization makes it possible to pass a message between people with different culture and different language.

Today the need for visualization is growing rapidly. Users tend to prefer graphically rich environments and therefore various applications are developed constantly for science visualization, engineering, multimedia, medicine or software development. Even business data like stock market values is presented today in form of graphs and diagrams in real time. People tend to understand the data that is presented in visual form more quickly, but what is more important is that visual representation of data is processed by the brain differently and more quickly. This phenomena was described by Lane and Kosslyn in their work ”Show Me! What Brain Research Says About Visuals in Power Point” [2].

In computer science visualization is widely used, for example in software engineering by presenting classes, interfaces and their interaction by the means of UML(Unified Modeling Language) [26] diagrams. UML provides a solid ground when you need to specify, visualize, construct, document and understand large software systems. In its essence UML works as a tool that makes it possible for project groups to understand the complex software system as a whole.

1.1 Compiler Visualization

When it comes down to compilers various visualization tools are used. There are many tools that visualize what happens in a compiler’s front end, especially the syntactic analysis phase. Most of them are made for educational purpose. For example, a system was implemented by Resler and Deaver [12] that visualized how scanner and LL(1) parser works when processing a real program. Another tool programmed by Lovato and Kleyn [13] shows an LR parser’s internal operations. There are also tools that visualize SLR and LALR parsers [14, 15].

Not only scanning and parsing is being visualized but other parts of compilers are visualized as well. The semantic analysis and code generation is illustrated in Andrews et al. [16] and Mernik and Zummer [17] works. Later works of Urquiza-Fuentes et al. [18] visualizes the symbol table in the compiler.
There is also some work on visualizing optimization passes that are performed in a compiler when proceeding intermediate representation. Harvey and Tyson [19] implemented a graphical interface for compiler optimizations.

The compiler is a complex system and its underlying work is usually hidden from a user. To provide the means for understanding compiler structure by visualizing its parts can be valuable for education purposes. It also gives an opportunity to quickly check the sanity of the resulting schedule. Additionally visualization can help in finding optimization flaws.

Currently in the OPTIMIST project in order to visualize intermediate representation produced by LCC another tool called VCG [20] which is graph visualizing framework is used. OPTIMIST produces an optimized schedule which is then visualized by Optvis, a visualizing framework presented in this thesis.

1.2 Background

This work has been performed at IDA (Department of Computer and Information Science) with Mattias Eriksson as supervisor and Christoph Kessler as examiner. The thesis was written as 15 hp final thesis.

1.3 Thesis question

OPTIMIST is a retargetable code generator that is being developed and extended as the time goes by in different projects. Before it could only generate optimized code for basic blocks. In recent projects OPTIMIST has been extended with a functionality that now allows it to produce optimized code for loops.

The solution that is produced by OPTIMIST is visualized now by the means of VCG. The project goal is to extend the visualizer so that it can produce visual output for software pipelined code that is generated by OPTIMIST.

Basically this means that the current visualizer code should be extended with a new functionality that provides the possibility of producing visual representation of software pipelined code. The visual representation of software pipelined code will follow the same format as it does now for basic blocks.

1.4 Method and Sources

The OPTIMIST framework is quite complex and consists of many different parts which work together. In order to be able to develop a working solution the framework and its parts should be studied in depth and reviewed one by one. We should clearly understand the underlying structure of the framework, the input files it uses to produce the optimized code and the solution itself. Therefore all work related to the OPTIMIST framework is inspected and studied. In addition various resources on the internet are reviewed as an additional source of information whether the need of supplementary information will take place. Moreover the current visualizer code base is looked into as a part of development process.
The literature that was used to prepare and gain knowledge about the OPTIMIST framework consists of Master and PhD thesis works. The fundamental theory base about the retargetable code generator is presented in Andrzej Bednarski’s PhD thesis “Optimal Integrated Code Generation for Digital Signal Processors.” [3]. This PhD thesis explains the need for a retargetable code generator and its architecture. Additional information regarding the OPTIMIST framework is shown in another master thesis work written by David Landén “ARM9E Processor Specification for OPTIMIST” [4]. The thesis work describes hardware specifications of the ARM9E processor for the retargetable code generator OPTIMIST. Another thesis “Integrated Software Pipelining” by Mattias Eriksson [5] gives a brief introduction into OPTIMIST structure.

In addition XML (Extensible Markup Language) mark-up language syntax and RFC [6] describing it was reviewed. The OPTIMIST framework uses xADML [3] (architecture description markup language) which is covered later in other chapters. The architecture description language is based on XML syntax and therefore the need for understanding its syntax is clear.

Computer architecture books that describe VLIW architecture, especially those that cover advanced VLIW architectures like uniclustered, multiclustered were also studied to gain better knowledge on the subject. By understanding the difference, advantages and disadvantages of each approach it became apparent how they are related to the compilers that generate code for such architectures.

Additionally existing projects that focus on presenting compiler visualizations were looked into. Unfortunately most of them are in-house projects where no details are available to the public. The public ones were used as a source of inspiration, as guidance.

1.5 Limitations

Since the project goal was to extend the current visualizer code with a new functionality some limitations have to be made. The current code base was written in C++ programming language and therefore the extension should follow the same standard and should be programmed in the same language. Currently the visualizer presents OPTIMIST solution as text, html or xml output. There will be no support for graphical representation like OpenGL because it will take too long time to develop. Instead the software pipelined code visualization will follow the same format as it produces now for basic blocks.

1.6 Audience

This report is intended foremost for people with knowledge in Computer Architecture, particularly VLIW architectures, Compilers and Code Generation as well as XML.

1.7 Report Structure

The following chapters are shortly described below.
• 2 Theoretical Background
   The chapter describes compiler principles, VLIW architecture and the OPTIMIST framework
• 3 Graphical Visualizer
   Optvis structure, interaction with a user and OPTIMIST-Optvis cooperation is presented here
• 4 Implementation
   This chapter covers implementation details of the Optvis framework
• 5 Testing
   Various tests which were used to verify the correctness of Optvis output are explained in this chapter
2 Theoretical Background

2.1 Compilers

A compiler is a computer program that transforms human readable source code of another computer program into the machine readable target code that a CPU can execute. Source code is often some sort of high level language like C/C++ which is translated into assembly code for the target architecture.

A source code goes through several phases in which it is being checked for syntax correctness and various optimizations take place as illustrated in the figure below. The code route starts at the gates of the compiler’s front-end where syntax and semantic checking takes place. It is the place where legal and illegal programs are recognized. Type checking is performed here as well. Additionally the front-end performs machine independent optimizations and afterwards generates intermediate representation (IR) which is fed to the compiler’s middle-end. The middle-end welcomes IR stream with additional assortment of optimizations. Here optimizations for performance such as a) removal of useless or unreachable code, b) discovering and propagating constant values, c) relocation of computation to a less frequently executed place (e.g., out of a loop), d) specializing a computation based on the context takes place. Afterwards IR for back-end is generated by the middle-end. The compiler’s back-end task is to translate middle-end IR into the target assembly code. The compiler will therefore map IR instructions to target instructions. In addition the back-end performs machine dependent optimizations based on a given architecture. The back-end tries to utilize the hardware to keep the highest degree of parallelization possible.

![Illustration 1: A structure of a compiler. Adapted from [11]](image)
2.2 VLIW Architecture

Very long instruction word (VLIW) architecture refers to a CPU that has multiple execution units and can execute multiple operations simultaneously (instruction level parallelism). It can also execute different sub-steps of sequential instructions simultaneously (pipelining). In other architectures like superscalar processors increased hardware complexity is required to perform these tasks. These architectures rely on a hardware scheduler that selects instructions from an instruction window and schedules them. Illustration 2 shows how a superscalar CPU relies on the hardware dispatcher in order to schedule instructions efficiently.

Illustration 2: VLIW uses compiler for instruction scheduling. Adapted from [23]

VLIW uses another approach. It has moved the hardware complexity into the software as illustrated in the figure above. A compiler takes care of discovering instruction level parallelism and produces code for the given architecture. As a result VLIW CPU has lower hardware complexity and therefore requires less energy and produces less heat which is very valuable for embedded devices. The reduced hardware complexity makes VLIW CPUs cheaper than superscalars because VLIW chips cost less to design, the design itself takes less time and they may require less debugging.

However the disadvantages are clear. VLIW relies on the compiler to discover instruction level parallelism. Even if the compiler has the ability to look at a much larger window of
instructions VLIW architecture requires a designated compiler which is written especially for a given architecture to perform the optimization.

2.2.1. Unicluster VLIW

Illustration 3 shows that a unicluster VLIW processor contains a number of parallel functional units (FU), a register file (RF) which is shared between the functional units and a network that interconnects functional units and register file. Each functional unit has an access through the network to other functional units and register file. It becomes obvious that the number of wires which is needed to connect all the functional units and the register file increases whenever a new functional unit is added to the CPU. This becomes a stepping stone when it comes down to VLIW performance.

![Illustration 3: Single-cluster(unicluster) VLIW processor architecture](image)

2.2.2. Clustered VLIW

To overcome the burden another approach when designing VLIW CPU is used. As illustrated in figure 4 a single register file is split into several separate register files. Every register file has its own functional units which are connected only to the given register file. In order to transfer the data between register banks and functional units a special network is used. The network can be formed in various formats: ring, fully connected, line, star, tree, bus or mesh depending on the design of clustered VLIW processor. Basically each register file with its functional units becomes a stand-alone working module. Several modules are connected into a network as described above to form the desired clustered VLIW. See figure 4 for an example with two clusters.

![Illustration 4: Two cluster VLIW processor architecture](image)

Nowadays in the world where rapid development of computer technologies takes place target hardware can become obsolete when the compiler is ready because the process of developing a compiler is usually time consuming and expensive. Therefore there exists a need for a compiler that can produce optimized code for different VLIW architectures.
without rewriting the compiler itself. The compiler that follows these qualifications is called retargetable compiler and the following chapter describes it in detail.

2.3 OPTIMIST Framework

OPTIMIST is a research project, whose goal is to provide a retargetable code generator, which produces optimal or highly optimized code for irregular VLIW and DSP architectures. OPTIMIST is a modular system which consists of various blocks that work together to produce optimal code.

Illustration 5: OPTIMIST structure [1]

As a front end it uses LCC [7, 8], a retargetable compiler for ANSI C, which can generate code, e.g. for the ALPHA, SPARC, MIPS R3000, and Intel x86 and its successors. LCC can produce up to 35 different IR-node types and its retargetability is obtained by a bottom up rewriting generator. LCC’s front-end performs lexical and syntax analysis and produces direct acyclic graphs (DAG) which are used as OPTIMIST input. The intermediate representation which is produced by LCC is fed together with an architecture description to OPTIMIST’s optimization engine which performs optimization by the means of:
• Dynamic programming,
• Integer linear programming,
• Heuristic.

The architecture description which is used by retargetable code generator OPTIMIST is called Extended Architecture Description Language (xADML) and is based on XML syntax. The language is capable of describing clustered, pipelined, irregular and asymmetric VLIW architectures. OPTIMIST uses the XML parser Xerces to parse xADML. Currently OPTIMIST supports following architectures: C62x, ARM9E and Motorola MC56K. When OPTIMIST was implemented it could only optimize on basic block level, but lately in other project the ability to produce optimized code for loops (software pipelined code) was added [21]. OPTIMIST can be configured to perform optimization following different criteria: execution time, energy, program size, or register usage.

2.4 xADML

In order to make OPTIMIST a retargetable code generator a specific language has been developed solely for this purpose. The language is called xADML (Extensible Architecture Description Mark-Up Language) and it is based on XML (Extensible Mark-Up Language). xADML describes different types of hardware, the structures of a target processor [3].

2.4.1. xADML Document Structure

An xADML file in its essence can be treated like any other XML file. It starts with a root node called <architecture> and then follow sections that describe specific hardware. Those sections are:

• Hardware resources such as registers, memory modules, etc

• Patterns that are mapped to instructions

• Instruction set

• Transfer section

• Formatting

xADML provides both structural and behavioural description of a target processor. Figure 6 illustrates the structure of a xADML document.
2.5 Software Pipelining

Software pipelining is a compiler loop optimization technique that reforms the loop to achieve faster execution rate by overlapping the executions of iterations.

The following example explains the technique. The basic loop looks like this:

```c
for(i = 0; i < n; i++)
{
    sum += a[i];
}
```

The same loop in pseudo assembly language (here we assume that load is pipelined and takes two cycles, addition takes one cycle and the loop is executed on unclustered VLIW with no transfer latency):

```assembly
L: r1 = L r0
    nop
    r2 = Add r2, r1
    r0 = Add r0, 4 || jump L
```

When unrolled 4 times and registers are allocated the loop transforms into the following code:

```assembly
L: r1 = L r0
    nop
    r2 = Add r2, r1
    r0 = Add r0, 16
    r4 = L r3
    nop
    r2 = Add r2, r4
    r3 = Add r3, 16
    r7 = L r6
    nop
    r2 = Add r2, r7
    r6 = Add r6, 16
```
r10 = L r9
nop
r2 = Add r2, r10
r9 = Add r9, 16 || jump L

Now we can schedule unrolled instructions in parallel:

r1 = L r0
r4 = L r3
r2 = Add r2, r1 || r7 = L r6

r0 = Add r0, 16 || r2 = Add r2, r4 || r10 = L r9
r3 = add r3, 16 || r2 = Add r2, r7 || r1 = L r0
r6 = add r6, 16 || r2 = Add r2, r10 || r4 = L r3
r9 = add r9, 16 || r2 = Add r2, r1 || r7 = L r6

... r0 = Add r0, 16 r2 = Add r2, r4 r10 = L r9
r3 = add r3, 16 r2 = Add r2, r7
r6 = add r6, 16 Add r2, r10
r9 = add r9, 16

The repeating pattern is called kernel – the steady state of pipeline during its work. The repeating pattern can be compressed and the loop becomes:

r1 = L r0
r4 = L r3
r2 = Add r2, r1 || r7 = L r6

r0 = Add r0, 16 || r2 = Add r2, r4 || r10 = L r9
r3 = add r3, 16 || r2 = Add r2, r7 || r1 = L r0
r6 = add r6, 16 || r2 = Add r2, r10 || r4 = L r3
r9 = add r9, 16 || r2 = Add r2, r1 || r7 = L r6

Prolog is a sequence of instructions that fills the pipeline, kernel shows the instructions that are in the pipeline during the steady state and epilog is the phase when pipeline drains.

2.6 CPLEX, AMPL

ILOG CPLEX is an optimization software package. In the context of OPTIMIST it is used in conjunction with mathematical modelling language AMPL to provide another approach to integer linear programming optimization. Instead of generating a schedule OPTIMIST generates an integer linear problem formulation which is used by CPLEX. Afterwards CPLEX optimizes the formulation and produces the solution in the standard AMPL output format.
3 Graphical Visualizer

In earlier project the graphical visualizer called Optvis was implemented. The motivation behind Optvis was to have some sort of visualization tool to aid in various OPTIMIST related projects.

Optvis is written in the C++ programming language. The visualizer has support for OPTIMIST compiler but in general is meant to be standalone program that can be used in conjunction with any VLIW compiler which provides the required data for visualization. Optvis can be statically linked with OPTIMIST. Additionally it works as a standalone application and it supports integer linear programming by the means of parsing a CPLEX solution and generating visual output.

Optvis reads the command line to specify the input and the output parameters as well as format of output. The output is written by Optvis to standard output which makes it possible to include the output in a tool chain. A user can get the list of all input and output options by issuing optvis –help command in the command line. Illustration 7 presents the structure of Optvis framework.

Illustration 7: Optvis structure

3.1 Input Formats

- **XML**
  Visualization data can be loaded from an XML file by issuing optvis –i=”filename” command. The data can be generated by rcc or optvis

- **CPLEX Solution, xADML Architecture Description**
  The solution together with architecture description can be used by Optvis to generate the visualization. Two command line parameters should be provided – optvis –archi=”xADML specification file” –solution=”CPLEX solution file”. The architecture description which is fed to Optvis should be the same as used when generating an integer linear programming solution by CPLEX / OPTIMIST.
3.2 Output Formats

- **XML**
  The output of Optvis is in XML format, because it provides flexibility and follows the same format as xADML. Additionally it can be transformed into XHTML giving the ability to view the schedule in any web browser that supports XML formatting. Moreover XML gives the possibility of parsing the schedule into any other program that has an XML parser, thus making it possible to use the schedule as input for other applications. In order to produce XML output the command line parameter –xml should be specified, for example optvis –xml. Illustration 8 shows the XML file which is generated by Optvis.
Illustration 8: Example of an XML file produced by Optvis
The Optvis generated XML file contains a reference to a XSL stylesheet which makes it possible for web browsers to perform XSLT transformation yielding a viewable XHTML document. XSLT transformation is an operation performed on an XML document using rules described in a XSL stylesheet to produce a XHTML file [22]. The XML file starts with a root node called <visdata>. Functional units are presented within <funits> section where each functional unit is given a description in <funit name="...">/</funit> format. Instructions are described inside <instructions> section, giving each instruction a set of distinctive parameters – its id, cycle when the instruction will be issued, the name of instruction and a transfer parameter. VLIW CPU cycles are shown in <cycles> section of XML file. Each cycle has its number as well as mappings that describe which instructions are performed by which functional unit during that cycle.

- **ASCII**

A simple text table representation is included so that the output can be easily viewable in command line environments. Optvis should be run with –ascii (optvis – ascii) flag to produce the output in this format.

```
<table>
<thead>
<tr>
<th>Cycle Issued</th>
<th>Functional Units</th>
<th>B1</th>
<th>B2</th>
<th>L1</th>
<th>L2</th>
<th>M1</th>
<th>M2</th>
<th>S1</th>
<th>S2</th>
<th>X1</th>
<th>X2</th>
<th>PA</th>
<th>PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADDP4.DIRASYNBOL(6)</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADDP4.L2ERRRBA(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ADDP4.DIRASYNBOL(16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SXL.SERRRECOMM(7)</td>
<td>11</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ADDP4.L2ERRRBA(6)</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ADDP4.DIRASYNBOL(14)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADDP4.DIRASYNBOL(13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>MACI41.M2(2)</td>
<td>S5</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Illustration 9: Optvis produced ASCII table

- **HTML**

Optvis can produce the output in HTML format, including CSS (cascading style sheet) [24] reference in the HTML file to provide more formatting/styling options. The command line parameter is –html (optvis –html). A HTML file is composed as a basic table with <table>, <tr> and <td> tags. This format is widely used and is good for printing.
Illustration 10: Optvis produced HTML file

### 3.3 Output Format Modifiers

The visualizer gives plenty of options to format the output which is produced with flags – html and –ascii. One of the options is to hide registers which are displayed in the two right columns of the table. By specifying –skipregs command those two columns won’t be shown in the output. Another option (-skipfus) allows the user to hide functional units. These options can be used together so that only cycles with instructions will be printed out. There are also two commands that work with data residence : -regcount and –regnoalloc. The first command prints the amount of occupied registers for each residence instead of register mappings. The second one prints instruction ids indicating which values are written to a residence but no register selection. There exists also the possibility to hide the unused registers and functional units thus providing a cleaner and more compact output. Three flags are used to control behaviour : -skipempty, -skipemptyfus and –skipemptyregs. The first flag tells Optvis to hide all unused functional units or registers, the second one hides unused functional units and the third flag is used when there is no need to show unused registers in the output. Additionally there is also a flag that could be used to explicitly specify which registers or functional units should not be shown even if they are not empty. The flag is –
skip="r1, fu1", where r1, fu1 are functional units or registers to skip in the output. These four flags give flexibility. We can also explicitly specify initialization interval for software pipelined code by issuing –II="number" command, thus overriding the parameter in a CPLEX solution.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-skipregs</td>
<td>No register units are shown in the output</td>
</tr>
<tr>
<td>-skipfus</td>
<td>No functional units are shown in the output</td>
</tr>
<tr>
<td>-regcount</td>
<td>Prints number of occupied registers</td>
</tr>
<tr>
<td>-regnoalloc</td>
<td>Prints instruction IDs, which values are written to register file</td>
</tr>
<tr>
<td>-skipempty</td>
<td>Hide all unused functional units and registers</td>
</tr>
<tr>
<td>-skipemptyfus</td>
<td>Hide only empty functional units</td>
</tr>
<tr>
<td>-skipemptyregs</td>
<td>Hide only empty registers</td>
</tr>
<tr>
<td>-ii=&quot;number&quot;</td>
<td>Override initiation interval for software pipelined code</td>
</tr>
</tbody>
</table>

Table 1: Optvis format modifiers

3.4 OPTIMIST – Optvis Interaction

As originally designed Optvis can function as both full-fledged command line application and an OPTIMIST’s extension, but only the command line application supports the whole range of formats and flags. In the OPTIMIST’s extension mode Optvis takes the compiled file and produces the visual output as shown in figure 11.

Illustration 11: Optvis and OPTIMIST integration

The library is integrated with OPTIMIST and uses the generated optimized schedule on the fly. In order to run Optvis via OPTIMIST it should be invoked in the command line as follows:


–target=opt flag tells to choose OPTIMIST as the back-end, -bb selects the basic block which should be compiled, -archi flag describes which architecture file is used. Those three flags are standard for OPTIMIST compilation procedure but the –vis flag is used by Optvis library to determine the format and the file to where the visualization data will be written. The flags
format is format:file where format can be xml, html, ascii, etc and the file is the file name. If file is not specified the output will be written to the standard output. If XML format is chosen the file can be used later as an input to Optvis standalone application to generate visual data.

When using Optvis as a standalone application its input file and other parameters should be specified via command line. As described earlier Optvis can generate visual representation from either earlier XML generated schedule or by inspecting optimized schedule together with architecture description file which is used to produce the schedule. The figure 12 shows this process.

Illustration 12: Optvis as a standalone application

Optvis reads previously generated visualization data by issuing –i flag. The full command is:

optvis –i=visdata.xml –format -flags

Alternatively we can specify CPLEX solution with architecture description:

optvis –solution=solution.sol –archi=architecture.xml –format -flags

Format and flag parameters define which output should be produced. Different formats and flags are described in earlier chapters.
3.5 Software Pipelining

Optvis can visualize software pipelined schedules produced by OPTIMIST. As a running example, a simple dot product calculation code is presented below:

```c
float dot_product(float *a, float *b) {
    int i; float c = 0;
    for (i=0; i<100; i++){
        c+= a[i]*b[i];
    }
    return c;
}
```

Illustration 13: dotp.c source file

The source file is compiled together with an architecture description by OPTIMIST. The retargetable code generator will generate an integer linear programming problem file which is solved by the CPLEX/AMPL module. The solver will produce a solution file with optimized schedule/software pipelined code:

```
c := 
2   MACI41.M2  0 13
4   MACI41.M2  2 13
5   LDI4.D2RBRAIN  0 7
6   ADDP4.L2XRBBRAIN  0 6
7   SHL.S2RBRCOIN  0 5
8   LDI4.D2RBRAIN  0 0
9   ADDRLP4  0 0
10  CNSTI4  0 0
11  LDP4.D1RASBRAIN  0 0
12  ADDRGPA  0 0
13  LDI4.D2RBRAIN  0 8
14  ADDP4.S2XRBBRAIN  0 7
15  LDP4.D1RASBRAIN  0 1
16  ADDRGPA  0 0
;
```

Illustration 14: dotp.c machine instructions

Illustration 12 shows how OPTIMIST translates dotp.c source code file into machine instructions for a given architecture description yielding software pipelined code. The first column represents the node number, the second column is the instruction mnemonic, the third column shows the internal node number and the last column is the cycle when the instruction will be issued. The solution file can be used together with the same architecture file as input to Optvis, thus generating a visualization of the software pipelined code which is presented in illustration 15. Cycle issued column shows the instructions which are issued at specific cycles, functional units column describes the FUs on the target VLIW CPU and RA, RB, ... ,Rn columns are the register banks. There are two register files – RA and RB in our
case. The values which are shown in RA and RB columns are the live ranges of the values computed by the instructions. For example the result of instruction number 7 will appear in register file B on 6th cycle and will stay there even during the 7th cycle, but the value is erased on 8th cycle.

```
Found initiation interval: 3
Num cycles: 19
BuildTable
Vis Kernel: 0
II: 3

| Cycle Initiated | Functional Units | |  |
|-----------------|------------------| |  |
| 0               | LDI4, D2RBSYMBO | 11 8 | | |
|                 | ADDLP4(9) CSTI4(10) | |  |
|                 | LDF4, D1RASYMBOL(11) | |  |
|                 | ADDROP4(12) ADDROP4(16) | |  |
| 1               | LDF4, D1RASYMBOL(15) | 15 | | |
| 2               | | |  |
| 3               | | |  |
| 4               | | |  |
| 5               | SLH.S2RBBCONST(7) | 7 11 | 8 | |
| 6               | ADDP4.L2IBBBRA(6) | 5 11 15 | 7 | |
| 7               | LDI4, D3BBR(5) | 15 | | |
|                 | ADDP4.S2RBBRA(14) | | |
| 9               | LDI4, D3BBR(15) | 13 | | |
| 10              | | |  |
| 11              | | |  |
| 12              | | | 5 |
| 13              | MCI4I.M2(2) MCI4I.M2(4) | | 13 | |
| 14              | | | 2 |
| 15              | | | 2 |
| 16              | | | 2 |
| 17              | | | 2 |
| 18              | | | 2 |
```

Illustration 15: Visualization of software pipelined code on clustered VLIW

Often there exists a need in showing how the software pipelined code fills the pipeline to inspect the efficiency of performed optimizations. For this purpose Optvis provides a flag that changes the presentation of the software pipelined code. Instead of usual output format Optvis can visualize the kernel of the loop. This functionality is obtained by issuing –kernel flag in the command line. The output is changed as follows:
Illustration 16: Kernel visualization on clustered VLIW

The ASCII output presented above is produced using C62x architecture description file with two register files (clustered VLIW) and several functional units. Illustration 17 and 18 shows how the output changes if we perform the same loop on the C62x architecture with a single register file (unicluster VLIW) but the same number of functional units.
As we can see the initiation interval is increased from 3 to 5 and amount of cycles is increased from 19 to 22 when the same schedule is executed on a VLIW with a single register bank. The visualization shows that the same loop is executed faster on clustered VLIW with two register files. The figure 18 shows what happens to the kernel of the loop when executed on a VLIW with a single register file.
<table>
<thead>
<tr>
<th>Cycle Issued</th>
<th>Instruction</th>
<th>Di</th>
<th>L1</th>
<th>M1</th>
<th>S1</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADDRIP4(9)</td>
<td>CNS14(10)</td>
<td>15</td>
<td>15</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADDRGP4(12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LDP4.DIRASYMBO(15)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADDRGP4(15)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LDI4.DIRASYMBO(8)</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>SHL.SIYARACONST(7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ADDI4.L1RARA[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LDP4.DIRASYMBO(11)</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>ADDP4.SIYARAR[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LDT4.DIRARA(5)</td>
<td>5</td>
<td>14</td>
<td>6</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>ADDP4.SIYARAR[14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LDI4.DIRARA(13)</td>
<td>13</td>
<td>4</td>
<td>14</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>MPY.MIYARARA(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Illustration 18: Kernel visualization on unicluster VLIW
4 Implementation

This section covers implementation details that can be valuable for the reader.

4.1 Visualization Data Classes

The visualization data is encapsulated into a C++ class called VisualizationData (visdata.cc, visdata.hh). The class functions as a container that holds in memory deserialized visualization data which can be transformed into different output formats by calling appropriate class functions. HTML representation is obtained by sending the instance of VisualizationData class to VisdataHTMLWriter::write(const VisualizationData& v) function (visdatahtmlwriter.cc, visdatahtmlwriter.hh) which prints output in HTML format. ASCII output is produced by invoking the function VisDataAsciiWriter::write(const VisualizationData& v) (asciiwriter.cc, asciiwriter.hh) provided an instance of VisualizationData class. XML document can be serialized from in-memory representation into file or output stream by XML writer class VisdataXMLWriter (visdataxmlwriter.cc, visdataxmlwriter.hh) which has functions for XML initialization, writing, generating XML tags and closing the XML document.

The instance of the VisualizationData class can be created from two possible sources: XML and CLPEX solution with architecture description file. In order to read XML file the XML parser class (visdataxmlbuilder.cc, visdataxmlbuilder.hh) is used. It parses an XML document and returns an instance of VisualizationData class with deserialized visualization data. Another class called XadmlCplexBuilder (builderxadmlcplex.cc, builderxadmlcplex.hh) is used to parse solution and architecture file. The constructor takes two objects: architecture and solution and reads the data from the objects building an instance of VisualizationData as shown in figure 19.

4.2 Color String Class

A special class (colorstring.cc) with extended functionality based on std::string was created to fulfill the need of various textual representation of the output. The class supports background and foreground color by appending different escape codes depending on the needed color of the string. The class has also overloaded unary and binary operators to simplify the class’s use as well as >> and << operators which allows colored string to be streamed to and from C++ input and output streams, thus making the class very similar to use compared to standard std::string.

4.3 Table Class

Optvis has a general purpose table class (new_tablebuilder.cc) which holds the visualization data in a format similar to HTML table prior to conversion into HTML or ASCII output. Conversion from table into HTML output is performed by printHTML(ostream& o, const Table<colored_string>& t, bool tableOnly, bool prettyPrint)
function (table_to_html.cc). ASCII translation is done by similar function
printAscii( std::ostream& o, const Table<string_t>& t ) (table_to_ascii.cc).

4.4 Visualization Data Transformation

Optvis can read data from an existing XML schedule which was generated earlier. Alternatively it can parse CPLEX solution file and xADML architecture description file. If the first path is taken Optvis will use the XERCES XML parser to deserialize the XML file. During the process it will fill an instance of VisualizationData class with visualization data. The second route uses XadmlCplexBuilder class that parses two files into a single VisualizationData object. An instance of VisualizationData holds now all information about registers, mappings, functional units, instructions and so on. It is transformed into table representation which can be visualized later by sending it to relevant functions. Whenever the user asks for HTML or XML output the table object is converted into a DOMDocument and is serialized. ASCII transformation is somewhat simpler – there is no DOM transformation step. Illustration 19 shows the data flow inside the Optvis framework.
4.5 XERCES Interaction

There are some classes that were employed to simplify interaction with the XERCES XML parser.

4.5.1. The SimpleDomBuilder Class

This class is used to build a DOM tree. It has functions for adding root node, children and attributes. An integer is returned representing the node id when the node is added. Once a DOM tree is created it can be sent to XERCES as DOMDocument.

4.5.2. The XercesStringWrapper Class

To simplify the use of XERCES strings, XMLCh*, the XercesStringWrapper class was implemented. The class functions as a wrapper for XMLCh* or const char* and functions more or less like std::string but its functionality is limited to comparisons, streaming out and conversions to XERCES string format.

4.5.3. The DomFromIStream Function

This function parses a serialized XML document from istream returning XERCES DOMDocument which can be transformed into VisualizationDate by XERCES XML functions.
5 Testing

Various tests were created to verify the correctness of implementation. A simple C++ test program was written for every crucial class or function. Program’s task was to test the class’s or function’s functionality. The programs were compiled and included into two different shell scripts which run the programs and compare the output with expected one. The first shell script’s task was to check the vital classes which use dynamic memory allocation for memory leaks, see figure 20.

Illustration 20: Memory tests

Second shell script checks for differences in output produced by the test programs and signals if the output doesn’t match with the one expected, see figure 21.

Illustration 21: Unit tests

Additionally it can be valuable to estimate the time needed to generate the visualization. In order to achieve this the command `time command` is issued in the unix command prompt yielding the elapsed time between invocation of `command` and its termination.

The following results were obtained in a consecutive set of 10 invocations: for the unicluster visualization presented in Section 3.5 it takes 0,0224s on average to generate the visualization. The time it takes to generate visualization for clustered architecture is higher - 0,0386s on average. The reason behind this is that it takes more time to parse bigger xADML architecture description file. The file used in unicluster example has around 40% fewer lines than the xADML file for clustered architecture used in Section 3.5. Illustration 22 shows the
distribution of values obtained by issuing `time` command to estimate the time needed to produce the visualization.

Illustration 22: Time estimated for unicluster and clustered visualizations
6 Conclusion

During an earlier unfinished thesis work a retargetable visualization framework for VLIW compilers called Optvis was implemented. This document describes it and other techniques which are of importance. Optvis is a retargetable framework which can be used in conjunction with any VLIW compiler. Optvis is an application which can function as both standalone executable and OPTIMIST plug-in thus giving the end user flexibility. The visualizing framework supports various output formats - XML, HTML and ASCII. Because of modular design Optvis can be easily extended to support new output formats. Additionally Optvis has a number of flags which can be used to modify the output to fulfil the end user demands on how the data should be presented. The time needed to produce the visualization is usually insignificant, Optvis produces the visual data in less than a second. To summarize this Optvis is a flexible solution aimed at providing a visual representation of compiler generated code on VLIW architectures.

Optvis was implemented in C++ programming language which is not the optimal language when it comes down to programming visualizations because the C++’s efficiency is not needed in this case. Additionally it takes more time for someone to understand one’s C++ code compared to Java or other object oriented language. On the other hand OPTIMIST is written in C/C++ and by writing Optvis in the same language they can be easily integrated.
7 References

1. OPTIMIST Framework homepage
   http://www.ida.liu.se/~chrke/optimist/

   http://www.indezine.com/articles/whatbrainresearchsays01.html

   http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-6568


   http://www.w3.org/TR/2004/REC-xml-20040204/

6. Introduction to VLIW computer architecture, 9397-750-01759


10. CPLEX Homepage


20. VCG, Visualization of Compiler Graphs


22. XSLT Transformation
   http://www.w3schools.com/xsl/xsl_transformation.asp


