HW/SW Codesignflow with LLVM
including a simple LLVM-VHDL backend

Tim Sander - ICS, Technische Universität Darmstadt,
Aditya Vishnubhotla Vijay - EED, IIT ROORKEE,
Sorin A. Huss - ICS, Technische Universität Darmstadt
Overview
Table of Contents

Overview
  Motivation
  Designflow
  Designflow Part II
  Binning pass
  Filter pass
  linking

Hardware
  Filter pass cont.

VHDL Backend
  Example
  Memory access
  Extracting hw addresses
  Merging software and hardware

Results
  Numbers

Conclusion
  To do

Resumé
Overview

Motivation

Setting:
Part of a research effort to create a hardware/software environment which allows a computing element switch. This is a switch between a processor and a reconfigurable hardware element.

- Have a common algorithm description (LLVM).
- Find out if the llvm optimization passes may be leveraged for use in hardware.
- Test the hardware backend on the easy parts i.e. datadriven parts use software for the rest.
- Have a powerful and easy extensible platform.

The rest of this talk will focus on the design and implementation of proof of concept implementation.
Use LLVM framework as to produce input.
Use LLVM framework as to produce input.
Use optimizations from LLVM.
Overview
Designflow

- Use LLVM framework as to produce input.
- Use optimizations from LLVM.
- Estimate properties of basic blocks to decide if they may be run in software or hardware.
Overview
Designflow

- Use LLVM framework as to produce input.
- Use optimizations from LLVM.
- Estimate properties of basic blocks to decide if they may be run in software or hardware.
- Store optimized version of bytecode as multiple starting points for the generation of software and hardware.
Create software part based on the saved LLVM bytecode. Filter out all functionality of hardware blocks. Insert \(\text{migration}\) LLVM-intrinsics which define the interface between software and hardware.
Create software part based on the saved LLVM bytecode. Filter out all functionality of hardware blocks. Insert «migration» LLVM-intrinsics which define the interface between software and hardware.

The interface instructions are lowered as function calls and are linked in with the software.
Overview
Designflow

▶ Create software part based on the saved LLVM bytecode. Filter out all functionality of hardware blocks. Insert «migration» LLVM-intrinsics which define the interface between software and hardware.

▶ The interface instructions are lowered as function calls and are linked in with the software.

▶ The hardware part is generated utilizing the VHDL backend.
Overview

Designflow

- Create software part based on the saved LLVM bytecode. Filter out all functionality of hardware blocks. Insert «migration» LLVM-intrinsics which define the interface between software and hardware.

- The interface instructions are lowered as function calls and are linked in with the software.

- The hardware part is generated utilizing the VHDL backend.

- Memory addresses are extracted from the software. The hardware design flow is based xilinx tools.
Overview

Migration Points

- The migration points define the interface between hard and software.
- Keep the dataflow defined.
- Define the interface between software and hardware.
- Seems to be an elegant way to interface between software and hardware.
- Hardware and software designflow use the migration intrinsics mutually for the basic blocks.
Overview
Migration Points

- The migration points define the interface between hard and software.
- Keep the dataflow defined.
- Define the interface between software and hardware.
- Seems to be an elegant way to interface between software and hardware.
- Hardware and software designflow use the migration intrinsics mutually for the basic blocks.
Overview

Hardware

- For memory accesses the PLB bus from IBM and used by Xilinx for their Platform Studio”.
- The hardware generated uses a unified memory architecture shared with the PowerPC processor.
- The hardware description in VHDL is generated similar to the LLVM C-Backend.

Xilinx University Program (XUP) board
The filter pass uses heuristics to decide which basic blocks are software and hardware.

**Basic Block Level Analysis**

- **Control flow features**
  - Terminator instructions
  - Call instructions
  - Loops

- **Data flow features**
  - Arithmetic instructions
  - Absence of PHI instructions

**Function Level Analysis**

- **Control flow features**
  - Edges of function CFG
  - Nodes of function CFG
  - Control flow basic blocks

- **Data flow features**
  - Function size
  - Data flow basic blocks

Currently only partitioning with basic block works.
Filter pass II
Pass details

The filter pass inserts migrate intrinsics:

migrate_begin intrinsic

- Replaces code not executed.
- Variable argument function
- Each Argument represents a Basic Block data dependency.

migrate_end intrinsic

- Represents outgoing data dependencies.
- One statement for each dependency.

Special handling for software side

PHI and Terminator instructions are not deleted.
Filter Example
Software part

```c
volatile int u=1,v=2;
int __attribute__((noinline)) blub(int a, int b) {
    int x=a+b;
x+=a*b;
x+=x*a;
x+=x*b;
    return x;
}
```

Software side (edited to fit on page)

```c
define i32 @blub(i32 %a, i32 %b) nounwind {
    entry:
%migrate_begin = call i32 (...)*
        @llvm.migrate_begin(i32 0, i32 2, i32 %a, i32 %b);
%migrate_end = call i32
        @llvm.migrate_end_int.i32(i32 0, i32 6);
    ret i32 %migrate_end
}
```
The PowerPC assembly has migrate_begin and migrate_end intrinsics lowered as function calls.

**Code of migrate_begin**

```c
unsigned int migrate_begin(unsigned int mb_enum,
                           unsigned int numargs, ... ){
    long int i;
    unsigned int offset = 100; //data register offset address
    va_list ap; //pointer to variable argument list
    va_start(ap,numargs); //Initialise the argument list
    for (i=0;i<numargs;++i){
        Datum = va_arg(ap, Xuint32);
        //send values to a new hardware location
        *((unsigned int*)(rbaddr+offset))=Datum;
        offset +=4;
    }
    va_end(ap);
    return mb_enum;
}
```
Filter pass example

Hardware part

```c
volatile int u=1,v=2;
int __attribute__((noinline)) blub(int a, int b) {
    int x=a+b;
    x+=a*b;
    x+=x*a;
    x+=x*b;
    return x;
}
```

Hardware side (edited to fit on page)

```assembly
define i32 @blub(i32 %a, i32 %b) nounwind {
    entry:
        %tmp6 = mul i32 %b, %a
        %tmp3 = add i32 %b, %a
        %tmp8 = add i32 %tmp3, %tmp6
        %tmp11 = mul i32 %tmp8, %a
        %tmp13 = add i32 %tmp11, %tmp8
        %tmp16 = mul i32 %tmp13, %b
        %tmp18 = add i32 %tmp16, %tmp13
        ret i32 %tmp18
}
```
The VHDL Backend starts off from the same optimized version as the software side.

**Basic block based VHDL backend**

- Can only process sequential code.
- State machine represented by a shift register (one hot coding).
- Memory accesses asynchronous ⇒ break shift register
- One to one mapping ⇒ uses lots of hardware resources
- Blocks containing migrate intrinsics are not created.
- The hardware is automatically connected to the PLB bus.
- The memory access implemented using a unified memory architecture.
architecture LLVM_VHDL of blub_entry is

signal entry_sync_0_5: std_logic_vector( 0 to 6-1);
signal tmp6: std_logic_vector(32-1 downto 0); -- mul
signal tmp3: std_logic_vector(32-1 downto 0); -- add

begin
    process (clk) is
    begin
        if (clk 'event and clk='1') then
            if (entry_sync_0_5(0)= '1') then
                tmp6 <= STD_LOGIC_VECTOR(UNSIGNED(b) * UNSIGNED(a));
                tmp3 <= STD_LOGIC_VECTOR(UNSIGNED(b) + UNSIGNED(a));
            end if;
            if (entry_sync_0_5(5)= '1') then
                tmp18 <= STD_LOGIC_VECTOR(UNSIGNED(tmp16) + UNSIGNED(tmp13));
            end if;
        end if;
    end process;
end architecture;
Based on work from Marc Stöttinger.

- For each memory access a separate port is created.
- Each of these ports is connected via a multiplexer to the bus interface.
- The bus interface fetches the data and signals ready signal back to the hardware.
- Bus interface is a slave device for easier debugging.
Extracting hw addresses

Details

Software and Hardware part have to agree on the addresses used for memory access.

▶ Currently the software runs in realmode.
▶ The addresses are extracted with the binutil „nm“.
▶ They are merged with „sed“.
▶ Process is fragile due to the reliance on symbol names.
▶ Work in progress.
Merging software and hardware
Details

The creation of the complete bitstream uses the xilinx design flow with some exceptions:

- The hardware is created and added automatically to an existing project.
- Xilinx sw interrupts to slow $\Rightarrow$ custom assembly interrupt handler.
- Thus the software project uses own start files and a custom linker script.

The created bitstream is then programmed into the XUP board.
## HDL Synthesis Report

### Macro Statistics incl. PLB interface

<table>
<thead>
<tr>
<th># ROMs</th>
<th>1-bit register</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x2-bit ROM</td>
<td>2-bit register</td>
<td>3</td>
</tr>
<tr>
<td># Multipliers</td>
<td>3-bit register</td>
<td>2</td>
</tr>
<tr>
<td>32x32-bit multiplier</td>
<td>32-bit register</td>
<td>16</td>
</tr>
<tr>
<td># Adders/Subtractors</td>
<td>4-bit register</td>
<td>2</td>
</tr>
<tr>
<td>32-bit adder</td>
<td>64-bit register</td>
<td>3</td>
</tr>
<tr>
<td># Counters</td>
<td>8-bit register</td>
<td>2</td>
</tr>
<tr>
<td>2-bit up counter</td>
<td># Multiplexers</td>
<td>1</td>
</tr>
<tr>
<td># Registers</td>
<td>32-bit 4-to-1 multiplexer</td>
<td>1</td>
</tr>
</tbody>
</table>
## Results XST synthesis

### Resource usage II

<table>
<thead>
<tr>
<th>Advanced Macro Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># FSMs</td>
<td>3</td>
</tr>
<tr>
<td># ROMs</td>
<td>1</td>
</tr>
<tr>
<td>4x2-bit ROM</td>
<td>1</td>
</tr>
<tr>
<td># Multipliers</td>
<td>3</td>
</tr>
<tr>
<td>32x32-bit registered multiplier</td>
<td>3</td>
</tr>
<tr>
<td># Adders/Subtractors</td>
<td>4</td>
</tr>
<tr>
<td>32-bit adder</td>
<td>4</td>
</tr>
<tr>
<td># Counters</td>
<td>1</td>
</tr>
<tr>
<td>2-bit up counter</td>
<td>1</td>
</tr>
<tr>
<td># Registers</td>
<td>858</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>858</td>
</tr>
<tr>
<td># Multiplexers</td>
<td>1</td>
</tr>
<tr>
<td>32-bit 4-to-1 multiplexer</td>
<td>1</td>
</tr>
</tbody>
</table>

### Resource usage for Device:

- 2vp30ff896-7
  - #Slices 228 of 13696 1%
  - #Slice Flip Flops 362 of 27392 1%
  - #4 input LUTs 245 of 27392 0%
  - #IOs 218
  - #bonded IOBs 0 of 556 0%
To do

- fix missing stuff
- create dedicated optimization passes suitable for hardware implementation.
- optimized hardware backend (in the making)
- fully migratable algorithms (partially in the making)
- operating system which may utilize the «free cpu power»
Goals:

- automatic hw/sw codesign flow out of llvm
- intrinsics
- ppc backend
- hw backend
- tool integration

Stuff not covered but might be interesting to llvm community:

- optimization pass which handles constant arrays for hardware backend.
- much more efficient hardware pass which uses genetic algorithm for multidimensional optimizations.
Thanks for listening
Design problems
HW backend

There is no intermediate representation between llvm and writeout ⇒ Problems.

▶ Design has been adapted from another implementation.
▶ Writeout does not occur from well defined places.
▶ This leads to code duplication which is bad for maintainability.

Proposed solution

▶ Create intermediate "wire representation.
▶ Use visitor classes for writeout.
▶ Create intermediate representation out of LLVM IR.
LLVM framework

Problems

▶ quite large (but good documentation)
▶ signedness not easy to resolve?
▶ no optimisation passes for hw (obviously)
▶ no support for «mixed» backends.

Idea: create framework for multiple backends

▶ interface for dividing input algorithm
▶ automatic creation of interface
▶ probably even creation of hw and sw thinkable