AhirV2: from algorithms to hardware

Madhav Desai
Department of Electrical Engineering
Indian Institute of Technology
Mumbai 400076 India

April 10, 2011

1 Introduction

We describe the essential features of the second version of the AhirV2 toolset developed at IIT-Bombay.

2 What is AhirV2?

Perhaps the simplest way to understand AhirV2 is through an example.

Suppose we start with a simple program.

```
int add(int a, int b)
{
        int c = (a+b);
        return(c);
}
```

This program describes an algorithm. We normally compile this program and convert it to a machine-level program which is executed on a processor.

What if we wished to convert this program to a circuit? Such a circuit would have inputs a, b and an output c (together with some interface handshake signals). When activated, the circuit should read its inputs and compute a response which is the same as would be expected if the program were executed on a computer.

AhirV2 consists of a set of tools which takes a C/C++ program and converts this collection to a hardware circuit (described in VHDL). This conversion is done in two steps:

- The high-level program is compiled to an interemediate assembly form. AhirV2 introduces an intermediate assembly language **Aa** which can serve as a target for sequential programming languages (such as C/C++) as well as for parallel programming languages.

- From the **Aa** description, a virtual circuit (described in a virtual circuit description language **vC**) is generated. The chief optimization carried out at this step is that decomposing the system memory into disjoint segments based on usage analysis (this considerably improves the available memory bandwidth and reduces system cost).
- From the **vC** description, a VHDL description of the system is generated. The chief optimization carried out at this stage is resource sharing. The **vC** description is analyzed to identify operations which cannot be concurrently active and this information is used to reduce the hardware required.
- The VHDL description produced from **vC** is in terms of a library of VHDL design units which has been developed as part of the AhirV2 effort. This library consists of control-flow elements, data-path elements and memory elements.
- For the C/C++ to **Aa** translation, the AhirV2 toolset uses the front-end C/C++ compiler infrastructure developed by the LLVM project (www.llvm.org), which compiles the C/C++ program to LLVM byte-code (which is a processor independent representation of the compiled program). The AhirV2 toolset has a utility which translates the LLVM byte-code to an **Aa** description.

In Section 4, we illustrate the use of these tools on the simple example shown above.

The earlier version of the AHIR flow has been used to convert non-trivial programs to hardware. The resulting circuits are upto two orders of magnitude more energy-efficient than a processor [1]. The AhirV2 toolset incorporates several optimizations which make the resulting circuits more competitive.

3 The tools

We assume that you have access to either **llvm-gcc** of **clang** as the front-end compiler which generates LLVM byte-code from C/C++. The current AhirV2 toolset is consistent with llvm 2.8 and clang 2.8.

The other tools in the chain are described below.

3.1 llvm2aa

This tool takes LLVM byte code and converts it into an **Aa** file.

```
llvm2aa bytecode.o > bytecode.aa
```

The generated **Aa** code is sent to **stdout** and all informational messages are sent to **stderr**. On success, the tool returns 0.

3.2 Aa2VC

This tool takes a list of **Aa** programs and converts them to a **vC** description.

```
Aa2VC [-O] file1.aa file2.aa ...  > result.vc
```

The generated **vC** code is sent to **stdout** and all informational messages are sent to **stderr**. On success the tool returns 0.

The tool performs memory space decomposition and if the **-O** option is specified, it also attempts to parallelize sequential code using dependency analysis.

3.3 vc2vhdl

Takes a collection of **vC** descriptions and converts them to VHDL.

```
vc2vhdl -t foo [-t bar -t bar2 ...]  -f file1.vc -f file2.vc ...  > system.vhdl
```

Using the **-t** option, one can specify the modules which are to be accessible from the ports of the generated VHDL system. Using the **-f** option, one specifies the **vC** files to be analyzed.

The tool performs concurrency analysis to determine operations which can be mapped to the same physical operator without the need for arbitration. It also instantiates separate memory subsystems for the disjoint memory spaces (in practice many of the memory spaces are small and are converted to register banks).

3.4 Miscellaneous: vcFormat and vhdlFormat

The outputs produces by **Aa2VC** and **vc2vhdl** are not well formatted. One can format **Aa** and **vC** files using **vcFormat** as follows

```
vcFormat < unformatted-vc/aa-file  > formatted-vc/aa-file
```

and similarly use **vhdlFormat** to format generated VHDL file.

4 An example

Let us revisit the simple example considered in the first section:

```
int add(int a, int b)
{
        int c = (a+b);
        return(c);
}
```

We wish to generate a circuit which *implements* the specification implied by this program.

We convert the program to LLVM byte code using the **clang** compiler (www.llvm.org)

```
clang -std=gnu89 -emit-llvm -c add.c
```

This produces a binary file **add.o** which is the LLVM byte-code. To make the byte-code human readable, we dis-assemble it using an LLVM utility

```
llvm-dis add.o
```

This is what the LLVM assembly code looks like

```
; ModuleID = 'add.o'
target datalayout = "e-p ..... "
target triple = "i386-pc-linux-gnu"

define i32 @add(i32 %a, i32 %b) nounwind {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %c, align 4
  %6 = load i32* %c, align 4
  ret i32 %6
}
```

To get to this point, we could have used several optimizations which are available in the LLVM frame-work. But we work with the unoptimized version to illustrate the storage decomposition which is carried out by the AhirV2 tools.

The LLVM byte-code is our starting point. We first convert it to **Aa** .

```
llvm2aa add.o | vcFormat > add.o.aa
```

This produces an **Aa** program

```
// Aa code produced by llvm2aa (version 1.0)
$module [add]
// arguments
$in (a : $uint<32> b : $uint<32> )
$out (ret_val__ : $uint<32>)
$is
{
  $storage stored_ret_val__ : $uint<32>
  $branchblock [add]
  {
    //begin: basic-block bb_0
    $storage iNsTr_0 : $uint<32>
    $storage iNsTr_1 : $uint<32>
```

```
    $storage c : $uint<32>
    iNsTr_0 := a
    iNsTr_1 := b
    // load
    iNsTr_4 := iNsTr_0
    // load
    iNsTr_5 := iNsTr_1
    iNsTr_6 := (iNsTr_4 + iNsTr_5)
    c := iNsTr_6
    // load
    iNsTr_8 := c
    stored_ret_val__ := iNsTr_8
    $place [return__]
    $merge return__ $endmerge
    ret_val__ := stored_ret_val__
  }
}
```

Now, this **Aa** code is converted to a virtual circuit **vC** representation.

```
Aa2VC -O add.o.aa | vcFormat > add.o.aa.vc
```

The virtual circuit representation is a bit too verbose to reproduce entirely here, but we show some critical fragments

```
$module [add]
{
  $in a:$int<32> b:$int<32>
  $out ret_val__:$int<32>
  $memoryspace [memory_space_0]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // ret-val is kept here
    $object [xxaddxxstored_ret_val__] : $int<32>
  }
  $memoryspace [memory_space_1]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // a is kept here.
    $object [xxaddxxaddxxiNsTr_0] : $int<32>
  }
  $memoryspace [memory_space_2]
  {
```

```
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // b is kept her
    $object [xxaddxxaddxxiNsTr_1] : $int<32>
  }
  $memoryspace [memory_space_3]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // c is kept here.
    $object [xxaddxxaddxxc] : $int<32>
  }
  $CP
  {
     // a control-flow petri-net..  verbose..
  }
  // end control-path
  $DP
  {
     // wires and operators.
  }

   // links between CP and DP
}
```

The important points to note are that the stored objects a,b,c and ret_val__ are mapped to different memory spaces. Thus, the chief difference between a **vC** description and a processor is that the **vC** program partitions storage into small units which are accessed only by operators that need them.

Finally, we take the **vC** description and convert it to VHDL

```
vc2vhdl -t add -f add.o.aa.vc | vhdlFormat > system.vhdl
```

This produces a VHDL implementation of the system with **add** marked as a top-level module. The VHDL that is produced is too voluminous to reproduce here, but the top-level system entity is

```
entity test_system is  -- system
  port (--
    add_a : in  std_logic_vector(31 downto 0);
    add_b : in  std_logic_vector(31 downto 0);
    add_ret_val_x_x : out  std_logic_vector(31 downto 0);
    add_tag_in: in std_logic_vector(0 downto 0);
    add_tag_out: out std_logic_vector(0 downto 0);
    add_start : in std_logic;
```

```
    add_fin   : out std_logic;
    clk : in std_logic;
    reset : in std_logic); --
  --
end entity;
```

There are ports corresponding to the arguments of the top-level module, and the add_start/add_fin is a handshake pair. One sets of the inputs, starts the system and waits until the fin is asserted. After the fin is asserted, one has the return value of at the appropriate port.

5 Handling complex programs

This example is trivial. Real, non-trivial programs can be mapped in this manner. Currently, there are only two restrictions

- No recursion, no cycles in the call-graph of the original program.
- No function pointers.

The AhirV2 flow produces a modular system, and by default, produces one VHDL entity for every function in the original program. Further, the concept of message pipes is native to the **Aa** and **vC** descriptions. Thus, it is easy to map parallel programs or programs consisting of multiple concurrent processes to hardware.

6 To be continued

This document is a work-in-progress. More details and related documentation will be added shortly.

The examples folder contains some explanations which might make things easier to understand.

References

[1] Sameer D. Sahasrabuddhe, Sreenivas Subramanian, Kunal P. Ghosh, Kavi Arya, Madhav P. Desai, "A C-to-RTL Flow as an Energy Efficient Alternative to Embedded Processors in Digital Systems," DSD, pp.147-154, 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, 2010