AhirV2: from algorithms to hardware
An overview

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

March 7, 2018

1 What is AhirV2?

AhirV2 is a set of tools which can convert a C description of a system to an equivalent hardware implementation (described in VHDL). Using these tools, it is possible to take an algorithmic approach to the design of hardware.

The flow of transformations is illustrated in Figure 1.

- Given a high-level C, we rely on an LLVM (www.llvm.org) compatible compiler such as clang (www.clang.org) to produce LLVM byte code. Currently, the AhirV2 flow uses LLVM byte code as a starting point.
- The LLVM byte-code program is compiled to an intermediate assembly form. AhirV2 introduces an intermediate assembly language **Aa** which serves as a target for sequential programming languages (such as C) as well as for parallel programming languages. An **Aa** program consists of modules (analogous to sub-programs in C) which can call each other, and can communicate through storage objects as well as through pipes (first-in-first-out buffers).
- From the **Aa** description, a virtual circuit (described in a virtual circuit description language **vC**) is generated. The chief optimizations carried out at this step are dependency based operation ordering, dynamic

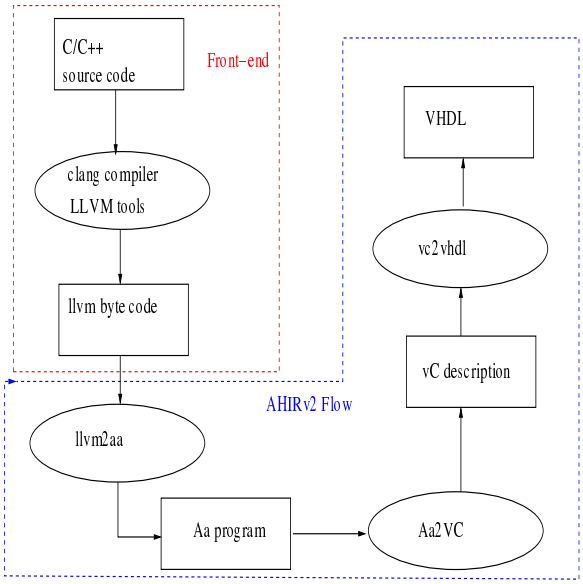

Figure 1: AhirV2 flow

loop-pipelining and decomposition of the system memory into disjoint spaces based on static pointer analysis (this considerably improves the available memory bandwidth and reduces system cost). A **vC** description also consists of modules: however, the modules are presented in a factored form (control X data X storage).

- From the **vC** description, a VHDL description of the system is generated. The system consists of modules, memory spaces and FIFO buffers. The modules are further broken down into a control-path (a live and safe Petri net), and a data-path (a graph of operators and wires). 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.

Thus, to generate hardware using the AhirV2 flow, it is possible to start

at the C-level, at the **Aa** level at the **vC** level or at the VHDL level (or a combination of all these levels). Starting at a higher level is easier for the programmer, but using lower level representations will usually lead to more efficient hardware. Typically, if we start from C, circuits produced by the AhirV2 flow are upto two orders of magnitude more energy-efficient than a processor [1].

Currently, there are only two restrictions in mapping a C program to VHDL using the AhirV2 flow:

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

2 Use Scenario 1: a transformational system

Many programs are transformational in nature. There is some input data X, and the program can be modeled as a function f which acts on X to produce output data Y.

2.1 A trivial example

Consider the following trivial example:

```
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
}
```

Note 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 ahir_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_req : in std_logic;
    add_start_ack : out std_logic;
    add_fin_req   : in std_logic;
    add_fin_ack   : out std_logic;
    clk : in std_logic;
    reset : in std_logic); --
  --
end entity;
```

The AHIR system is a purely synchronous implementation which uses only the rising edge of the **clk** input. The **reset** is synchronous, and is active high.

The VHDL implementation of a module in the AHIR system corresponds to that of a pipeline stage (see Figure 2). The ports

```
add_a add_b
```

correspond to the input arguments of the top-level function **add**. The port

```
ret_val_x_x
```

corresponds to the value returned by **add**. The pair of ports

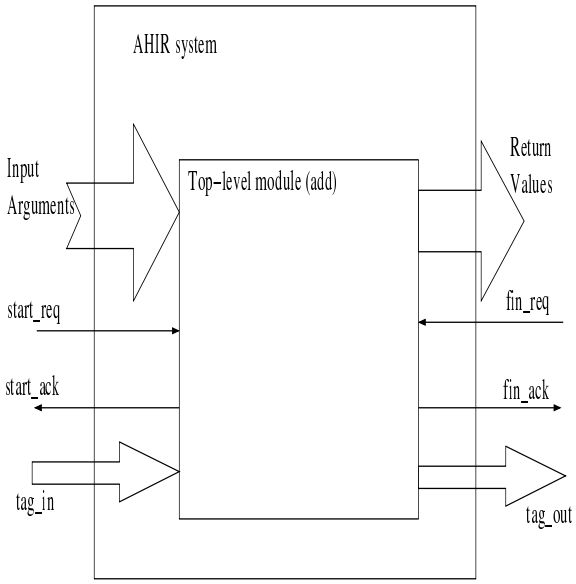

Figure 2: System and Module Interfaces

`add_start_req` `add_start_ack`

implement a **start** protocol; The environment asserts add_start_req when it wants to start **add**, and the AHIR system asserts **add_start_ack** whenever it is ready to start. The environment is required to hold the input arguments steady until it observes the acknowledge from the AHIR system. The pair of ports

`add_fin_req` `add_fin_ack`

implement a **finish** protocol; The environment asserts add_fin_req when it is in a position to accept the returned value from a previously started **add**, and the AHIR system asserts **add_fin_ack** whenever it has a return-value avaiable. The returned value is valid only when the acknowledge from the AHIR system is asserted. The ports

`add_tag_in` `add_tag_out`

provide a mechanism by which a tag can be presented by the environment to the AHIR system to identify a particular invocation of the **add** function. The width of the tag is chosen (by the AhirV2 tool-chain, specifically, the vc2vhdl tool) to be large enough that each active call to **add** can be identified uniquely.

3 Use Scenario 2: pipelines

A pipeline is a collection of parallel processes which work together to accomplish a certain function (or to finish a job). Pipelines are commonly used in software and in hardware systems.

The concept of pipes or sockets provide a natural mechanism for communication between parallel processes in a software pipeline. In **Aa** (and **vC**), pipes are first-in-first-out buffers which provide a corresponding communication construct in the generated hardware.

3.1 An example of a trivial pipeline

Consider the following pipeline which has two stages **foo** and **bar**:

```
Environment --> foo --> bar --> Environment
```

The stage **foo** takes a 32-bit integer from the external world, complements it and passes it on to stage bar. The stage **bar** takes the 32-bit number from **foo**, complements it and sends it to the external world (in effect, nothing useful is done, this is only an illustration).

If we are writing this pipe-line as a program, we could implement two independent processes (or threads) and use named pipes to perform the communication between these processes/threads (and the "outside world). For example:

```
#include <iolib.h>
void foo()
{
   while(1)
   {
     uint32_t data = read_uint32("inpipe");
     write_uint32("midpipe", ~data);
   }
}

void bar()
{
   while(1)
   {
```

```
    uint32_t data = read_uint32("midpipe");
    write_uint32("outpipe", ~data);
  }
}
```

In this example, the read/write functions are provided as part of a *pipeHandler* library which is bundled with the AhirV2 distribution.

The two functions can be compiled into separate processes or can be used in threads in a multi-threaded program (using pthreads, for example), and one gets a software pipeline, in which the inter-process communication is done using the read/write function calls, which use named pipes.

One can also map this pipeline to a VHDL system, using the following flow. One uses the standard flow that has already been described:

```
clang -std=gnu89 -I../../../iolib/ -emit-llvm -c prog.c
llvm2aa prog.o | vcFormat >  prog.o.aa
Aa2VC -O -I mempool -C prog.o.aa | vcFormat > prog.o.aa.vc
vc2vhdl -C -s ghdl -T foo -T bar -f prog.o.aa.vc\
           | vhdlFormat > system.vhdl
```

Note that in this case, we use the -T option in vc2vhdl to specify that foo and bar are free-running top-level modules. The resulting AHIR system in system.vhdl implements the pipeline (the block diagram of the system is shown in Figure 3) with the following interface:

```
entity ahir_system is  -- system
  port (--
    clk : in std_logic;
    reset : in std_logic;
    inpipe_pipe_write_data: in std_logic_vector(31 downto 0);
    inpipe_pipe_write_req : in std_logic_vector(0 downto 0);
    inpipe_pipe_write_ack : out std_logic_vector(0 downto 0);
    outpipe_pipe_read_data: out std_logic_vector(31 downto 0);
    outpipe_pipe_read_req : in std_logic_vector(0 downto 0);
    outpipe_pipe_read_ack : out std_logic_vector(0 downto 0)); --
  --
end entity;
```

The system has interfaces corresponding to the pipes inpipe and outpipe through which data is exchanged.

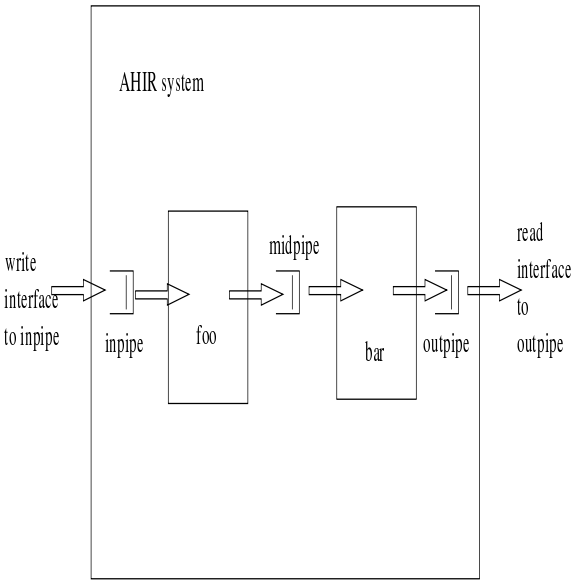

Figure 3: Hardware implementation of Foo-Bar pipeline

In practice, one can have any number of pipes and interacting processes in a pipeline implementation.

4 Storage variables and memory spaces

In an **Aa** program, variables can be of three kinds: storage variables, pipe variables, or single-static-assignment variables. Storage variables are implemented in memory, pipe variables as FIFO buffers, and single-static-assignment variables as registers.

While transforming an **Aa** description down to a **vC** description, the storage variables in the **Aa** program need to be grouped into memory spaces. Two storage variables are put in the same memory space only if we determine that a pointer de-reference in the **Aa** program can point into either of the two storage variables (this is determined by a conservative static analysis).

A memory space in a **vC** description is characterized by a word-length (the greatest common divisor of the widths of accesses to this memory space), an address-width (wide enough to allow access to all words in the memory space), and a capacity (the number of words in the memory space). Typically, a program will have many small memory spaces corresponding to scratch

storage and some large memory spaces which correspond to arrays etc.

There is one small issue, however. Consider the following C program:

```
int main(int* b)
{
   int q[2];
   q[0] = *b;
   q[1] = q[0];
   return(q[1]);
}
```

When this program is mapped to a circuit, we identify two distinct memory spaces, one which contains the array q and the other corresponding to the external world (the one referred to by the pointer b). Where is the external memory physically located? In the AhirV2 flow, we can either locate it outside the system or inside the system which is being described by this program.

If the external memory is to be placed outside, then accesses to it from within the system must be routed outside the system. On the other hand if it is to be placed inside, a storage object corresponding to it must be created and all accesses to the external memory must be directed at this storage object. Further, the external world must have a mechanism for accessing this storage object.

Both options are supported in the AhirV2 flow through **AaLinkExtMem**.

4.1 Keeping the external memory outside the system

In this scenario, all memory accesses which are resolved to be to a storage object which is not declared in the **Aa** program are redirected outside the system by using pipes.

If you want to keep the external memory outside, you will have to go through the following sequence

```
# first use clang (or llvm-gcc) to generate llvm-byte-code
clang -std=gnu89 -emit-llvm -c foo.c
#
# disassemble so that you can make sense of the llvm bc.
llvm-dis foo.o
#
```

```
# OK, now take the llvm byte code
# and generate an Aa description.
# use the storageinit option to initialize
# global storage.
# (the pipe to vcFormat is to prettify the output)
 llvm2aa -storageinit foo.o | vcFormat > foo.o.aa
#
#
# Do an Aa -> Aa transformation: map external
# memory outside..
AaLinkExtMem foo.o.aa | vcFormat > foo.o.memlinked.ExternalOutside.aa
#
# Now take the Aa code and generate a virtual
# circuit..
# the -O flag does dependency analysis in straight-line
# code and parallelizes it.
#
Aa2VC -O foo.o.memlinked.ExternalOutside.aa | vcFormat\
                > foo.o.memlinked.ExternalOutside.aa.vc
#
# finally, generate vhdl from the vc description.  Note that
# you will have to mark the module foo as well as the
# extmem_store_32/load_32 modules as top-level modules
# so that it is possible for the outside world to serve
# requests made from inside.
#
vc2vhdl -O -t foo -t extmem_store_32 -t extmem_load_32\
         -f foo.o.memlinked.ExternalOutside.aa.vc | vhdlFormat\
              > foo_o_aa_memlinked_external_outside_vc.vhdl
```

If you look at the generated top-level VHDL entity, its ports will be

```
entity ahir_system is  -- system
  port (--
    clk : in std_logic;
    reset : in std_logic;

    --   some-lines-omitted --
```

```
    --   foo-related ports --
    --   some-lines-omitted --

    extmem_read_address_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_read_address_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_read_address_32_pipe_read_ack : out std_logic_vector(0 downto 0);
    extmem_read_data_32_pipe_write_data: in std_logic_vector(31 downto 0);
    extmem_read_data_32_pipe_write_req : in std_logic_vector(0 downto 0);
    extmem_read_data_32_pipe_write_ack : out std_logic_vector(0 downto 0);
    extmem_write_address_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_write_address_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_write_address_32_pipe_read_ack : out std_logic_vector(0 downto 0);
    extmem_write_data_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_write_data_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_write_data_32_pipe_read_ack : out std_logic_vector(0 downto 0)); --
  --
end entity;
```

The external memory read and write address and data are clearly visible. The outside world is responsible for serving the read/write requests made from the inside.

4.2 Keeping the external memory inside the system

In this scenario, we will assume that all accesses to storage variables not defined in the **Aa** program are to be directed to a storage variable which is to be a viewed as a shared memory pool that is visible to the **Aa** program as well as to the outside world. The visibility to the outside world is provided by access functions that the outside world can use to read/write from this shared memory pool.

You will have to go through the following sequence:

```
# use clang (or llvm-gcc) to generate llvm-byte-code
clang -std=gnu89 -emit-llvm -c foo.c
#
# disassemble so that you can make sense of the llvm bc.
llvm-dis foo.o
#
# OK, now take the llvm byte code
```

```
# and generate an Aa description.
# use the storageinit option to initialize
# global storage.
# (the pipe to vcFormat is to prettify the output)
 llvm2aa -storageinit foo.o | vcFormat > foo.o.aa
#
#
# Do an Aa -> Aa transformation: map external
# memory to a storage area inside the system...
# -I 1024 says that the amount of memory that will be
# referred to is 1024 bytes.
# -E mempool says that the storage object corresponding
# to external memory is named mempool.
AaLinkExtMem  -I 1024 -E mempool foo.o.aa | vcFormat\
               > foo.o.memlinked.ExternalInside.aa
#
# Now take the Aa code and generate a virtual
# circuit..
# the -O flag does dependency analysis in straight-line
# code and parallelizes it.
# the -I mempool option says that external memory is
# to be mapped inside the system to object mempool..
#
Aa2VC -O -I mempool foo.o.memlinked.ExternalInside.aa\
           | vcFormat > foo.o.memlinked.ExternalInside.aa.vc
#
# finally, generate vhdl from the vc description.
# note that you will have to mark mem_load__ and mem_store__
# as top-level modules, so that the external world can
# access its memory pool inside the system.
#
vc2vhdl -O -t foo -t mem_load__ -t mem_store__ \
     -f foo.o.memlinked.ExternalInside.aa.vc\
       | vhdlFormat > foo_o_aa_memlinked_external_inside_vc.vhdl
```

In this example, we are saying that the shared memory pool variable is named mempool, and it is an array of 1024 **bytes**. The generated top-level VHDL entity has the following ports:

```
entity ahir_system is  -- system
  port (--
    foo_b : in  std_logic_vector(31 downto 0);
    foo_ret_val_x_x : out  std_logic_vector(31 downto 0);
    foo_tag_in: in std_logic_vector(0 downto 0);
    foo_tag_out: out std_logic_vector(0 downto 0);
    foo_start : in std_logic;
    foo_fin   : out std_logic;
    mem_load_x_x_address : in  std_logic_vector(31 downto 0);
    mem_load_x_x_data : out  std_logic_vector(7 downto 0);
    mem_load_x_x_tag_in: in std_logic_vector(0 downto 0);
    mem_load_x_x_tag_out: out std_logic_vector(0 downto 0);
    mem_load_x_x_start_req : in std_logic;
    mem_load_x_x_start_ack : out std_logic;
    mem_load_x_x_fin_req   : in std_logic;
    mem_load_x_x_fin_ack   : out std_logic;
    mem_store_x_x_address : in  std_logic_vector(31 downto 0);
    mem_store_x_x_data : in  std_logic_vector(7 downto 0);
    mem_store_x_x_tag_in: in std_logic_vector(0 downto 0);
    mem_store_x_x_tag_out: out std_logic_vector(0 downto 0);
    mem_store_x_x_start_req : in std_logic;
    mem_store_x_x_start_ack : out std_logic;
    mem_store_x_x_fin_req   : in std_logic;
    mem_store_x_x_fin_ack   : out std_logic;
    clk : in std_logic;
    reset : in std_logic); --
  --
end entity;
```

The system provides memory load and memory store function interfaces to the external world (through mem_load.. and mem_store..). The shared memory variable is guaranteed to have a base address of 0. Thus, byte mempool[I] will be present at address I.

5 The tools

We assume that you have access to either **llvm-gcc** or **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.

5.1 llvm2aa

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

```
llvm2aa options 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.

The options:

- **-modules=listfile** : Specify the list of functions in the bytecode which should be converted to **Aa** . The names of these functions should be listed in the text-file listfile. If absent, all functions are converted.
- **-storageinit** : Storage objects in the llvm bytecode are explicitly initialized in the generated **Aa** code. An initializer routine named

  ```
  global_storage_initializer
  ```

  is instantiated in the **Aa** code for this purpose.
- **-pipedepths=filename** : Specifies a file which contains the depths of pipes which are part of the generated **Aa** code.
- **-extract_do_while** : Innermost loops which are marked using a call to the special function

  ```
  _loop_pipelining_on_
  ```

  are extracted as pipelined do-while loops. This is necessary for automatic cross-iteration parallelization of inner loops in the generated hardware (substantial performance benefits can be realized).

5.2 AaLinkExtMem

This linker tool takes a list of **Aa** files, elaborates the program, creates a global storage initializer, and does memory space decomposition. The externally visible memory space is linked in one of two ways: either it is assumed to be external and all accesses to it are routed out of the **Aa** program, or it is assumed to be internal and assumed to consist of a memory object (an array of bytes). External pointer dereferences are handled as if they are directed at this memory object.

```
AaLinkExtMem options file1.aa file2.aa ...  > linked.aa
```

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

The options:

- **-I n**: specifies that external references to memory are to be mapped as if they are to an internal object whose size is n bytes.
- **-E obj-name** : specifies that the object to which external references are mapped is to be named obj-name.

We recommend that you use the **-I** and **-E** options to locate externally visible memory into a specified object in the **Aa** program.

If the **-I** option is not used, then all external memory references are routed out of the **Aa** program through pipes. In this case, if the **Aa** compiler determines that there is some pointer in the program which can point to both internal and external memory, then this will be declared as an error!

If the programs being linked contain memory initialization routines, the linker generates a global storage initialization function which is named

```
global_storage_initializer
```

This global initializer calls all the memory initializers in the programs being linked.

5.3 AaOpt

The optimization utility **AaOpt** takes an **Aa** program (list of **Aa** files) and produces an optimized version of the source program.

```
AaOpt options file1.aa file2.aa ... > optimized.aa
```

The optimized **Aa** code is printed to **stdout**. On success, the tool returns a 0 (else a non-zero). Macro and inlined function calls in the source code are substituted in place in the optimized code.

The options:

- **-r module-name** (optional) : specifies a root module in the system. Multiple root modules can be specified. All dead code (which is not reachable from a root module) is eliminated.
- **-I extmem-object** (optional) : similar to AaLinkExtMem, this option specifies the name of the extmem-object in the source **Aa** files.
- -B (optional) : if specified, add buffering to balance pipelined loops so that loop performance is not bottlenecked by inadequate buffering.

5.4 Aa2VC

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

```
Aa2VC options 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 options:

- **-O** : if used, sequential statement blocks are parallelized by doing dependency analysis.
- **-C** : if used, a C stub is created for every module that is not called from within the system. These stubs can be used to interface to a VHDL simulator (or even drive hardware) to verify the VHDL code generated by downstream tools.
- **-U** : memory subsystems will be unordered (that is, will not guarantee in-order completion of accesses). This leads to a simpler memory subsystem, but more conservative control flow. The default is that all memory subsystems are ordered (will complete read/write requests in the order that they are accepted).

- **-r root-module** (optional): specifies a root module. Code which is not accessible from a root-module is considered as dead code and is ignored.

- **-I obj-name** : if specified, all external memory references are considered as being directed at the storage object named obj-name. If not specified, then the tool will throw an error if it finds a pointer dereference that cannot be resolved as pointing only to storage objects declared inside the **Aa** program.

5.5 vc2vhdl

Takes a collection of **vC** descriptions and converts them to an AHIR system described in VHDL.

```
vc2vhdl [-O] [-C] [-q] [-a] [-e <entity-name] [-w]\
         -t/-T foo [-t/-T bar -t/-T bar2 ...]\
         -f file1.vc -f file2.vc ...  > system.vhdl
```

The options:

- **-t** : to specify the modules which are to be accessible from the ports of the generated VHDL system. Such modules have to be top-level (that is, they cannot be called from within the program). Multiple top-level modules can be specified in this way. The control and argument ports for these modules are visible at the interface of the generated AHIR system.

- **-T** : to specify top-level modules which are to be free-running inside the AHIR system. Multiple top-level modules can be specified in this way. Such modules do not have any arguments and do not return any values. Their only mechanism of communication with the world outside the AHIR system is through pipes. The control ports for these modules are **not visible** at the interface of the generated AHIR system. In the AHIR system, these modules are started on reset and are run forever (restarted after they finish, forever).

- **-f file-name** : specifies the **vC** files to be analyzed. Multiple **vC** files may be specified. An object must be defined before it is used, so the **vC** files must be specified in the correct order.

- **-O** : optimize the generated VHDL by compacting the control-path. This does not change the resulting hardware, but makes the generated VHDL file smaller.

- **-C** : the VHDL code has a system test bench which interfaces to foreign code using a VHPI/Modelsim-FLI interface. If this is not specified, the generated test bench simply instantiates the system and starts all top-level modules off (you will need to fill in your own test bench here). The C testbench is usually easier to write (it probably already exists in the form of the original program).

- **-a** : try to minimize the area of the resulting VHDL by sharing operators to the maximum extent possible (allowing potential contention for resources). This will result in a slower (usually by 2X) system, but will also reduce the area (usually by 0.5X). If not specified, two operations will be mapped to the same operator only if it can be proved that they cannot be active simultaneously.

- **-q** : if specified, do aggressive register insertion to minimize the clock period.

- **-S bypass-stride** : by specifying the bypass stride (an integer ≥ 1), the user can trade-off clock cycles versus clock period. The lowest clock period will be obtained for -S 1.

- **-e top-entity-name** : The generated top-level VHDL entity corresponding to the AHIR system is named top-entity-name. The default is ahir_system.

- -L function-library : AhirV2 provides some built in operator functions which can be called from your code. These are organized as function libraries and this option specifies a function library to look into when generating VHDL. For example *-L fpu* gives access to the floating point library which provides some useful built in functions (e.g. fpalu32, fpalu64 etc.).

- **-w** : If specified, the VHDL system and test-bench are generated as separate unformatted VHDL files. You will need to format these using the vhdlFormat command.

- **-s ghdl/modelsim** : If **ghdl** is specified with the -s option, then the generated testbench (if -C is specified) uses the VHPI interface to link with foreign code. Otherwise, the generated testbench (if -C is specified) uses the Modelsim FLI interface to link with foreign code.

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).

5.6 Aa2C: convert an Aa description into a C program

The AhirV2 flow offers considerable flexibility to a system designer. For example, it is possible to write code directly in **Aa** in order to get more optimal implementations (relative to those obtained starting from **C**). In such cases, if we wish to simulate the **Aa** description, we would use the **Aa2C** utility to convert the **Aa** code to ANSI **C**, and then compile it in the usual way.

The **Aa2C** program can be summarized as

```
Aa2C [-I <ext-mem-object>] <aa-file> (<aa-file>)*
```

The only option is:

- **-I ¡ext-mem-object¿**: the same behaviour as in **Aa2VC**.

The remaining arguments are **Aa** files which will be linked and converted to **C** code. Two outputs files are created:

- aa_c_model.h : a header file declaring functions in the generated source code.
- aa_c_model.c : a source file containing function definitions corresponding to the **Aa** modules.

External calls into the generated **C** code must have the form:

```
void foo ( Ctype_1 in_1, Ctype_2 in_2, Ctype_3* out_1, Ctype_4* out_2);
```

where Ctype is either a float or double or (int/uint)(64/32/16/8)_t type. You can then link your external code with the generated **C** code in the usual way.

5.6.1 Restrictions in using Aa2C

The current implementation of **Aa2C** produces un-threaded code. Thus, if you have a parallel block in your **Aa** code, the statements in the parallel block are serialized in the resulting **C** program. This can result in the generated **C** program potentially hanging (if one of the statements in the parallel block runs for-ever). Another situation is when two concurrent blocks in the **Aa** program are writing and reading from the same pipe. In such a case, the serialized code may get dead-locked. You need to be careful that your **Aa** code does not have such situations (the simplest option is to not use parallel blocks in the **Aa** code!).

This issue will be fixed in a future release of **Aa2C**.

5.7 Miscellaneous: vcFormat and vhdlFormat

The outputs produced 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 files.

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