AhirV2: Tools which take algorithms to hardware
A reference manual

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

October 3, 2023

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).

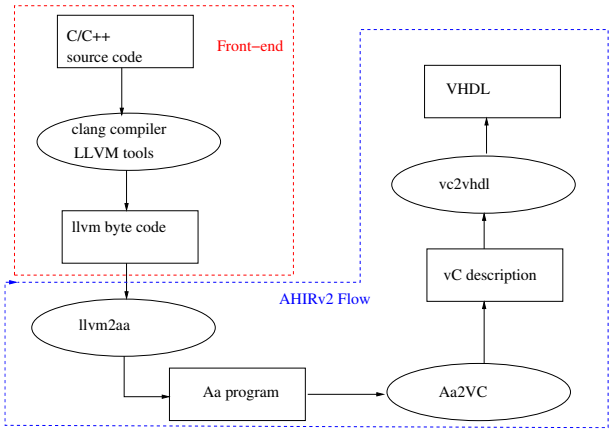

Figure 1: AhirV2 flow

- 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 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.

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 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.

2.1 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).

- There are zillions of other LLVM optimizations that are available in llvm2aa (type llvm2aa –help to see these). These involve program level optimizations such as loop unrolling, garbage collection optimizations etc. In general anything that increases instruction level parallelism will result in faster hardware, but with a cost.

2.2 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.

2.3 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.
- -C (optional) : if specified, try to eliminate registers to the maximum extent possible by marking statements which have a fanout of one as volatile.
- -B (optional) : if specified, add buffering to balance pipelined loops so that loop performance is not bottlenecked by inadequate buffering.

Note that the -B and -C options cannot be used together.

2.4 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:

- **-h**: print help message and quit.
- **-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.
- **-P** (optional): Normally, the **Aa** compiler resolves accesses to pipes and storage objects by analysing modules. But if the -P flag is specified, all modules are treated as opaque and are assumed to not have any side-effects. This can be dangerous.

2.5 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 ...]
         [-s ghdl/modelsim] [-L <file-name>]
         [-v] [-D] [-W <work-lib>]
         [-U] [-H]
         -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.

- **-U**: input/output pipes will be printed with depth 0.

- **-H**: system interface in .hsys format will be printed. This is used by another AHIR-V2 tool called hierSysBuild which can put together multiple AHIR systems using this hsys information.

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).

2.6 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>] [-P <prefix>]  <aa-file> (<aa-file>)*
```

The only option is:

- **-I ext-mem-object**: the same behaviour as in **Aa2VC**.
- **-P prefix**: Specify a prefix string (default is the empty string) which will be used to name generated files (see below).

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

prefix aa_c_model.h : a header file declaring functions in the generated source code.

prefix aa_c_model_internal.h : an internal header file with lots of macros.

prefix 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.

2.6.1 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**.

2.7 AaPreprocess: Preprocessor for Aa description

The **AaPreprocess** utility takes an input **Aa** file and applies pre-processing directives in the file to produce an output **Aa** file. The invocation of the **AaPreprocess** utility is as follows:

```
AaPreprocess [-I <search-directory>]* -o <output-aa-file> <list-of-input-aa-files>
```

The utility then applies the pre-processing directives in the input **Aa** files to produce an output **Aa** file.

The pre-process directives that are supported are

#include Specify a file to be included. This is typically specified as

```
#include foo.aa
```

The directories specified by the -I option are searched and the first matching file with name "foo.aa" is substituted in place at the point of the include.

#define Specifies a key-word definition. For example

```
#define P  const_56
```

This defines the string "P" to be an alias of the string "const_56". A #define can override another #define. The last definition is used.

#undefine This removes the specified define from the list of preprocessor defines.

```
#undefine P
```

Any further references to "P" in the source will be flagged as errors by the pre-preocessor.

This the paste directive. For example, if we have defined

```
#define P  const_56
```

and we have a line in the input file of the form

```
a := ( ##P + 1)
```

then, this will be expanded to

```
a := (const_56 + 1)
```

#if - #endif Code can be conditionally included using this construct.

```
#define TFLAG 1
#define SFLAG 0
#if TFLAG
a := b
#endif
#if SFLAG
b := a
#endif
```

will be expanded to

```
a := b
```

To summarize: the preprocessor can be used to

- include a file into the code.
- paste strings into the code.
- conditionally include code.

This adds flexibility to the programmer and can reduce the amount of repetitive code.

2.8 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.