Aria VMX Emulation Framework

v 0.4

Created By Oriol Prat
Revised and Modified by Mauricio Alvarez
Contact information: alvarez@ac.upc.edu

1. Introduction and design objectives

The main idea was to add a VMX emulation functionality to the Aria execution -simulation environment [1]. So far, Aria's emulation framework allowed to emulate instructions by codifying them in assembler language. This codification had the strong restriction that only 4 registers, called scratch, could be used. VMX instructions are usually emulated with a loop that checks simultaneously three or four virtual registers (which are implemented in memory). This forces to have three or four registers to carry out a pointing function, while three more are still needed to do the instruction operation itself. This forces to use the (memory) stack in order to temporally store the registers and reduces drastically the application performance due to the intensive usage of memory. In order to solve this limitation, we've decided to create a new Aria's emulation framework.

This framework has been designed as a dynamic library so that:

- It is able to emulate any kind of instruction, not only VMX instructions.
- It is independent from Aria's code, so there is no need to compile Aria every time that the emulation library is modified.
- It is easy to add new emulated instructions and Aria's code does not need to be modified.

2. Modifications included in Aria

These are Aria main modifications:

- *The inclusion on a new user program state for VMX registers*. This was implemented as an extension to the user program state that Aria have defined. Also we included a set of macros for accessing and displaying the contents of VMX registers.
- *The definition of an Interface between Aria and the dynamic emulation library*. This was done through a check of a simulation function table, in a similar way to the check that is done for the assembler emulated instructions.
- *The modification of Aria basic block detection and translation mechanism in order to detect VMX instructions* (or other emulated instructions). Each VMX instruction is emulated separately, in a similar way to a basic block with only one instruction.

Next we describe each modification made in Aria and its correspondent file.

All modifications included in Aria are preceded by the comment:
/* Oriol Prat modified */

Next, all files and modifications are listed.

43translate/include

globdefs.h

The following code was added at line 92:
```
#include "aria_emulib.h"
#include "proc_state.h"
```

The following code was added at line 266:
```
struct emutable_struct _sim_function_table[OPC_NUM];
```

The following code was added at line 372:
```
#define sim_function_table (the_global_flags->_sim_function_table)
```

proc_state.h

The following code was added at line 132:
```
vmx_emu_vector_t  _aria_VMXS[32];     /* VMX registers */
vmx_emu_vector_t _aria_VSCR;         /* VMX state */
BASE_TYPE   _aria_VRSAVE;       /* VMX Vector save/restore register */
```

The following code was added at line 189:
```
#define proc_state_CR_PTR(x)            (&(x)->_aria_condition_reg)
#define proc_state_VMXs(x)              (&(x)->_aria_VMXS)
#define proc_state_VMX(x,y)             (&(x)->_aria_VMXS[y])
#define proc_state_VSCR(x)              (&(x)->_aria_VSCR)
#define proc_state_VRSAVE(x)            (&(x)->_aria_VRSAVE)
#define  set_proc_state_saturate(x)           (((x)->_aria_VSCR[3])  ||
128) /* 10000000 */
#define  clr_proc_state_saturate(x)           (((x)->_aria_VSCR[3])  &&
127) /* 01111111 */
#define  set_proc_state_nonjava(x)            (((x)->_aria_VSCR[1])  ||
128) /* 10000000 */
#define  clr_proc_state_nonjava(x)            (((x)->_aria_VSCR[1])  &&
127) /* 01111111 */
```

The following code was added at line 209:
```
void warn_vmx_processor_state(proc_state_type * state_ptr);
```

43translate

Makefile.Aix4_3

The following code was added at line 48:
```
EMULIB_SRCDIR = $(AriaROOT)/aria_emulib
EMULIB_INCDIR = $(AriaROOT)/aria_emulib/include

INCDIR_FLAG = -I$(INCPATH) -I$(OPC_INCDIR) -I$(EMULIB_INCDIR)
LIBDIR_FLAG = -L$(LIBDIR) -lm -laria_emulib
```

The following code was added at line 82:

```
do_function_simulation_translation.c \
warn_vmx_proc_state.c \
```

aria_interface.c

The following code was added at line 358:

```
/** this is a function simulation basic block (1 instruction), so no
translation is required **/

if (global_do_aria_simulation &&
sim_function_table[opc_index(*next_bb_address)].active
&&
sim_function_table[opc_index(*next_bb_address)].exec_func != NULL
&&
(
(sim_function_table[opc_index(*next_bb_address)].check_func != NULL &&
sim_function_table[opc_index(*next_bb_address)].check_func
(*next_bb_address, opc_index(*next_bb_address)))
||
(sim_function_table[opc_index(*next_bb_address)].check_func == NULL)
)
) {

IFDEBUG(fprintf(mystderr,    "[aria_interface]    Simulation    Function
detected.\n");fflush(mystderr);)

next_xlate_address =
do_one_function_simulation_translation (next_bb_address,
    the_hash_entry_ptr, speculative_not_taken_flag,
    INTERACT_XLATE_MODE);
```

The following code was added at line 435:

```
/*  execute  the  function  associated  with  a  function  emulated
instruction */

if (global_do_aria_simulation &&
sim_function_table[opc_index(*next_bb_address)].active
&&
sim_function_table[opc_index(*next_bb_address)].exec_func != NULL
&&
(
(sim_function_table[opc_index(*next_bb_address)].check_func != NULL
&&
sim_function_table[opc_index(*next_bb_address)].check_func
(*next_bb_address, opc_index(*next_bb_address)))
||
(sim_function_table[opc_index(*next_bb_address)].check_func == NULL)
 )
 ) {
  IFDEBUG(char dis_buf[256];opc_dis(*next_bb_address, dis_buf);
    fprintf(mystderr, "Executing emulated vmx function %s\n",
    dis_buf);fflush(mystderr);)
 /* execution = emulation function call */

sim_function_table[opc_index(*next_bb_address)].exec_func
(*next_bb_address, opc_index(*next_bb_address), opc_ifformat(opc_index
(*next_bb_address)), saved_proc_state_ptr, *microtrace_buffer_ptr);
}
```

aria_main.c

The following code was added at line 84:

```
#include <memory.h>
```

The following code was added at line 117:

```
#include "aria_emulib.h"
```

The following code was added at line 455:

```
/* Initialize function simulation table */
/* aria_emulib_init is defined in the aria_emulib dynamic library */

warn("Initializing function simulation table from libaria_emulib...");

memset ((void *)sim_function_table, 0, sizeof(struct emutable_struct)
*OPC_NUM);

if (!aria_emulib_init (sim_function_table)) {
  printf ("[aria_main] Error initializing sim_function_table.\n");
}
```

do_translation.c

The following code was added at line 371:

```
/* Oriol Prat modified */
/* Determine if the instruction is simulated by function in this run,
and thus ends the BB */

if (global_do_aria_simulation &&
sim_function_table[opc_index(iword)].active &&
sim_function_table[opc_index(iword)].exec_func != NULL &&
 ((sim_function_table[opc_index(iword)].check_func != NULL &&
sim_function_table[opc_index(iword)].check_func(iword,        opc_index
(iword)))
||
 (sim_function_table[opc_index(iword)].check_func == NULL)
)
) {
    char buf[256];
    opc_dis(iword, buf);
    warn_xlate("Ending block due to function simulation instruction...
               %s\n", buf);

    goto block_ends_before_a_function_simulated_instruction;
  }
```

warn_vmx_proc_state.c

This new file is a copy of warn_proc_state, where the following lines have been added at the end of the function.
Currently it isn't used in Aria.

```
#include "libvmx.h"
```

```
  i=0;
  while (i<32) {
    printf ("        VMX reg (%2d) = ", i);
    vec_print_hex ((unsigned char *)proc_state_VMX (state_ptr, i));
    i++;
  }
  warn_proc_state("\n");
```

3. The dynamic library for instruction emulation using functions

All functionality focuses on aria_emulib.c and aria_emulib.h files. The instruction emulation table is defined in aria_emulib.c. This table are initialized in compiling time with all information needed for each emulated instruction. These files are placed in aria_emulib directory at Aria's root directory.

```
struct emutable_struct emutable_table[] = {

#include "aria_vmx_emu.table"

/* for add new emulation functions copy aria_table_skeleton.table */

  /* this entry is mandatory, don't remove it */
  /* it must be the last entry in this table */
  { 0, 0, NULL, NULL }
};
```

In aria_emulib.c is implemented the initialization function that copy the partial emulation table to the general emulation table passed by parameter.

```
int aria_emulib_init (struct emutable_struct et[])
{
  unsigned int i=0;

  while (emutable_table[i].opc_index != 0) {
    if (et[emutable_table[i].opc_index].exec_func != NULL) {
      printf ("[aria][emutable] repeated entry number %u in library
         emutable (%u opc_index).\n", i, emutable_table[i].opc_index);
    } else {
      et[emutable_table[i].opc_index].exec_func =
                                        emutable_table[i].exec_func;
      et[emutable_table[i].opc_index].active =
                                        emutable_table[i].active;
      et[emutable_table[i].opc_index].check_func =
                                        emutable_table[i].check_func;
    }
    i++;
  }
  return (1);
}
```

3.1 How to add the emulation of a new instruction

- **Define the new instruction:** First it is necessary to define the new instruction in the opcode database used by Aria. This file is located at $ARIA/opcode/opcode.table. For creating new instructions it is necessary to find an empty opcode in the PowerPC ISA defined in the PowerPC architecture book [2].
  As an example of an instruction definition, below there is the definition of a new instruction called VINTERUB (vector interpolation of unsigned bytes) :

```
VINTERUB
format        VX_form
full_format   VX_1
ppc_name      vinterub
ppc32_name    vinterub
opcode        4
extended_op   1901
dlength       1
VMX
@end
```

- **Register the emulation of the new instruction:** Then it is necessary to register the emulation of the new instruction in the file $ARIA/aria_emulib/aria_vmx_emu.table

Edit the file and add the necessary lines, each one for each one of the new instructions

```
{ OPC_VINTERUB, ACTIVE, NULL, emulation_func },
```

The first field corresponds to the instruction index within the Aria opcode database (opc.h). In the previous step, and after the compilation of Aria, the declaration of the new instruction is generated. OPC_VINTERUB corresponds to the index of the *vinterub* instruction.

The second field must be filled with ACTIVE or INACTIVE, which disables the emulation of the instruction temporally, without having to remove it from the table.

The third field contains the pointer to a function that checks if the instruction must be emulated or not. This function is needed only when the decision of emulating an instruction depends on one of its operands. For instance, in the case of *mfspr*, where the SPR operand identifies to which special purpose register it reads from, when SPR equals 256 the instruction is referring to the VMX extension.

The forth field is the emulation function. This function decodes the instruction's operand, calculates the register's addresses in the process state, emulates its operation and fills the microtrace buffer, in case it is necessary, with the referenced address. For the VMX extension the emulation function is called: *vmx_emulate*

- **Implementation of the emulation function**

The emulation function carries out the following tasks:

- Decodes the instruction.

- Looks for the pointers to the operand registers within the process state structure.
- Emulates the functionality of the instruction.

For the VMX instructions there is an implementation of the emulation function called *vmx_emulate* in the file $ARIA/aria_emulib/aria_vmx_emu.c.

It is necessary to add the new instruction to the decoder of the emulation function by adding a *case* for the new instruction with a call with the specific emulation function itself, as it is shown in the next example:

```
case OPC_VINTERUB
        vmx_emu_vinterub(vt, ra, rb);
        break;
```

- **Implementation of the specific emulation function**
  Finally it is necessary to implement the code that emulates the functionality of the new instruction. For VMX instructions this can be done at the file $ARIA/aria_emulib/libvmx.c

3.2 VMX emulation using the new emulation framework

The VMX instruction set extension was implemented using the new Aria's emulation framework. All files are placed in *aria_emulib* directory at Aria`s root directory. VMX instruction emulation table was implemented in *aria_vmx_emutable* file than was included by *aria_emulib.c* file. Check and decode functions are placed in aria_vmx_emu.c file. Emulation functions are implemented in *libvmx.c*

NOTE:
In order to use the emulation library it must be defined the environment variable EMULIB_PATH with the exact path to the emulation library file *libaria_emulib.a.*

The emulation library implemented in libvmx.c currently supports the integer subset of VMX extension. Some floating point instructions are not implemented yet. The implemented instructions have been tested using two different testbenchs. The first one is a direct test of the functionality of each functions with some test inputs. The second test was done using TestVMX (developed at CIRI UPC-Barcelona) that is an extensive test of the correctness of VMX instructions implemented in C language. This test was done using the emulation library with Aria in a Power4 machine with AIX 5.2

3.3 Bugs found in Aria during the development of the emulation library

At *opcode/opcode.table* file:
Instructions vcfsx, vcfux, vctsxs and vctuxs contain the attribute VX_1 but it should be VX_2. This causes the emulation function to decode those instructions erroneously.

At *opcode/opcode.pl*:
In lines 2453 and 3029, the word FULFORM must be substituted by FULLFORM, or the full formats archive is wrongly built. The full format of all instructions are displaced one position back as the OPC_ILLEGAL_FORMAT first line is missing.

4. Generating VMX code for Power based systems with AIX OS

In order to generate VMX code for AIX systems we have built a custom gcc compiler suite.

- We have use gcc-3.3.3 (also we have tested gcc-4.0.2). VMX support is added by modifying $GCC_SOURCE_DIR/gcc/config/rs6000/aix.h file where the definitions of ALTIVEC are placed:
  #define TARGET_ALTIVEC 1
  #define TARGET_ALTIVEC_ABI 1
  #define TARGET_ALTIVEC_VRSAVE 1

- There are some issues related to the requirement to have 128b alignment for all VMX references. GCC attempts to maintain 128b alignment using .align directives but fails in the use of .csect. For example in the next piece of code, generated by gcc, it can be shown that the *.align 4* directive is useless if the *csect* directive are not aligned to 128b (in the final part of the directive ,3).

```
.csect _VECintIDCT.rw_c[RO],3
      .align 4
LC..0:
      .short      2
      .short      2
      .short      2
      .short      2
      .short      2
      .short      2
      .short      2
      .short      2
```

In order to resolve that problem it is necessary to hard code gcc in such a way that always produce a .csect with 128b alignment, even for non VMX accesses. This can be implemented in the file *$GCC_SOURCE_DIR/gcc/config/rs6000/xcoff.h* and define 16 bytes (2^4) as the biggest alignment for the global variables.

```
#define READ_ONLY_DATA_SECTION_FUNCTION                 \
void                                                    \
read_only_data_section (void)                           \
{                                                       \
  if (in_section != read_only_data)                     \
    {                                                   \
      fprintf (asm_out_file, "\t.csect %s[RO],4\n",     \
               xcoff_read_only_section_name);           \
      in_section = read_only_data;                      \
    }
      \
}
```

```
#define PRIVATE_DATA_SECTION_FUNCTION                \
void                                                 \
private_data_section (void)                          \
{                                                    \
  if (in_section != private_data)                    \
    {                                                \
      fprintf (asm_out_file, "\t.csect %s[RW],4\n",  \
             xcoff_private_data_section_name);       \
      in_section = private_data;                     \
    }                                                \
}

#define READ_ONLY_PRIVATE_DATA_SECTION_FUNCTION\
void                                                       \
read_only_private_data_section (void)                      \
{                                                          \
  if (in_section != read_only_private_data)                \
    {                                                      \
      fprintf (asm_out_file, "\t.csect %s[RO],4\n",        \
             xcoff_private_data_section_name);             \
      in_section = read_only_private_data;                 \
    }                                                      \
                                                           \
}                                                          \
```

- For generating the object files with VMX instructions it is necessary to use the GNU binutils assembler *GAS* and pass to it (at assembling time) the -maltivec flag. We have tested our codes with binutils-2.6.1.

- When the object files are generated it is necessary to make the link process with the xlc linker for avoiding problems with the Aria dynamic linking.

5. Example of VMX emulation using the new framework

With the new code of Aria we have provided some examples of code using the VMX emulation library.

- ***vand_example*:** The first one is a simple code with one VMX instruction and can be used a test for the compiler and the Aria emulation environment. Simple go to the vand example directory, change the Makefile, and type:
  $ make
  $ make aria
  The latest command calls Aria with the *simple_driver* executes the program and at the end print some statistics.

- ***testVMX*:** testVMX is a test program designed to check the correctness of VMX instructions generated by a compiler that support them. It was developed by Jose Maria Cela, Raul de la Cruz,Rogeli Grima, Xavier Saez at the CEPBA-IBM Research Institute. It can be downloaded at http://www.ciri.upc.es/cela_pblade/.
  We have included some minor modifications in order to support the VMX emulation based on Aria using gcc on Power4/AIX systems. If you want to

run it under Power/AIX systems you ned to have a VMX enabled gcc compiler and gas assembler. Go to the directory *testVMX-0.2.2*

1) modify the Makefile and put the path to your path to the gcc compiler and gas assembler

2) put in the makefile the path to aria

3) compile the code: $ make testVMX-gcc-aix

4) run Aria: $ make aria

testVMX checks all the VMX instructions and prints a message that indicates of the results were wrong or not

- ***IDCT***: Finally we have selected the Inverse Discrete Cosine Transform, that is a common kernel in many image and video applications. The code was taken from Apple web site
http://developer.apple.com/hardware/ve/examples.html

- First we compile and link the code with gcc and xlc respectively

```
VECintIDCT.s: VECintIDCT.c
      gcc -maltivec -mabi=altivec -c VECintIDCT.c
      gcc -maltivec -mabi=altivec -c VECmainintIDCT.c
      xlc VECintIDCT.o VECmainintIDCT.o -o vmx_idct
```

- Second we can call Aria using an available driver and the generated vmx_idct binary

```
$(Aria_PATH)/example_run_scripts/runAria vmx_idct
```

- Finally we obtain the correct results of the vmx_idct

```
output data:
    37     13    -16    -23     24     25     28     42
   -17     28     22     14    -20     62     49     -7
    34    -26     -4     -5     78     62     46    -52
     8     21    -20     48     27     47    -16     16
    60     28     -6    -82     47     71     57    -44
    43     43     -2      9    -42     35     28     16
    77     -7     -9      1     45     16     21    -13
    -8     45     35     16     23     46      4    -31

Done with program execution...
I PROMISE IT IS...
--------------------------------------------------------------------
The final statistics (gathered by Aria) are:
User program vmx_idct_asm.aria invoked as:
     vmx_idct.aria

Translation resulted in 403 block translations, and
                        9828 basic block executions

There are a total of 1268 original (static) instructions,
      and a total of 16902 translated (static) instructions.

There are a total of 46638 original instructions executed,
      and a total of 565013 translated instructions executed.

The user program returned an exit code of (0)
Of the 403 translated basic blocks,
       0 were broken due to maximum code size limit,
       0 were broken due to register addresibility,
```

```
and 0 were broken due to user stop conditions.
```

It is necessary to note that the statistics shown above do not include the VMX emulated instructions.

5. To do

There are two important things necessary to do with the current version:

- To optimize the VMX emulation by eliminating the generation of a translated basic block for each VMX instruction that performs no useful operations.
- To make an extensive test of Aria with the emulation library using real applications. We are starting to make more test using MPEG video codecs.

References

[1]. ARIA Excecution driven trace generator.
http://www.research.ibm.com/MET/Toolset/Aria/aria.html

[2]. Book I: PowerPC User Instruction Set Architecture.
http://www-128.ibm.com/developerworks/eserver/articles/archguide.html