

AMRM Prototype Board Software API *

Dan Nicolaescu Prashant Arora Ana-Maria Badulescu
Alexander Veidenbaum

Department of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697-3425
{dann,arora,ancuta,alexv}@ics.uci.edu

Technical Report #00-38

Dept. of Information and Computer Science
Univ. of California at Irvine

June 2000

*This work was supported in part by the DARPA ITO under Grant DABT63-98-C-0045.

Contents

1	Introduction	2
1.1	Goals	2
2	Initialization and Clean-up	2
3	Reading and Writing to AMRM Board Memory	3
3.1	Cached access	3
3.2	Uncached access	3
3.2.1	Uncached SRAM	4
3.2.2	Uncached DRAM	4
4	Configuring the AMRM board	4
4.1	Setting the Timing Parameters	5
4.2	Reading the Timing and the Counters from the AMRM Board	5
4.3	Resetting the Counters	5
4.4	Reading the Status Registers	6
5	The AMRM GUI program	6
6	Trace-AMRM, a Trace-Driven Simulation Program	6
7	Using SimpleScalar simulator with AMRM board	7
7.1	Changing the line size dynamically	8

1 Introduction

The AMRM board is a platform for experimenting with reconfigurable memory hierarchies. The current implementation supports 256MB DRAM and 2MB SRAM that can act as Level1 cache.

The size of the cache, the line size, the write policy and associativity of the cache can be configured at run time.

The board contains counters for the number of accesses (reads and writes), number of cache hits, and a virtual time interface for accurate timing.

1.1 Goals

The central goal of the driver software is to present the programmer a software interface that closely follows the features of the AMRM board, but does not expose too much the internals, or how the board functions.

The software API allows the user to access all the features of the board from normal user-space C/C++ programs, no knowledge about the PCI interface, or the kernel internals is needed.

The driver is cross-platform, it runs both in Windows-NT and Linux, by doing this we can exploit the strength of both platforms: initial ease of developing a graphical application used for debugging the board and the ability to use the power standard UNIX tools afterwards.

This document presents the software interface for working with the board, and documents the capabilities of the following:

- AMRM-gui program, a GUI for accessing and testing the board
- Trace-AMRM, a trace driven simulation program that can be used to exercise the board
- Interfacing the AMRM board with the SimpleScalar simulator

2 Initialization and Clean-up

The board accesses use a 32bit unsigned integer data type, thus the following typedef. If on your platform "int" is not 32bit wide change the following to a data type that is 32bit.
typedef unsigned int UINT;

int connectToBoard ();

This function should be called in a program before any other access to the AMRM board, it initializes the driver, to board and the data structures for communication with the board.

void setCacheOnOff (int value);

Is used to enable/disable the cache. The cache is initially disabled, so this function needs to be called in order to enable it.

void clearCache ();

Clear the cache. Invalidate all the entries.

void disconnectFromBoard ();

This function should be called at the end of the program, it frees all the resources used in the communication with the board.

3 Reading and Writing to AMRM Board Memory

The AMRM board has both SRAM and DRAM. They can be accessed as a cache memory hierarchy, or separately.

All the *read* and *write* functions take a *delta_t* parameter that is the time that has passed from the previous board access instruction. This time is added to the internal virtual time counter and it can be used to accurately keep track of time.

32bit and 64bit values can be read and written to the board.

All the *read* functions block until the result is returned.

3.1 Cached access

These function access do a cached access to the board.

UINT read32Cached (UINT address, unsigned short int delta_t);

Returns a 32bit value from the address *address* from the board.

void write32Cached (UINT address, UINT value, unsigned short int delta_t);

Writes the 32bit *value* to the address *address* on the board.

Some compilers for 32bit processors don't have a 64bit data type, that is why the 64bit accesses are done using 2 32bit words.

void read64Cached (UINT address, UINT addrFirstWord, UINT* addrSecondWord, unsigned short int delta_t);*

Read a 64bit value from address *address* on the board and put the first 32bit word to address *addrFirstWord* and the second 32bit word to address *addrSecondWord*.

void write64Cached (UINT address, UINT firstWord, UINT secondWord, unsigned short int delta_t);

Write a 64bit value made up from a 32bit first word *firstWord* and a 32bit second word *secondWord* to the address *address*.

3.2 Uncached access

It is possible to do uncached accesses to the AMRM board, both the DRAM and the SRAM can be accessed separately.

3.2.1 Uncached SRAM

The functions used for uncached SRAM access are presented below. The parameters and behavior is identical to the cached access functions.

```
UINT read32UncachedSram (UINT address, unsigned short int delta_t);
void read64UncachedSram (UINT address, UINT* addrFirstWord, UINT* addrSecondWord, unsigned short int delta_t);
void write32UncachedSram (UINT address, UINT value, unsigned short int delta_t);
void write64UncachedSram (UINT address, UINT first, UINT second, unsigned short int delta_t);
```

3.2.2 Uncached DRAM

Similarly the functions for doing uncached DRAM access are:

```
UINT read32UncachedDram (UINT address, unsigned short int delta_t);
void read64UncachedDram (UINT address, UINT* addrFirstWord, UINT* addrSecondWord, unsigned short int delta_t);
void write32UncachedDram (UINT address, UINT value, unsigned short int delta_t);
void write64UncachedDram (UINT address, UINT first, UINT second, unsigned short int delta_t);
```

4 Configuring the AMRM board

Various parameters of the AMRM board can be changed using the functions below.

```
void setL1Size (int lengthInKBytes);
```

Sets the size of the cache to be *lengthInKBytes*. The possible values are: 8, 16, 32, 64, 128, 256 and 512.

```
void setLineSize (int lengthInBytes);
```

Sets the cache line size to be *lengthInBytes*. The possible values are: 8, 16, 32, 64, 128, 256 and 512.

```
void changeWrPolicy (int wrpolicy);
```

With a true argument set the write policy to be write-back and with a false argument to be write-through.

```
void setAdaptivity (int value);
```

With a true argument make the cache be adaptive. NOTE: this is currently not implemented.

```
void setSramSize (int value);
```

Set the size of the SRAM to be *value*. The size should be initialized correctly on boot-up, so there's no real need to call this function.

```
void setDramSize (int value);
```

Similar to the previous function, but for DRAM.

```
void setAssociativity (int value);
```

Set the cache associativity to be *value*. Possible values are: 1, 2 and 4. For now only a direct mapped cache is implemented.

4.1 Setting the Timing Parameters

The timing parameters for the cache can be set using the functions below. These values will be added to the virtual time register in each specific case.

void setCacheHitTime (int value);

Set the cache hit time to be *value*.

void setMissFetchTime (int value);

Set the miss fetch time to be *value*.

void setWriteThruMissTime (int value);

Set the write-through miss time to be *value*.

void setWriteBackMissFetchTime (int value);

Set the write-back miss time to be *value*.

4.2 Reading the Timing and the Counters from the AMRM Board

The internal board counters can be read using the following functions.

UINT getVirtualTime (void);

Read the virtual time from the board.

UINT getNumReads (void);

Read the number of reads.

UINT getNumWrites (void);

Read the number of writes.

UINT getNumReadHits (void);

Read the number of read hits.

UINT getNumWriteHits (void);

Read the number of write hits.

UINT getNumWriteBacks (void);

Read the number of write backs. This should be zero when the cache is set to write-through.

4.3 Resetting the Counters

The counters on the AMRM board can be reset using the following functions.

void resetVirtualTime (void);

Reset the virtual time counter.

void resetNumReads (void);

Reset the number of reads counter.

void resetNumWrites (void);

Reset the number of writes counter.

void resetNumReadHits (void);

Reset the number of read hits counter.

void resetNumWriteHits (void);

Reset the number of write hits counter.

void resetNumWriteBacks (void);

Reset the number of write backs counter.

void resetCounters ();

Is a convenience function that resets all the counters.

4.4 Reading the Status Registers

The status registers can be read using the functions below.

UINT getStatusInterrupt (void);

Read the status of the interrupt register.

UINT getStatusCounterOverflow (void);

Read the status of the counter overflow register. If it the value is true one of the counters has overflowed.

5 The AMRM GUI program

The AMRM GUI program was designed for the testing the AMRM board and is an easy interface to all the boards features. It runs under Windows NT.

It has dialog boxes that allow to:

- change the cache size
- change the cache line size
- change the cache associativity
- read the performance counters
- reset the performance counters
- run programs instrumented to use the AMRM board, and then display the performance counters
- read and write data from DRAM
- access a cache location, a cache line or the tag corresponding to a cache location

6 Trace-AMRM, a Trace-Driven Simulation Program

The AMRM board can be used to do trace-driven simulation using the *trace-AMRM* program. *Trace-AMRM* reads a memory trace in a format presented bellow executes the actions in the trace and when at the end it prints the read/write hit/miss counters and the virtual time.

The trace contains lines in the following format:

- 32 and 64 bit read commands
R 32—64 ADDRESS DELTA_T [VALUE]
32 or 64 bit access
ADDRESS - a 32bit value representing the address of the access
DELTA_T - a 16bit value representing the time from the previous instruction
- 32bit write command
W 32 ADDRESS DELTA_T VALUE
VALUE - a 32 value to be written for a write instruction, this field is not present for a read instruction
- 64bit write command
W 64 ADDRESS DELTA_T VALUE1 VALUE2
VALUE1 and VALUE2 are 32bit values, VALUE1 is the MSW of the 64bit value to be written and VALUE2 is the LSW.
- set line size
L DECIMAL_VALUE
VALUE is in bytes can be: 8, 16, 32, 64, 128, 256, 512
- set cache size
C DECIMAL_VALUE
VALUE is in KBytes and can be: 8, 16, 32, 64, 128, 256, 512
- set associativity
A DECIMAL_VALUE
VALUE can be 1, 2 and 4 for a direct-mapped, 2-way or 4-way set-associative cache.
Only the direct-mapped cache is currently implemented.

VALUE, VALUE1, VALUE2, ADDRESS and DELTA_T are hexadecimal.

Usage:

trace-AMRM TRACE_FILE_NAME

7 Using SimpleScalar simulator with AMRM board

We ran several test programs and benchmarks through SimpleScalar simulator, using both the AMRM board and SimpleScalar's cache simulator. The miss rates and number of memory accesses collected by SimpleScalar and the board were identical. The virtual time from the board and the time reported by SimpleScalar are comparable. They may slightly differ if the last instruction in the program is not a memory access, since the board virtual time doesn't get updated.

In order to use the AMRM board, we firstly call the initialization functions, and then the functions for setting up the parameters of the cache. They should be the same for the AMRM board and for SimpleScalar's cache simulator. To use any of the functions, just include the following in the source code:

```
#include "amrmAPI.h"  
#include "amrmAPI-aux.h"
```

Secondly, for each access to the SimpleScalar's cache simulator we inserted calls to the functions for accessing the board (read or write), with the appropriate value for `delta_t`.

The programs to be simulated through SimpleScalar do not need to be recompiled for using the board. Only some source files of the simulator were modified and needed to be recompiled:

- the source-file with the simulator itself. We had to insert the calls to the board functions for both initialization and read/write accesses
- the source-file of the cache simulator. SimpleScalar's cache simulator models only non-blocking write-back cache. We wanted write-through, blocking cache.

7.1 Changing the line size dynamically

To dynamically change the line size from SimpleScalar, we inserted annotated instructions in the source code of the sample programs to be ran, and then extended SimpleScalar to recognize these annotated instructions and make the appropriate calls to the board functions.

For instance, for changing the line size to a value `XX`, we used

```
#define CHANGE(XX) asm("addi/a 2,3," #XX)
```

Whenever SimpleScalar fetches the annotated instruction `addi/a`, it will call the function for changing the line size on the AMRM board. Note: when changing the line size on the board, you also have to flush the cache on both the AMRM board and SimpleScalar.