XMSIM: A TOOL FOR EARLY MEMORY HIERARCHY EVALUATION

Grigoris Dimitroulakos, Theodoros Lioris, Christakis Lezos, Konstantinos Masselos

University of Peloponnese, Department of Computer Science and Technology
Terma Karaiskaki, 22100 Tripoli, Greece
{*dhmhgre, tlioris, lezos, kmas*}@*uop.gr*

ABSTRACT

In this demonstration we present the usage of XMSIM, a tool for memory hierarchy evaluation of multimedia applications. The input is a high level C code application description and a memory hierarchy specification and the output are the statistics characterizing the memory operation.

Index Terms— Memory simulation, memory hierarchy evaluation tools, computer aided design, code transformations

1. INTRODUCTION

Memory architecture design is one of the major tasks in a formalized embedded system design flow. Memory operation has a large impact on the overall efficiency of the system in terms of performance and power consumption. The use of cache memories has long provided a way to hide such latencies and reduce power consumption [1]. For the above reasons, it is highly acceptable today that memory hierarchy design is one of the major issues in modern embedded systems design.

In this paper we present XMSIM, a memory hierarchy evaluation framework that unifies the algorithmic development and memory hierarchy optimization phase. XMSIM tool is part of a tool chain developed in the context of the FP7 ENOSYS project [2] with the objective of designing efficient memory architecture for the application's input specification.

The tool's main features are the capability to: 1) simulate any subset of the application's data types, 2) support user defined mapping of data to memories , 3) simultaneously simulate multiple memory hierarchy scenarios, 4) immediate feedback to code transformations effect on memory hierarchy behavior and 5) verification utilities for the validation of code transformations. Once the tested algorithm is correctly integrated into the XMSIM environment, one can use the external GUI to easily model many memory hierarchies in parallel and collect the results in a file, instead of modifying the external memory description file and run the tool from the command prompt.

*This work has been partially supported by the FP7 ENOSYS (FP7 ICT-2009.3.4 248821) project, funded by the European Community.

This paper is organized as follows: section 2 illustrates the way a C file is integrated and the memory hierarchy description grammar, section 3 describes the demonstration example and section 4 summarizes the work and future development.

2. SYSTEM DESCRIPTION

XMSIM is a library of memory simulation routines that derives its stimuli from array accesses happening in the application code. The code of the tested application along with the XMSIM code compile as a single compilation project and the derived executable supplies the simulation results (for the given scenarios of memory hierarchies) after execution. Multiple memory scenarios can be simulated in one step while for defining each memory architecture scenario a graphical user interface is developed in Tcl/Tk. More details on the tool can be obtained from [3].

The memory description grammar is of the form *attribute: value*. Every virtual memory unit is named and its physical or functional characteristics can be described. Memory units can be virtually connected in order to operate hierarchically. There is an explicit command that dictates direct connection of a memory unit (DRAM or cache) to at most one CPU. Energy dissipation of the SRAM caches, dictated using the 'Cost per access' directive, is calculated using results retrieved from CACTI [4].

An example of memory hierarchy description is given in figure 1. Two memory architectures are defined in a single file. The first is comprised of a single DDR RAM memory unit which is made of four banks of 1024 columns, 2048 rows each. Every memory word consists of 32 bits. This memory is standalone and therefore directly connected to the CPU. In the second architecture, the memory hierarchy is made of three memory units. The main memory is a single RAM of 1024 words of 8 bits, assisted by the L2 cache, which operates under N-way policy, where N is 3. It acquires a write-through miss policy and is made of 24 words of 8 bits each. The L2 cache memory unit is aided by the L1 cache memory, which functionally operates under a write-back miss policy is direct-mapped organized and physically bears 8 words of 8 bits.

In order to monitor memory accesses, the native-C type arrays of the algorithm are converted into XMSIM's objects in C++. Figure 2 depicts an example of the engagement of the algorithm's arrays to the XMSIM environment. The CArrayWrapper objects must be mapped to every tested memory architecture. When these object arrays are referenced in the algorithm, their accesses to every memory unit is recorded.

```
#Architecture 1
Name: DDR RAM;
Type: DDR;
Banks: 4;
Rows: 2048;
Columns: 1024;
Wordsize: 32;
CPU connected: yes;
end: DDR2;

#Architecture 2
Name: Single Ram;
Type: RAM;
Words: 1024;
Wordsize: 8;
end: Single Ram;

Name: L2;
Type: Cache;
Helps memory: Single Ram;
Memory policy: N-way;
N is: 3;
Words: 24;
Wordsize: 8;
Miss policy: write-through;
end: L2;

Name: L1;
Type: Cache;
Helps memory: L2;
Memory policy: direct;
N is: 1;
Words: 8;
Wordsize: 8;
Miss policy: write-back;
CPU connected: Yes;
end: L1;
```

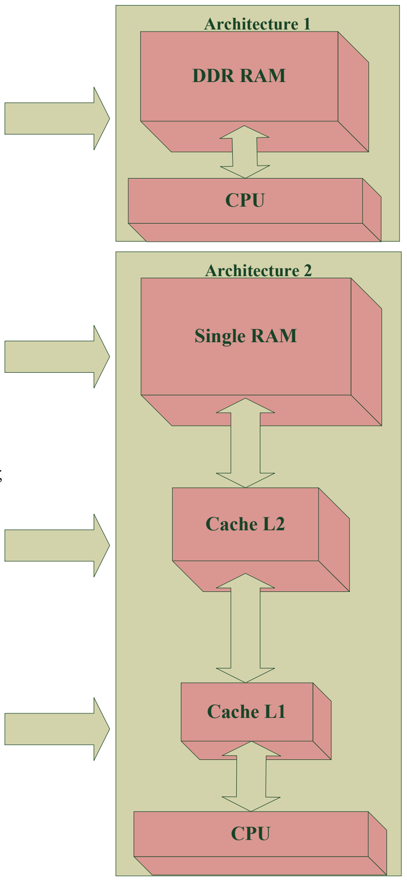
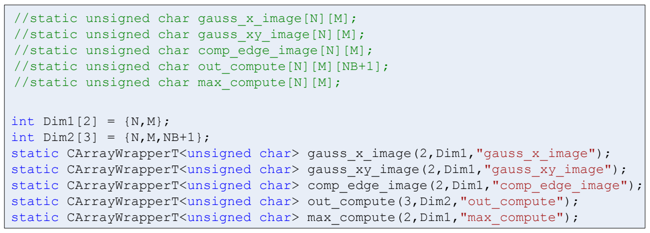


Fig. 1: Memory hierarchy description grammar.



```
//static unsigned char gauss_x_image[N][M];
//static unsigned char gauss_xy_image[N][M];
//static unsigned char comp_edge_image[N][M];
//static unsigned char out_compute[N][M][NB+1];
//static unsigned char max_compute[N][M];


int Dim1[2] = {N,M};
int Dim2[3] = {N,M,NB+1};
static CArrayWrapperT<unsigned char> gauss_x_image(2,Dim1,"gauss_x_image");
static CArrayWrapperT<unsigned char> gauss_xy_image(2,Dim1,"gauss_xy_image");
static CArrayWrapperT<unsigned char> comp_edge_image(2,Dim1,"comp_edge_image");
static CArrayWrapperT<unsigned char> out_compute(3,Dim2,"out_compute");
static CArrayWrapperT<unsigned char> max_compute(2,Dim1,"max_compute");
```

Fig. 2: Native-C type arrays conversion example.

The tool is equipped with validation tools to ensure that the conversion process of native-C data types to C++ objects is correct. When the engagement of the algorithm to the XMSIM environment is complete, the execution of the tool produces a new output file about memory usage. Write / read accesses are reported along with hits and misses for every memory unit. The results can be imported into a spreadsheet for statistical evaluation. Furthermore the tool provides a mapping of the memory contents at one or more desired execution points.

3. DEMONSTRATION

The file bearing the algorithm must be integrated into the XMSIM project as described in detail in [3]. As depicted in figure 2 the native-C type arrays must be converted to C++ objects in order to make their references in algorithm traceable. The CArrayObjects are referenced in the code as native array types, therefore no other adjustment in the core algorithm is necessary.

Afterwards the memory units must be declared as in figure 1 in an external text file. An arbitrary number of architectures can be described and tested in parallel. The CArrayObjects are mapped into the main memory of every architecture.

After successful compilation of the project, XMSIM can be invoked from the command line. The file containing the memory architecture description is used as input. The result is a text file that reports all memory accesses. For the purposes of the demonstration the memory demanding algorithm cavity detector [5] is tested, which is used for image processing. The results obtained are identical with the results reported by DineroIV [6].

4. CONCLUSION AND FUTURE WORK

In this paper, a memory hierarchy evaluation framework for multimedia applications is presented. In the future the tool is aimed at 1) automating the necessary native-C type arrays to C++ object conversion, 2) support other types of memory architectures (eg scratch-pad).

5. REFERENCES

[1] F. Catthoor, K. Danckaert, K.K. Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. Achteren, and T Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Springer, 2002.

[2] FP7 ENOSYS project, *http://www.enosys-project.eu/*.

[3] T. Lioris, G. Dimitroulakos, and K. Masselos, "Xmsim: Extensible memory simulator for early memory hierarchy evaluation," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, july 2010, pp. 375 –380.

[4] CACTI An integrated cache, area leakage memory access time, cycle time, and dynamic power model, *http://www.hpl.hp.com/research/cacti/*.

[5] M. Bister, Y. Taeymans, and J. Cornelis, "Automated segmentation of cardiac mr images," in *Computers in Cardiology 1989, Proceedings.*, sep 1989, pp. 215 –218.

[6] Dinero IV Trace-Driven Uniprocessor Cache Simulator, *http://pages.cs.wisc.edu/ markhill/DineroIV/*.