XMSIM: EXtensible Memory SIMulator for Early Memory Hierarchy Evaluation

Abstract—**This paper presents a memory hierarchy evaluation framework for multimedia applications. It takes as input a high level C code application description and a memory hierarchy specification and provides statistics characterizing the memory operation. Essentially the tool is a specialized C++ data type library which is used to replace the application's data types with others that monitor memory access activity. XMSIM's operation is event driven which means that every access to a specific data structure is converted to a message towards the memory model which subsequently emulates memory hierarchy operation. The memory model is highly parametric allowing a large number of alternatives to be modeled. XMSIM's main advantage is its modularity allowing the designer to alter specific aspects of the memory operation beyond the predefined ones. The main features are the capability to: 1) simulate any subset of the application's data types, 2) 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, 5) verification utilities for the validation of code transformations.**

Keywords: Memory Simulation, Memory Hierarchy Evaluation Tools, Computer Aided Design, Code Transformations

I. INTRODUCTION

Microprocessor speeds have risen dramatically during the last years in discrepancy to current memory operation speeds. This fact indicates a major bottleneck in the processor-memory intercommunication, deteriorating system performance and increasing time delay. Moreover, applications in the embedded system domain require drastic reduction of power consumption for long life battery operation. 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 processors' implementation.

Nowadays system design flow is a two phase process. Initially the application is developed in a high level language (such as C/C++) by designers bearing an algorithmic background. More or less they provide a functional code which is by far not optimized. In the second phase the mapping of the application to a specific embedded processor is implemented and optimized. Today's mainstream practice requests collecting results of the application first and feeding the results to a separate profiling/optimization program afterwards. Our work facilitates the completion of the optimization process in the following ways: 1) application's code development and profiling are performed on the same development framework and 2) instant verification of the code transformation.

Usual practice is to develop algorithms in C language which is more close to hardware implementation. More specifically, a small subset of the C language's data types are used that is, arrays and scalars which are determined at compile time. Although there has been an extensive research on the automation of the optimization, it is still an area that human interaction is required. In most cases the designer should have a minimum knowledge about how the application works to perform optimizing decisions. Additional design effort can be saved if the algorithm's designer has an insight about how its decisions affect the performance of the memory system. Our work provides the designer with the ability to evaluate his decisions on the back-end of the hardware design flow.

In this paper, a memory hierarchy evaluation framework is proposed that unifies the algorithmic development and memory hierarchy optimization phase. It is an event-driven environment where the application's arrays ands scalars are substituted by objects with the same behavior, providing profiling capabilities. Profiling is accomplished by means of event generation when memory accesses take place that is, on access to a data structure. These messages bear information about data structure accesses and are forwarded to a memory model that records and translates them to memory accesses. The output of the tool is a series of profiling data, characterizing the memory hierarchy performance.

The advantages of the tool are: 1) The memory model is highly parametric and models a large number of memory hierarchy scenarios, 2) The overall algorithmic development, debugging and optimization take place under a single environment which is the Microsoft Visual Studio IDE [2]. 3) Verifying the program's integrity after code transformation's is made easy by verification routines developed for this purpose. 4) Multiple memory hierarchy scenarios can be examined simultaneously, 5) The designer can fully control data mapping to memories and the subset of data structures that he wants to simulate, 6) The tool is extensible that is, the designer can built new classes inheriting the default ones providing user defined analysis as well as configure the memory's operation by means of callback functions and 7) It can be extended to cooperate with existing tools (CACTI [3],[4]) that provide power estimation.

The paper is organized as follows: Section 2 includes the related work while Section 3 explains the simulator architecture. Section 4 describes the development environment Section 5 presents the experimental results, and finally Section 6 concludes the paper.

II. RELATED WORK

Most cache simulators nowadays target more on educational purposes than cache design. Although our work can be used on both disciplines, our main purpose is to facilitate code optimization towards cache efficient design. Currently there are two types of cache simulators: execution-

driven and trace driven. Execution-driven simulators ([5],[6],[7]) display cache operation, as well as cache-processor communication, giving a more detailed view of the computer system at the expense of simulation time and clarity. Trace-driven simulators [8] simulate the result on cache state and contents of a specific memory access trace which is fed as input to the simulator. Moreover, execution-driven simulators bind on specific processor architecture while trace-driven are architecture independent. From this point of view, our work is categorized to the trace driven division.

One of the first attempts to build a cache simulator was the PC-Spim [9] which was extended with the appearance of SpimCache [10]. Unfortunately, the latter was only suitable for one cache level simulations and could not alter cache characteristics like the line size or different policies. An extension of SpimCache was the SpimVista tool [6] intended to represent cache hierarchy and the interaction between cache levels. Simplescalar [5] is another representative example of an execution-driven cache simulator, as well as the mlcache simulator [7]. On the other hand, trace-driven simulators are easier to understand. Dinero IV [8] is a trace-driven simulator written in C where various cache design parameters (like cache associativity, write-back or write-through, cache line size etc) can be configured.

Our work differentiates in many aspects. In both types of existing simulators it is difficult to associate the C code statement executed and the particular action taking place in the cache because the input to all simulators is the assembly language. In our work the designer observes the results of every C statement in memory state and contents. Moreover, it is extensible such that new measurements can be introduced by the designer. Finally, it is equipped with the capability to verify correctness of the application's transformed code.

III. SIMULATOR ARCHITECTURE

The simulator engine consists of two main parts: 1) the Event Generation Engine and 2) the memory model. The Event Generation Engine is the core of XMSIM. It is a library of two template classes emulating the C language's arrays and scalars. Each array or scalar can be fully emulated from a properly parameterized template class. The main purpose of these classes is to record data structure access activity and produce events to experimental objects such as a virtual memory hierarchy or CPU. Since, these classes can fully substitute the C native types the application executes in an identical way providing a trace of events able to trigger update in state and contents of the experimental objects. The overall state transition can be studied either step by step or in summary making the evaluation possible.

Fig.1 illustrates XMSIM architecture. The tool engagement to the application involves two steps: 1) substitution of the application's arrays and scalars with the tool's data types and 2) mapping of these data types to Memory Model. As the application is executed, events originating from data type accesses are directed to the Memory Model, which responds with data flowing to and from the application. The whole simulation/analysis process is based on a well established development IDE, the Microsoft Visual Studio which is exploited to extended the tool's capabilities. Each simulation context consists of the application and XMSIM libraries. Both are compiled and linked as one C/C++ project and the execution provides the experimental results.

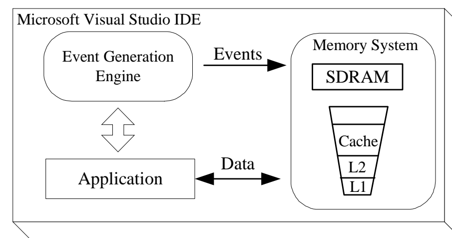


Figure 1. XMSIM architecture

The Event Generation Engine includes two classes one for arrays (called *CArrayWrapper*) and one for scalars (*CScalarWrapper*). It uses operator overloading [11] to emulate all native C language's operators by user defined ones functionally equivalent but with the ability to profile and driving experimental objects through the use of events. For this reason, each template class includes an array of pointers to objects corresponding to an abstract class [11] called *CContext* representing the interface of experimental objects to the Event Generation Engine. The *CContext* class consists of a set of member functions each corresponding to a type of event. To support event capture and service each class representing an experimental object should be derived from the class *CContext.*

The Memory Model is a set of classes each corresponding to different types of memory technology. All the classes are derived from the *CContext* class to support event handling and interface with the Event Generation Engine. Currently our framework supports parametric models for SDRAM, static RAM, cache and scratch-pad memory. Each, memory object includes pointers to other memory objects so as to model memory hierarchies. In addition, for the event handling each memory class provides specialized implementation of the member functions inherited from *CContext* according to its operation.

To summarize, events are generated upon access to any scalar or array data structure. That is in every operator function inside *CArrayWrapper* and *CScalarWrapper*. Events take the form of calls to *CContext* member functions inside each experimentation object. The objects handle these messages by altering their state and contents according to the information conveyed by each message. The carried information includes the type of data access (read or write), virtual address, etc. Fig.2 depicts the event generation and handling mechanism.

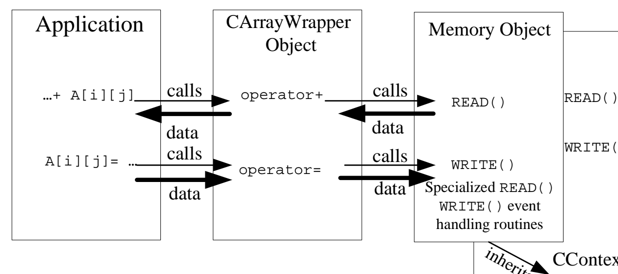


Figure 2. Event generation and handling

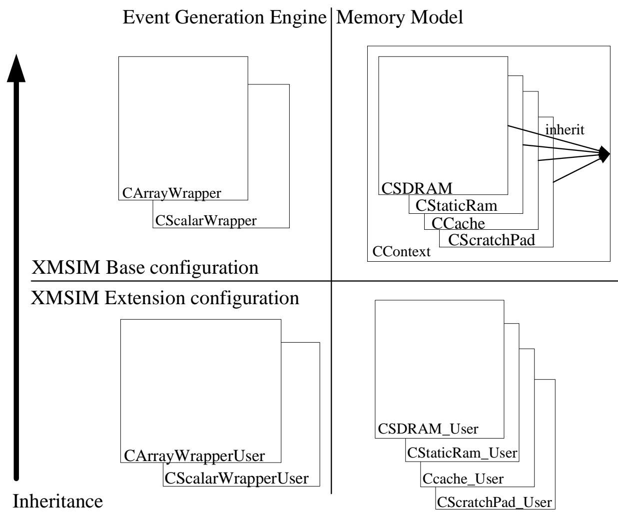


Figure 3. XMSIM Extensibility options

The overall tool architecture gives prosperous ground to extend its functionality. Every class in both the Event Generation Engine and the Memory Model can be inherited by a user defined class to widen the functionality and simulation utilities. Hence, inheritance [11] provides event handling support necessary to trace data type access events and gives user defined opportunities to extend memory access analysis and behavior. Fig.3 summarizes the extensibility options of XMSIM. The designer may combine one of the Event Generation Engine components (base or extended) with one of the Memory Model's. Totally there are 4 combinations by which the base and extended components can be combined. By extending the Event Generation Engine new opportunities of analysis emerge from the application's data types. On the other hand by extending the Memory Model classes, memories with different characteristics and functions can be introduced. Moreover, new analysis scenarios can be built based on the XMSIM event driven concept.

Moreover, having one conceived the organization of the project can alter or enhance at will the behavior of the memory models. The core of the functionality of the Memory Model Service lies within the four main routines of each memory class, which are the MAP, ADDRESS, WRITE and READ functions. These functions can be re-written to fit one's needs (like time delays from data transfer or specific sorts of misses), ornamented with enhanced functionality (like split caches or write-around policy), or even extended by calling additional routines. Furthermore all the classes can be extended in order to implement brand new concepts of memory layout. Thus the researcher can model any memory unit and test any memory hierarchy, restricted merely by his/her programming aptitude and his ambition.

The following example illustrates how one aspect of XMSIM functionality can be configured by rewriting the callback function ADDRESS(). The default realization of XMSIM assumes row-wise mapping of array data to memory. The example shows how the row-wise mapping can be transformed to column-wise. In XMSIM the Physical Address (PA) of an array data value in memory is the sum of the array's Base Address (BA) and the Virtual Address (VA). The latter determines the ordering of array elements in memory. The ADDRESS() function calculates the virtual address and provides this value to the memory model to compute the physical address. The physical address is estimated from the following equation

$$PA = BA + VA \quad (1)$$

The VA for row-wise mapping is given by the following equation

$$VA = I_n + \sum_{i=1}^{n-1}\left(\prod_{j=i+1}^{n} N_j\right)\cdot I_i \quad (2)$$

where n is the number of array dimensions and Nj is the size of the j-th dimension. On the other hand, column-wise mapping is realized by equation (3). Fig.4 shows the two versions of the ADDRESS() function corresponding to the row and column-wise mapping respectively.

$$VA = I_1 + \sum_{i=2}^{n}\left(\prod_{j=1}^{i-1} N_j\right)\cdot I_i \quad (3)$$

ROW-WISE MAPPING

```
template <typename T>
void CArrayWrapperT<T>::ADDRESS(unsigned int &VirtualAddress) const{

        unsigned int i,j;
        int icoef ;

        // MACROS
        // ARRAY_DIMENSIONS    : Returns the number of array dimensions
        // DIMENSION_SIZE(i)   : Size of the ith dimension
        // ARRAY_SUBSCRIPT(i)  : Value of the ith array subscript
        VirtualAddress =0;
        for ( i = ARRAY_DIMENSIONS-1; i>= 0 ; i-- ){
                icoef = 1;
                for ( j=ARRAY_DIMENSIONS-1; j>=i+1; j--){
                        icoef *= DIMENSION_SIZE(j);
                }
                VirtualAddress += icoef*ARRAY_SUBSCRIPT(i);
        }
        return;
}
```

COLUMN-WISE MAPPING

```
template <typename T>
void CArrayWrapperT<T>::ADDRESS(unsigned int &VirtualAddress) const{

        unsigned int i,j;
        int icoef ;

        // MACROS
        // ARRAY_DIMENSIONS    : Returns the number of array dimensions
        // DIMENSION_SIZE(i)   : Size of the ith dimension
        // ARRAY_SUBSCRIPT(i)  : Value of the ith array subscript
        VirtualAddress =0;
        for ( i = 0; i< ARRAY_DIMENSIONS-1 ; i++ ){
                icoef = 1;
                for ( j=0; j>=i-1; j++){
                        icoef *= DIMENSION_SIZE(j);

                }
                VirtualAddress += icoef*ARRAY_SUBSCRIPT(i);
        }
        return;
}
```

Figure 4. ADDRESS() function transformation example

IV. DEVELOPMENT ENVIRONMENT

In the sequel the environment supporting the design, analysis and optimization phases will be described. The overall framework is based on a popular development environment the Microsoft Visual Studio. It is an extensible and mature environment used extensively in many contexts. Microsoft Visual Studio provides an extensive set of debugging capabilities step by step execution and trace of the application's variable values.

The experimental setup requires the memory architecture, the application and the variables' set to examine. Memory hierarchy scenarios are described in a flat input file in a very simple language format. The parser instantiates and links the memory objects to form memory hierarchies. The tool informs about the memories made, or about erroneous input. Many memory hierarchies can be imported in a single input file. Thus one can simultaneously challenge a code against alternative memory hierarchies.

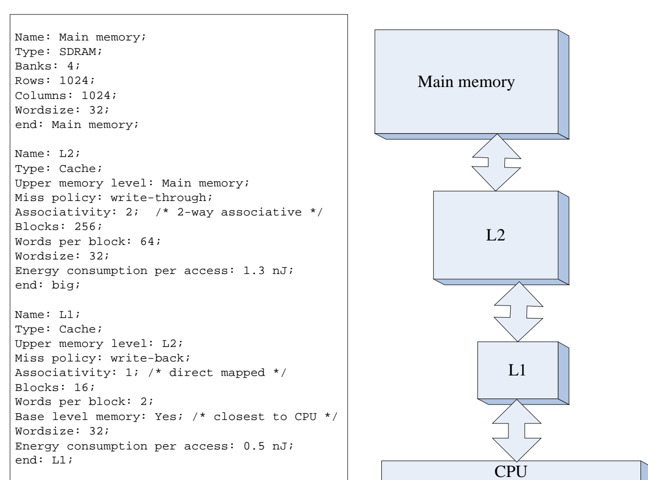


Figure 5. Memory hierarchy description file example.

The language used at the input file to depict the memory model is straight-forward and easy to understand. Every command must take up a single line and be delimited by the semicolon (;) as in C/C++ grammar. All commands are of the form: "parameter: value" and the command repertoire hardly exceeds fifteen members. The script can include one-line comments after the hash sign (#) at the beginning of the line. Every memory unit is uniquely named and declared separately from the rest. The directives describe physical characteristics of every memory unit such as the number of blocks, words of every memory block, the size of every word in bits, number of banks in the case of SDRAM as well as operation policies like write-back/write-through and associativity. Energy consumption can be defined and the way different memory units are connected to form a multilevel hierarchy. Fig.5 shows an example of memory hierarchy description file.

The second stage involves two steps. The first concerns the replacement of the application's data types with the ones provided by XMSIM. At this point the designer may decide to have two versions of the code the one engaged to XMSIM and the other being the original. This option is essential to verify code correctness when the designer applies code transformations. For this reason, XMSIM provides verification routines in *CArrayWrapper* and *CScalarWrapper* classes to compare equality between the XMSIM's objects and the C language's data types. Fig. 6 depicts how XMSIM transforms C language data types to XMSIM data type declarations. The outcome of this process is the application's statements to remain untouched operating on XMSIM's data types.

```
//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");
```

Figure 6. XMSIM engagement example

In the second step the designer has the ability to map the XMSIM's arrays and scalars to specific memory locations. If this step is overridden for any data type then XMSIM places it automatically to main memory following a sequential placement on empty address space. Fig.7 illustrates the assignment of a specific memory address to an array while Fig.8 depicts the application's code layout after XMSIM engagement. As it is obvious, nothing is changed in the code except for the data type declarations.

```
unsigned int base_address = 0xFFA3;

ATTACH_ARRAY_TO_MEMORY(array, main_memory);

ATTACH_ARRAY_TO_MEMORY(array, main_memory, base_address);
```

Figure 7. Attachment of an array to memory example

The ultimate goal is to evaluate an application on a memory hierarchy in respect to power, area and speed metrics. For this reason XMSIM can be used in cooperation with existing power, time and area estimation kits developed for the different memory technologies supported by XMSIM. The CACTI tool [3] can be used for cache and scratch-pad [12] while for SDRAM exist power and speed estimation models such as the ones provided by Micron [4]. Before running the simulation, the aforementioned tools supply power, area and time metrics for each memory. The outcome of simulation provides the operational profile of the application in terms of power speed and area.

Finally, there are two spots in XMSIM where the designer can broaden its utilities. The first involves the Event Generation Engine while the second the Memory Model. XMSIM provides default functionality that refers to the analysis of the data structure access patterns and also the memory system performance. The designer can extended the analysis utilities following the principles described in Section 3 to record the frequency by which array elements are accessed, data locality etc.

```
void cav_detect(){ Original Application Code

/*C declarations */
static unsigned char gauss_x_image[N][M];
static unsigned short gauss_x_compute[N][M][(2*GB)+2];

/* C Statements */
for (x=GB; x<=N-1-GB; ++x){
  for (y=GB; y<=M-1-GB; ++y) {
        gauss_x_compute[x][y][0]=0;
           for (k=-GB; k<=GB; ++k)
                  gauss_x_compute[x][y][GB+k+1] =
           gauss_x_compute[x][y][GB+k] +
(image_in_traf[x+k][y]*Gauss[abs(k)]);
                  gauss_x_image[x][y]=
                     gauss_x_compute[x][y][(2*GB)+1]/tot;
   }
}
…
…
}
```

```
void cav_detect(){ Application Code after XMSIM engagement

/*C declarations */
/* Step 1 data type substitution */
//static unsigned char gauss_x_image[N][M];  // commented
//static unsigned short gauss_x_compute[N][M][(2*GB)+2]; // commented

int Dim1[2] = {N,M};
int Dim2[3] = {N,M,2*GB+2};
static CArrayWrapperT<unsigned char> gauss_x_image(2,Dim1,"gauss_x_image");
static CArrayWrapperT<unsigned short> gauss_x_image(2,Dim2,"gauss_x_compute");

/* Step 2 (optional) map arrays to specific memory segments */
unsigned int base_address = 0xFFA3;
ATTACH_ARRAY_TO_MEMORY(gauss_x_image, main_memory, base_address);

/* C Statements */
for (x=GB; x<=N-1-GB; ++x){
   for (y=GB; y<=M-1-GB; ++y) {
         gauss_x_compute[x][y][0]=0;
            for (k=-GB; k<=GB; ++k)
                  gauss_x_compute[x][y][GB+k+1] =
                  gauss_x_compute[x][y][GB+k] + (image_in_traf[x+k][y]*Gauss[abs(k)]);
                  gauss_x_image[x][y]= gauss_x_compute[x][y][(2*GB)+1]/tot;
   }
}
…
…
}
```

Figure 8. Example of code layout

V. EXPERIMENTS

Our experimental setup consists of the base XMSIM configuration and an image processing application called cavity detector [13]. Basically, cavity detector is a medical application consisting of three loop kernels. The first performs lowpass filtering to blur the input image, the second applies an edge detection algorithm and the third derives the negative of the image picture. For the experiments an optimized version of the cavity detection is also derived. A series of loop transformations [1] are applied to the original code focusing on data locality improvement. Subsequently XMSIM's utilities verified the correctness of the loop transformation, prior experimentation. The resulting code has one loop kernel corresponding to the union of the initial three.

Since there are many memory configuration parameters any many values to decide among them, a huge design space is created. The following experiments intent to give an overview of the feedback provided from XMSIM during the algorithm development or optimization phase and not how a fully optimized memory hierarchy is produced for the testbench application. In this context, the measurements presented concern the cache's and SDRAM's observed performance challenged over different major characteristics. In respect to the cache, these include the cache bank size, the number of blocks, the number of words per block and the number of cache hierarchy levels. Regarding the SDRAM these include various internal organizations concerning the cache rows and columns size. Finally, performance is evaluated with either of the following: the memory hits or misses, and the total memory references (read/write).

The graph in Fig.9 depicts cache hits and misses for the initial and optimized (traf) cavity algorithm for one level direct-mapped write-through cache consisting in each case of 256 block of 4 words, 128 blocks of 8 words, 62 blocks of 16 words and 32 blocks of 32 words. As words per block increase we get a better performance for both algorithms while the optimized version of cavity has a lower cache activity because the sum of hits and misses is reduced in the case of the optimized version. Fig.10 illustrates cache performance for the initial and optimized (traf) cavity algorithm for one write through cache consisting of 256 block of 4 words, when sets are 1 (direct-mapped), 2(2-way associative), 4 and 8. Numbers are in thousands. We see a slightly better performance when sets are increased. Fig.11 depicts SDRAM row hits and misses [14] at various architectures for original and optimized cavity algorithm. From a range of 262144 rows by 8 columns up to 2048 by 1024 columns we see that hits constantly rise when the number of columns increases.

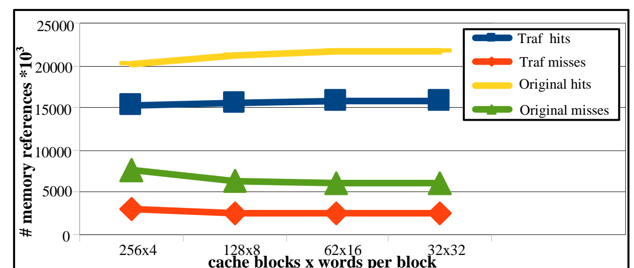


Figure 9. Cache performance figures for various cache block configurations

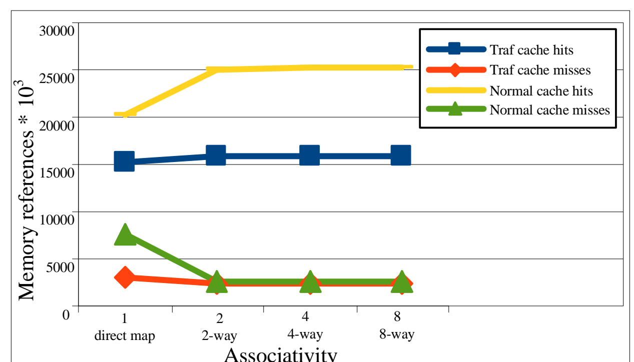
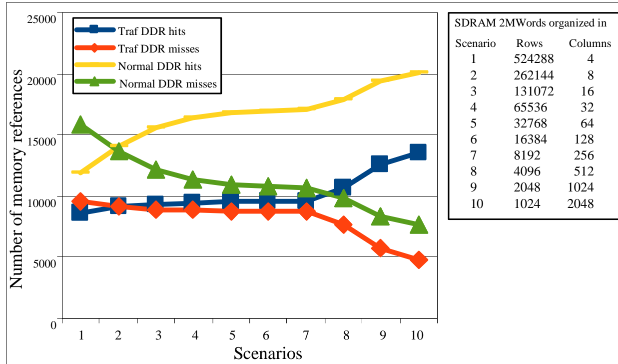


Figure 10. Cache performance figures for various cache assoviativities

Fig.12 and 13 present the explored memory hierarchy configurations for the two different versions of cavity detector. In all cases the main memory is an SDRAM memory chip with 4 banks each organized in 2048 rows and 1024 columns. Totally, three different cache banks have been used in the various scenarios with the following characteristics: 1) Bank C1 is a 128Kb 4-way write back cache with 128Kblocks and 1 word per block ,2) Bank C2 is a 64Kb 4-way write back cache with 64KBlocks and 1 word per block and 3) Bank C3 is a 256Kb 4-way write-back cache with 256Kblocks and 1 word per block.



SDRAM 2MWords organized in

Scenario	Rows	Columns
1	524288	4
2	262144	8
3	131072	16
4	65536	32
5	32768	64
6	16384	128
7	8192	256
8	4096	512
9	2048	1024
10	1024	2048

Figure 11. SDRAM Performance figures

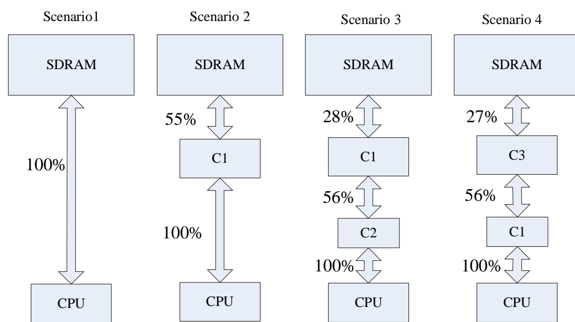


Figure 12. Memory References for the optimized version of cavity

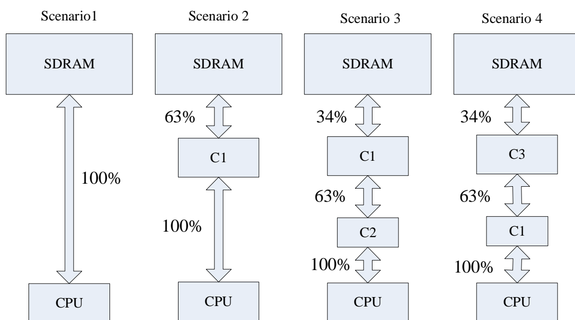


Figure 13. Memory References for the initial version of cavity

Fig. 12 and 13 exposes the experimental results over the aforementioned memory hierarchies for the optimized and initial version of cavity detector respectively. In detail the number of memory references among two adjacent levels of memory hierarchy has been recorded. The results have been normalized for clarity with the number 18288756 and 27724634 representing the 100% in Fig.12 and 13 respectively. Finally, it must be noticed, that the overall experimental results have been derived in a very short amount of time which includes the editing of the memory configuration file and the execution of the testbench application. Hence, the proposed tool gives the opportunity to challenge a large number of scenarios in a direct and immediate way.

VI. CONCLUSIONS

To conclude we believe that XMSIM, in its present form, is an evaluation framework that facilitates the software designer in the exploitation of existing memory hierarchies or the derivation of new ones for a given application. The framework is equipped with validation routines to guarantee correctness of the transformations imposed on the application. The designer can amend the XMSIM's C++ native code, in order to extend the memory classes or enhance the functionality of the existing ones. Additionally, the tool provides platform independent exploration which is by far easier to understand and manipulate than existing execution-driven simulators that delve into the details of memory processor communication. On going work, is directed to the development of a graphical interface to further automate input and output, producing graphics for direct result analysis, best case finding and so on.

REFERENCES

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

[2] http://msdn.microsoft.com/en-us/vstudio/default.aspx

[3] N. Muralimanohar, R. Balasubramonian, N. P. Jouppi, CACTI 6.0: A Tool to Model Large Caches, Technical Rep. HPL-2009-85 HP Laboratories.

[4] http://www.micron.com/support/part_info/design_analysis_kits, 2010

[5] http://www.simplescalar.com/,2010

[6] Leticia Pascual, Alejandro Torrentí, Julio Sahuquillo and Josẻ Flich, “Understanding cache hierarchy interactions with a program-driven simulator”, Proceedings of the 2007 workshop on Computer architecture education, pp. 30-35, 2007

[7] by Edward S. Tam , Jude A. Rivers , Gary S. Tyson , Edward S. Davidson, “mlcache: A Flexible Multi-Lateral Cache Simulator”, Proceedings of MASCOTS'98, 1998

[8] J. Edler, M. Hill, Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://www.cs.wisc.edu/~markhill/DineroIV/,2010

[9] http://pages.cs.wisc.edu/~larus/spim.html,2010

[10] Sahuquillo, J. Tomas, N. Petit, S. Pont, A, ”Spim-Cache: A Pedagogical Tool for Teaching Cache Memories Through Code-Based Exercises”, Education, IEEE Transactions on, pp. 244-250, Aug.2007

[11] Bjarne Stroustrup, "The C++ Programming Language", Addison Wesley, Special Edition, 2008

[12] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel, “Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems”, 10th International Symposium on Hardware/Software Codesign, pp. 73-78, 2002

[13] K. Danckaert, F. Catthoor and H. De Man, “Platform independent data transfer and storage exploration illustrated on a parallel cavity detection algorithm”, CSREA Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 1669-1675, 1999

[14] Bruce Jacob, Spencer Ng, David Wang, “Memory Systems: Cache, DRAM, Disk”, Morgan Kaufmann, 2007