Exploring Memory Hierarchy with ArchC

Pablo Viana, Edna Barros
Federal University of Pernambuco
Informatics Center
Recife-PE, Brazil
{pvs, ensb}@cin.ufpe.br

Sandro Rigo, Rodolfo Azevedo, Guido Araújo
University of Campinas
Institute of Computing
Campinas-SP, Brazil
{srigo, rodolfo, guido}@ic.unicamp.br

Abstract

This paper presents the cache configuration exploration of a programmable system, in order to find the best matching between the architecture and a given application. Here, programmable systems composed by processor and memories may be rapidly simulated making use of ArchC, an Architecture Description Language (ADL) based on SystemC. Initially designed to model processor architectures, ArchC was extended to support a more detailed description of the memory subsystem, allowing the design space exploration of the whole programmable system. As an example, it is shown an image processing application, running on a SPARC-V8 processor-based architecture, which had its memory organization adjusted to minimize cache misses.

1. Introduction

In accordance with what was predicted by Moore's law, the chip capacity continues doubling at every 18 months. Nowadays, System-on-chip (SoC) technology can integrate, into a single chip, entire digital systems consisting of numerous components like microprocessors, memories, coprocessors, and peripherals that used to occupy one or more boards [17].

This progress has led the growth on the electronics market and its applications, since lower costs and higher performances attract technology consumers. However, it has imposed a pressure over design technology communities to cope with all this increasing potential. Although silicon process technology continues to evolve at an accelerated pace, design reuse and design automation technology are now seen as the major technical barriers to progress, and this productivity gap is increasing rapidly[11].

Making use of the available technology, advances in design methodologies have been proposed, and as a result of this effort, the Platform-based design paradigm has been adopted as a good way for reusing and rapid development of digital systems.

A typical SoC architecture usually has a processor core, one or more caches, on-chip bus hierarchy, on-chip memory, and a large number of peripheral cores that provide application-specific functions such as multimedia and communication processing (Figure 2). Following the Platform-based design methodology, each of these SoC cores are parameterized, enabling the designer to tune a core's settings for a specific application.

An assignment of different values to each parameter will impact the overall performance and power consumption of the SoC architecture [7]. Thus, for the correct assessment of a platform configuration, it is interesting to have a model of the whole system, able to rapidly simulate the application behavior as well as reflect the key features of the customized architecture before its real implementation.

By means of an architecture description language (ADL) like ArchC [14], the architecture of a programmable system can be modeled. It is possible to map a software application over this model, and the simulation of the software running on the architectural model helps the designer to evaluate the performance and adjust the parameters of the system.

Taking into account the importance of the memory hierarchy on the performance of programmable systems, an extension of the storage model adopted by ArchC is proposed. With this addition, ArchC is presented as a promising architecture description language (ADL), able to capture the main features of the processor description, as well as the memory subsystem. Through this complete model, the architecture can be explored by adjusting the platform parameters, in order to tune it for a specific application, and find the platform configuration that offers the best performance.

This paper is organized as follows: Section 2 presents the main works related to our research; Section 3 illustrates the system's components modeling with ArchC, by presenting the use of the language for processor modeling and the advances obtained by extending the language to model memory hierarchies. Section 4 presents the architecture simu-

lator automatically generated by ArchC for platform tuning. An example in Section 5 illustrates the exploration of memory configurations aiming to improve the performance of an image processing application. The results are showed in that section, where interesting performance numbers are analyzed. Section 6 remarks the next goals and finally, Section 7 concludes this paper, summarizing the main ideas and experiments that have been done.

2. Related Work

Methodologies like the Platform-Based Design paradigm [11] have been adopted for the development of digital systems. Supported by such a platform, the digital system design may be started from an architecture description where the application's behavior can be mapped on. According to the system requirements and constraints, the system modules are tuned towards higher performance and lower cost (Figure 1).

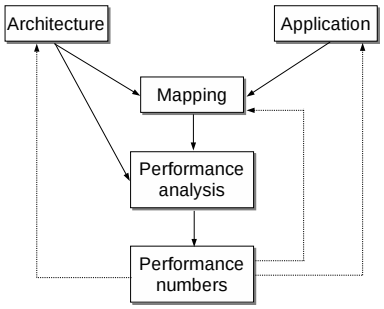

Figure 1. Y-chart scheme for design space exploration.

The clear distinction between application and architecture enables a systematic exploration of the architecture, tuning it up to an application domain. Following this methodology, the designer maps the behavioral description into a specific architecture and the performance analysis helps the designer to adjust architecture parameters, adapting it to the application, aiming to achieve the desired performance.

During the performance analysis metrics including execution time, power consumption, silicon area, cache miss rate etc, are considered. Such metrics depend on system configuration parameters such as supply voltage, memory size, silicon technology, cache policies, etc. By modifying the configuration parameters of the architecture description, the designer is faced with a huge space of possibilities to explore, composed by a lot of implementation alternatives and their respective system performance features. In fact, accurate results for performance could only be measured after the final implementation of the system on the chip. However, good estimates can be early obtained if an executable model of the system is available for simulation.

Modeling is the act of representing a system or a subsystem formally [4]. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or dimensions. A model can also be constructive, which are also called executable models. Depending on the abstraction level of a constructive model, more accurate details can be captured, however its execution tends to be slower.

The Dalton project [17] from the University of California at Riverside, has focused at the development of project methodologies related to IP cores tuning for low power. The power consumed by an application running on a particular platform can be evaluated by means of an executable model. Each IP core is modeled as a class of an Object-oriented programming language like C++ or Java, in such a way that the entire system is modeled as communicating objects. The methods (functionalities or instructions) available for each object is low-level power analyzed and back-annotated using a variety of parameter configurations and instruction data values. By executing the complete C++ model for a particular platform configuration, the designer obtains accurate power and performance information.

Based on this fast power evaluation, methods to efficiently explore the configuration space can be investigated. Pareto-optimal configuration points are those that offers the best performance for a given cost [7]. In this way, configuration points that tends to be more expensive with no performance improvement are discarded from the design space exploration (DSE). The design space pruning of non-Pareto-optimal points limits the design space, reducing the time needed to find a suitable configuration for a given application. As a proposal of an integrated environment for the design space exploration guided by Pareto-optimal, the Platune tool [8] presents some interesting results that can be obtained from an architecture configuration. This tool enables the exploration of memory or bus sizes and energy supply by simulating an application, running over an architecture composed by a MIPS processor connected to independent single-level instruction and data caches. The source code in C of the application is compiled by the tool, while candidate configurations are selected from a sort of possibilities, such as varying cache block size, associativity, word width, supply voltage, etc. The exploration of candidates can be made through exhaustive simulation of all possible combinations or by a Pareto-optimal pruning.

On the exploration of cache configurations based on their hit and miss rates, Dinero [1] is largely utilized. This tool was developed at the University of Wisconsin and had already its fourth version released. Dinero takes as input a

trace file, describing the memory accessing of data and instruction. Actually, Dinero is neither a timing nor a functional simulator. It is only based on the address tags of the references, and it has not any data or instructions to move in and out of the caches. Trace-driven simulation is frequently utilized to evaluate memory hierarchy performance. By using the same trace sequences, distinct cache configurations can be analyzed.

A trace is usually obtained by the execution of a program or set of programs in a simulator. Shade [3] combines efficient instruction-set simulation with the trace generation capability. It consists in an assembly-based simulator of processor instructions. Even though the software description in such low level language favors fast and flexible modeling, the powerful abstraction offered by Object-oriented languages seems to be, of course, a more attractive way to describe entire systems.

On this way, Architecture Description Languages (ADLs) can help the designer to create an executable model of the system to simulate the application running over the proposed architecture. Most of the ADLs are based on a parser that extracts information from a brief architecture description, translating into a source code written in a traditional programming language. This source code is compiled and so executed as a simulation of the architecture. In [16], ADLs are classified as 1)Synthesis-oriented, 2)Compiler-oriented, 3) Simulation-oriented, 4) Validation-oriented among other approaches.

EXPRESSION [10], classified as a compiler-oriented ADL, is an architecture description language focused on architectural design space exploration for SoCs and automatic generation of a compiler/simulator toolkit. Designed to support DSE of a wide class of processor architectures, ranging from RISCs, DSPs, ASIPs and VLIWs, coupled with a variety of memory system organization and hierarchies, EXPRESSION contains an integrated specification of both structure and behavior of processor-memory systems, supporting the specification of novel memory hierarchies.

ArchC also allows the mixed-level paradigm of behavioral/structural architecture description. To the best of our knowledge, ArchC is currently the only ADL totally based on SystemC [9]. ArchC has many features that distinguish it from other ADLs, such as non-proprietary SystemC compatibility and instruction format behavior modeling. However, its key feature is a storage-based co-verification mechanism that automatically checks the consistency of the refined SystemC RTL model against the SystemC behavioral model generated by ArchC. [14].

The gathering of all features offered by ArchC, enable the entire system modeling, where the processor and memory descriptions can be correctly refined, or even to be connected to others SystemC IPs cores composing a complete system, like the one showed in the Figure 2.

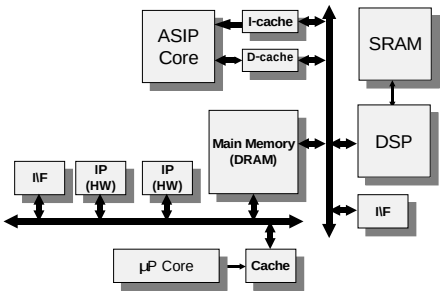

Figure 2. Example of an architecture.

3. System Modeling with ArchC

A SoC design can be started by the definition of the application, followed by its implementation by means of a software programming language. Usually, designers utilize a C/C++ compiler to generate an initial software prototype of the application's behavior. The next step is to choose an architecture, that will run the application to compose the digital system. ArchC enables designers to describe the processor element as well as the memory modules at a high abstraction level, automatically generating the software toolkit, formed by the assembler, the simulator, and also a powerful co-verification tool that will check the description throughout the refinement steps.

Aiming to develop a system to run an image processing algorithm, we started from an architecture composed by one processor and some cache and memory modules. This architecture can be seen in Figure 10. Notice that this first description of the architecture is highly dependent on the designer's experience, defining some of the main components of the system configuration.

In order to compose our platform, we chose to model the Leon[6] processor in ArchC. Leon is a real world SPARC V8 implementation, first developed by Jiri Gaisler at European Space Agency (ESA) to enable the development of system-on-a-chip (SoC) devices using the LEON core. Plenty of technical documentation about the Leon, as well as the full VHDL source code is freely available under the GNU LGPL license. Readers should refer to [15, 13] for more details about the SparcV8 architecture.

3.1. Processor Description using ArchC

ArchC is an architecture description language (ADL) specialized for processor architecture description. Its main goal is to facilitate and accelerate processor description, combined with enough expression power to model several classes of architectures (RISC, CISC, DSPs, etc), allowing users to explore a new ISA and automatically generate software tools like assemblers and simulators.

A processor architecture description in ArchC is divided in two parts: the `Instruction Set Architecture (AC_ISA)` and the `Architecture resources (AC_ARCH)` descriptions. Into the `AC_ISA` description, the designer provides to ArchC details about instruction formats, size and names combined with all information necessary to decoding and the behavior of each instruction. The `AC_ARCH` description informs ArchC about storage devices, pipeline structure etc. Based on these two descriptions, ArchC will generate a behavioral simulator written in SystemC for the architecture.

The modeling of the processor features in ArchC starts with a detailed architecture study, gathering knowledge about the instruction set, pipeline, register bank system and the special purpose registers. Figure 3 presents the architecture resources description for our Leon model.

```
AC_ARCH(leon){

  ac_mem MEM:256K;

  ac_regbank RB:520;
  ac_reg PSR, Y, WIN;

  ac_pipe   pipe = {IF, ID, EX, MEM, WB};

  ARCH_CTOR(leon){
    ac_isa("leon_isa.ac");
  };
};
```

Figure 3. Leon (SparcV8) architecture declaration in ArchC.

Register window is a typical feature in Sparc architectures [13]. In order to be able to simulate this feature, our implementation uses the ArchC keyword for register bank declaration (`ac_regbank`) to allocate a register bank comprising 520 registers. These are 8 general purpose registers, along with 512 registers divided into 32 register windows of 16 registers each. Taking a look at the example, notice that special purpose registers (`PSR, Y` and `WIN`) are declared through the `ac_reg` keyword, immediately followed by the 5 stage pipeline declaration using `ac_pipe`.

The second part of our Leon model is the instruction set description (`AC_ISA`), which is illustrated by Figure 4. In ArchC, a format is declared through the keyword `ac_format` and initialized with a string that represents its subdivision into fields. We have used five different formats to represent the SparcV8 instruction set. The example shows some instruction declarations (`ac_instr`), with their respective formats associated. Inside the instruction-set constructor (`ISA_CTOR`), the designer has to provide some information about each instruction, which basically is their assembly syntax and decoding data. Based on the `AC_ISA` description, ArchC is capable of automatically building a decoder for the architecture being modeled. Decoders generated by ArchC are not restricted to a specific class of architectures, being able to work with instruction sets that are not as regular as the Sparc's (RISC) ISA, including multi-cycle instructions with variable length. The decoding information is passed to ArchC using the `set_decoder` method, which expects as argument a list of pairs <*field* = *value*> that informs which fields have to be checked in order to identify each instruction.

```
AC_ISA(leon){

  ac_format Type_F1   = "%op:2 %disp30:30";
  ac_format Type_F2A  = "%op:2 %rd:5 %op2:3 %simm22:22";
  ac_format Type_F2B  = "%op:2 %an:1 %cond:4 %op2:3 %disp22:22";
  ac_format Type_F3A  = "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %asi:8 %rs2:5";
  ac_format Type_F3B  = "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %simm13:13";

  ac_instr<Type_F1>  call;
  ac_instr<Type_F2A> sethi;
  ac_instr<Type_F2B> ba, bn, bne, be, bg, ble, bge, bl;
  ac_instr<Type_F3A> ldsb_reg, ldsh_reg, ldub_reg, lduh_reg;
  ac_instr<Type_F3B> ldsb_imm, ldsh_imm, ldub_imm, lduh_imm;
  ac_instr<Type_F3B> stb_imm, sth_imm, st_imm, std_imm;

  ISA_CTOR(leon){

    ldsb_reg.set_asm("ldsb %rd, %rs1, %rs2");
    ldsb_reg.set_decoder(op=0x03, op3=0x09, is=0x00);

    ldsh_reg.set_asm("ldsh %rd, %rs1, %rs2");
    ldsh_reg.set_decoder(op=0x03, op3=0x0A, is=0x00);

    ldub_reg.set_asm("ldub %rd, %rs1, %rs2");
    ldub_reg.set_decoder(op=0x03, op3=0x01, is=0x00);

    std_imm.set_asm("std %rd, %simm13(%rs1)");
    std_imm.set_decoder(op=0x03, op3=0x07, is = 0x01);

    bvc.set_asm("bvc %an, label");
    bvc.set_decoder(op=0x00, cond=0x0F, op2=0x02);
          ...
};
```

Figure 4. Leon (SparcV8) Instruction set declaration in ArchC.

Finally, the designer has to provide to ArchC is the behavior of each instruction. This is done through the `ac_behavior` method, like showed in Figure 5. In the example, `readReg` and `writeReg` are functions that we have implemented, using the usual `read` and `write` methods provided by ArchC for all storage devices, to simulate register windows. This is an example of a cycle-accurate behavior description in ArchC. This is intended for more precise performance measurements, where the designer informs, for each instruction, what it exactly does at every pipeline stage. Notice that registers are being read at the second stage, the actual operation is executed in the third stage and finally, the result is stored in the correct destination during the fifth stage.

Another valuable feature in ArchC are its *format* and *generic instruction* behavior, which are behavior methods that can be declared for a given instruction format or even for all instructions. During simulation, when a instruction

```
void ac_behavior( add_reg ){

  switch( stage ) {
  case _IF:
    break;
  case _ID:
    ID_EX.rs1 = readReg(RB, rs1);
    ID_EX.rs2 = readReg(RB, rs2);
    break;
  case _EX:
    EX_MEM.alures = ID_EX.rs1 + ID_EX.rs2;
    EX_MEM.regwrite = ID_EX.regwrite;
    EX_MEM.memread = ID_EX.memread;
    EX_MEM.memwrite = ID_EX.memwrite;
    EX_MEM.rdest = ID_EX.rd;
    break;
  case _MEM:
    break;
  case _WB:
    writeReg(RB, EX_MEM.rdest, EX_MEM.alures);
    break;
  default:
    break;
  }
};
```

Figure 5. Leon (SparcV8) instruction behavior description in ArchC.

is fetched the generic instruction behavior executes first, followed by corresponding instruction format behavior, followed by specific instruction behavior. In other words, when an `add_reg` instruction is fetched and decoded, the simulator first executes the generic instruction behavior, then executes the `Type_F3A` behavior and terminates this simulation step by executing the `add_reg` behavior. Such features ease a lot the codification of *register forwarding* and *hazard detection* by factorizing a large amount of operations that must be performed, at every single clock cycle, by all instructions of a given format. The reader may refer to [14] for more details on ArchC's syntax, semantics and features.

3.2. Memory Hierarchy Modeling

Originally, storage device objects in ArchC were all instances of the same base class, called `ac_storage`. This class provides a number of simulation features, like delayed assignment support, update logs for simulation debugging, statistics collection for architecture exploration, support for various architecture wordsizes, etc. Considering how data is stored, these objects act like containers, i.e., data is stored into arrays and accessible through `read` and `write` methods. In order to allow a more detailed description, including a memory subsystem composed by caches and memories, this storage base class was inherited, extending the ArchC Class Hierarchy, as shows the Figure 6.

An object of the `ac_cache` Class is defined by the setting of the following attributes: number of words in each block, number of blocks in the cache, set associativity, replacement policy and the writing scheme. By setting these parameters the device can be configured to be a direct-mapped, set-associative or a fully-associative cache, and defines its behavior during the read and write operations.

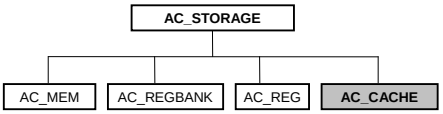

Figure 6. Storage device Class hierarchy.

By using the `ac_cache` class, it is possible to describe the architecture of the Figure 3 in a more detailed way, defining different cache modules to compose a memory hierarchy. As exemplifies the declarations of the Figure 7, `icache`, `dcache` and `ul2cache` are the names given by the designer to three different cache modules, in order to compose a memory subsystem. According to the parameters declared for each module in that example, a direct-mapped cache with 128 blocks and write-through scheme named `icache` will be instantiated, as well as a two-way cache, with four words by line, replaced by LRU policy (`dcache`) and a second level cache of 4096 words (`ul2cache`).

```
AC_ARCH(leon){

  ac_cache   icache("dm", 128, "wt")
  ac_cache   dcache("2w", 64, 4, "wt", "lru")
  ac_cache   ul2cache("dm", 4k, "wt")

  ac_regbank RB:520;
  ac_reg PRS, Y, WIM;

  ac_pipe   pipe = {IF, ID, EX, MEM, WB};

  ARCH_CTOR(leon){
    ac_isa("leon_isa.ac");

    icache.bindTo( ul2cache );  //Memory hierarchy
    dcache.bindTo( ul2cache );  //construction
  };
};
```

Figure 7. Memory hierarchy declaration for the Leon (SparcV8) architecture in ArchC.

The cache hierarchy is defined by the connection between the memory modules, establishing the cache levels that compose the memory subsystem. In ArchC the cache hierarchy is described by the cache's method `bindTo`. This method takes as argument the cache device which is the next cache level in the hierarchy. In the example (Figure 7), the `icache` and `dcache` modules are supplied by the lower level `ul2cache`, as declares the two last lines of the description.

As occur with the processor description, the memory configuration declared in the `AC_ARCH` input file is preprocessed and its behavior is included in the SystemC simulator that is automatically generated.

Distinctively from most of the cache simulators, the behavioral description generated by ArchC is not just based on the address information references. Indeed, the storage devices models are constructed as a data structure composed

by data blocks, valid and dirty bits, tag field and comparators. In this way, it is actually possible to simulate the data interchanging among the various levels of caches as well as I/O memory-mapped modules.

The accesses to storage device models during the simulation are performed through `write` and `read` methods. These methods were implemented taking into account the behavior expected for a given cache configuration. Thus, the referenced address is decomposed in portions like Tag, Index and Offset to access the respective cache position.

At each reference, the corresponding memory module searches for the requested data, resulting in a hit or a miss access. If a hit occurs, the data reading or writing is normally performed. However, if a miss happens, the processor fetch unit is stalled and the next memory level is invoked, in order to supply the reference absence. When the reference is ready, the processor returns to its normal condition. This mechanism allows the simulation of a complete memory hierarchy, including multi-level caches, and the use of a large amount of different performance parameters.

4. Behavioral Simulator of the Architecture

As illustrates the Figure 8, a SystemC model of the architecture is generated when the `Instruction Set Architecture (AC_ISA)` and the `Architecture resources (AC_ARCH)` descriptions are processed by the ArchC pre-processor.

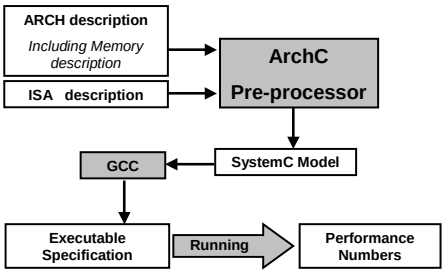

Figure 8. Behavioral simulator generation flow.

When compiled by GCC, this SystemC behavioral model produces an executable specification of the architecture. This specification may be a cycle-accurate simulator of the architecture and its execution generates performance metrics such as cache miss ratio, number of cycles to run the application, data transferred, etc. The numbers generated by running the simulator are used by the designer to evaluate the performance of the architecture for a given application.

5. A Case Study

5.1. The Application

As an illustrative example it is proposed a convolution algorithm, often used in image processing applications as high-pass or low-pass filters. Briefly, a convolution operator takes as input a digital image, in our case a 32×32 pixel grayscale image (1kB), and replaces each pixel value with the computing of the internal product with a small size kernel operator, 3×3 pixel for instance. The result, depending on the numerical value of the kernel's elements, will be a sharpened or a blurred version of the input image [12].

Due to the nature of the operation, massive and regular memory accesses are performed. This information can help the designer to choose adequately a set of architecture candidates to run the application.

Figure 9 shows the algorithm of the convolution. It describes a median filter which replaces each pixel value by a mean among its nine (3×3) neighbors, building as output a blurred version of the input image. Note that this example does not convolve the pixels on the image's edge, since they have not got a complete neighborhood. Compiled by the LECCS [5], the 62 instruction lines from the optimized object code are automatically loaded into the target architecture's memory by the ArchC's initialization resources.

```
#define height 32
#define width 32
#define k_size 3

int main () {
  int k_range = (k_size -1)/2;
  unsigned int k_elements = k_size * k_size;
  unsigned char imageIn [height][width];
  unsigned char imageOut [height][width];
  unsigned char kernel [k_size][k_size];
  int x, y, i, j;
  for(y=1; y < height-1; y++) {
     for(x=1; x < width-1; x++) {
        imageOut[x][y] = 0;
        for (i = - k_range; i <= k_range; i++) {
            for(j = - k_range; j <= k_range; j++ ) {
                imageOut [x][y] += imageIn [x+i][y+j] * kernel[i+1][j+1];
            }
        }
        imageOut[x][y] /= k_elements;
     }
  }
  return (0);
}
```

Figure 9. Application's source code: a convolution with a 3×3 median kernel for low-pass image filtering.

5.2. The Architecture

Figure 10 illustrates our architecture instance to run the application in this example. It consists of a Leon 32-bit processor connected to a memory hierarchy composed by in-

struction cache (I$), data cache (D$) and a second level of unified cache (L2$).

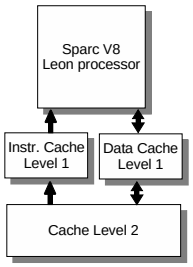

Figure 10. Architecture instance based on a Sparc V8.

The initial configuration is composed by direct-mapped I$ and D$ caches, both with 128 words (32-bit) organized in blocks with one word per line. In addition, a second level cache is present (4096 bytes), which is large enough to store all application's data references, including both input and output image, and also the kernel matrix. Starting from this configuration, the architecture's tuning process is performed.

It is a naive concept to believe that enlarging indefinitely the size of the first level cache (D$) will reduce completely the data cache miss ratio, increasing the application's performance. The point is that, as the data to be processed are originally stored in the lower level memory, unavoidable cache misses will occur at the first access for each reference. Also, for the application in matter, the data are regularly accessed, in such way that the next pixel to be convolved will utilize only six out of nine neighbors used at the previous pixel, always loading three new pixels to compose the operation. Finally, it is important to realize that the algorithm operates on a whole line (x-loop in the Figure 9) before passing to the next. So, the following y-iteration will make use of some previous data values, with the same horizontal position (x-iteration) to compose the pixel's neighborhood of the subsequent line.

Aiming to adjust the cache parameters like size, number of blocks per line and associativity, the application is executed on the architecture model using several different configurations. Taking into account the remarks above, a selective design space exploration was performed, aiming to minimize data cache misses while also minimizing chip area, by reducing the memory size.

5.3. Results

Table 1 presents performance results of sixteen different memory architecture configurations with varying cache sizes. A miss rate around 30% was expected, since the application basically reuses 6 (six) from the 9 (nine) input values required to calculate each output pixel, as mentioned above.

As it can be seen for direct-mapped single-word blocks (Group A), the read miss rate decreases while the number of cache lines is increased. Yet, the most sensitive falling is observed in the transition between 32-blocks and 64-blocks. The experiments from Group B show that around 5% of miss rate reduction can be achieved when 4-words blocks are used instead of 2-words. This improvement is due to the regular triple loading at each output pixel calculation. Two-way and four-way configurations were analyzed (Group C) and the results suggest that small caches do not take advantage of the line associativity with regard to miss rates.

The last experiment (D) shows the mixed effect of the two-way associativity with four-word lines in a 64-blocks cache. Its storage capability is equivalent to the 256-blocks cache from Group A, but its read miss rate is better than all other configurations presented.

	Data configuration		Read miss rate	Write miss rate
A	32-blocks	1-word	43.29%	12.23%
	64-blocks	1-word	23.31%	12.23%
	128-blocks	1-word	21.16%	12.20%
	256-blocks	1-word	19.30%	12.16%
	512-blocks	1-word	18.46%	12.08%
B	32-blocks	2-words	27.54%	12.23%
	32-blocks	4-words	22.01%	12.23%
	64-blocks	2-words	25.57%	12.23%
	64-blocks	4-words	18.92%	12.23%
	128-blocks	2-words	23.92%	12.23%
	128-blocks	4-words	17.61%	12.23%
C	128-blocks	2-way	19.94%	15.30%
	128-blocks	4-way	36.88%	23.35%
	256-blocks	2-way	18.84%	15.49%
	256-blocks	4-way	18.79%	18.75%
D	64-blocks	4-words, 2-way	17.54%	17.39%

Table 1. Data miss rates for different memory configurations

To facilitate the performance analysis, the design space explored can be graphically represented, as shows the Figure 11. The figure illustrates the relationship between read miss rate versus data cache size for the various configurations proposed, showing that for the same cache size (Bytes), it is possible to obtain different results.

6. Further Works

The next step of the our research project is to couple the behavioral description generated by ArchC with bus specifications, such as the AMBA AHB and APB [2], described in SystemC, in order to interface peripheral components to the processor core. This coupling will be made through the

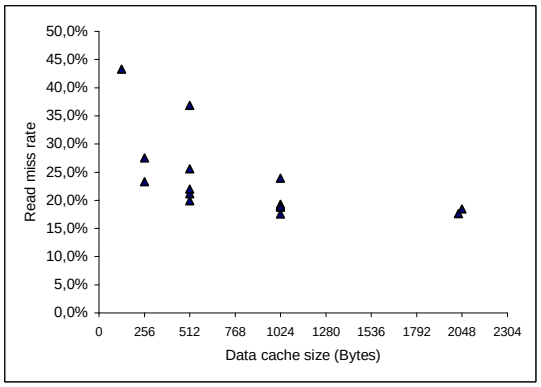

Figure 11. Design Space Exploration.

cache sub-system, by connecting the lower level cache to the bus.In this way, the memory hierarchy model may be expanded to more complex configurations, by adding PROM and SRAM memories, or even I/O units and storage devices, connected across the bus. This extension will enable the simulation of the entire system for performance evaluation, including input and output devices, as well as IP and others sub-systems, taking into account all the running time spent during the data moving among the system's modules.

7. Conclusion

We summarize this paper, enhancing the contributions achieved through the use of ArchC language to describe complete programmable systems involving processor and memory subsystems. It is worthy it to enhance the easy to model and the fast simulation allowed by ArchC, showing its suitability for architecture platform tuning, by enabling the development of an entire system, including its software and hardware components. As a SystemC based language, ArchC permits the addition of elements that do not compose the processor but are present on chip, like buses and peripherals. The generated SystemC model may be combined with additional descriptions, making possible the covalidation of the whole system.

Timing features like latency combined with the available cycle accurate information is being added to the memory models, for a more detailed penalty measurement at each memory access. Actually, beyond the results already reached up to now, it is expected that many others research works can arise soon, based on the ArchC skills to model and simulate architectures for SoC designs.

8. Acknowledgements

We would like to thank Capes and FAPESP (00/14376-2) for supporting this cooperation work, allowing the interchanging of scientific production. We really appreciate this initiative and gratefully acknowledge their contribution.

References

[1] G. Ammons, T. Ball, M. Hill, B. Falsafi, S. Huss-Lederman, J. Larus, A. Lebeck, M. Litzkow, S. Mukherjee, S. Reinhardt, M. Talluri, and D. Wood. Wisconsin architectural research tool set. *Computer Architecture News*, Vol. 21(4), 1993.

[2] ARM. *AMBA - Advanced Microcontroller Bus Architecture Specification*, arm ihi 0001d edition, April 1997.

[3] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128 – 137, May 1994.

[4] J. Davis, I. C. Hylands, J. J. E. A. Lee, J. L. X. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, P. Whitaker, and Y. Xiong. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, Department of Electrical Engineering and Computer Science, University of California, 2001.

[5] J. Gaisler. *The LEON/ERC32 GNU Cross-Compiler System*. Gaisler Research, version 1.1.5 edition, July 2002.

[6] J. Gaisler. *The LEON Processor User's Manual Version 1.0.10*. Gaisler Research, January 2003.

[7] T. Givardis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configuration in parameterized system-on-a-chip. *IEEE Trans. on VLSI Systems*, 2002.

[8] T. Givargis and F. Vahid. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computer Aided Design (TCAD)*, Vol. 21(11):1317 – 1327, November 2002.

[9] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Desing with SystemC*. Kluwer Academic Publishers, 2002.

[10] A. Halambi, P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test(DATE)*, March 1999.

[11] Henry Chang et al. *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, USA, 1999.

[12] A. K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, Englewood Cliffs, 1989.

[13] Richard P. Paul. *SPARC Architecture, Assembly Language Programming, and C*. Prentice Hall, 2nd edition, 2000.

[14] Sandro Rigo, Rodolfo Azevedo and Guido Araujo. The ArchC Architecture Description Language. Technical Report IC-03-15, Institute of Computing, University of Campinas, June 2003.

[15] SPARC International, Inc. *The SPARC Architecture Manual - Revision SAV080SI9308*.

[16] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system–on–chip design. In *Proc. APCHDL, Fukuoka, Japan, October 1999.*, October 1999.

[17] F. Vahid and T. Givargis. Platform tuning for embedded systems design. *IEEE Computer*, Vol. 34(3):112 – 114, March 2001.