The ArchC Architecture Description Language and Tools

Article *in* International Journal of Parallel Programming · October 2005
DOI: 10.1007/s10766-005-7301-0 · Source: DBLP

CITATIONS
113

READS
154

6 authors, including:

Rodolfo Azevedo
University of Campinas
98 PUBLICATIONS **596** CITATIONS
SEE PROFILE

Sandro Rigo
University of Campinas
53 PUBLICATIONS **390** CITATIONS
SEE PROFILE

Marcus Bartholomeu
University of Campinas
4 PUBLICATIONS **216** CITATIONS
SEE PROFILE

Guido Araujo
University of Campinas
151 PUBLICATIONS **1,404** CITATIONS
SEE PROFILE

International Journal of Parallel Programming, Vol. 33, No. 5, October 2005 (© 2005)
DOI: 10.1007/s10766-005-7301-0

The ArchC Architecture Description Language and Tools

Rodolfo Azevedo,[1,3] Sandro Rigo,[1]
Marcus Bartholomeu,[1] Guido Araujo,[1]
Cristiano Araujo,[2] and Edna Barros[2]

This paper presents an architecture description language (ADL) called ArchC, which is an open-source SystemC-based language that is specialized for processor architecture description. Its main goal is to provide enough information, at the right level of abstraction, in order to allow users to explore and verify new architectures, by automatically generating software tools like simulators and co-verification interfaces. ArchC's key features are a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against a reference (functional) description, memory hierarchy modeling capability, the possibility of integration with other SystemC IPs and the automatic generation of high-level SystemC simulators and assemblers. We have used ArchC to synthesize both functional and cycle-based simulators for the MIPS and Intel 8051 processors, as well as functional models of architectures like SPARC V8, TMS320C62x, XScale and PowerPC.

KEY WORDS: Architecture description language; SystemC; ISA simulator; compiled simulation.

1. INTRODUCTION

With the advent of System-on-Chip (SoC) designs, it is becoming possible to design an entire embedded system onto a single chip. As we approach

[1]Computer Systems Laboratory, Institute of Computing, University of Campinas, Cidade Universitaria Zeferino Vaz, P.O. Box 6176, Campinas-SP, Brazil. E-mail: {rodolfo, sandro, bartho, guido}@ic.unicamp.br
[2]Computer Science Institute, Federal University of Pernambuco, P.O. Box 7851, Recife-PE, Brazil. E-mail: {cca2, ensb}@cin.ufpe.br
[3]To whom correspondence should be addressed.

the availability of 100 Million gates, for the 90 nm VLSI technology, a new scenario has been drawn in which SoCs can be composed of a number of specialized processors interconnected by an elaborate *Network-on-a-Chip* (NoC). Moreover, as new specialized processor architectures are been proposed, it becomes evident the need for flexible simulation models that enable the designer to tailor processor ISA to the application. Such tools are commonly based on processor models written in some architecture description language (ADL).

In this paper, we introduce a new SystemC-based architecture description language (ADL) called ArchC. SystemC[1] is among a group of design languages and extensions being proposed to raise the abstraction level for hardware design and verification. SystemC is entirely based on C/C++ and the complete simulation library source code is freeware. SystemC is composed by a set of C++ class libraries, that extends the language to allow hardware and system-level modeling. Though SystemC supports a wide range of computation models and abstraction levels, it is not possible to extract information from a generic SystemC processor description in order to automatically generate tools to experiment and evaluate a new *Instruction Set Architecture* (ISA). ArchC is a simple language, specialized for processor architecture description, that follows C++/SystemC syntax style. Its main goal is to provide enough information, at the right level of abstraction, in order to allow users to explore and verify a new architecture by automatically generating software tools like assemblers, simulators, and co-verification interfaces. ArchC's key features are a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against an ArchC reference description, memory hierarchy modeling capability and the possibility of integration with other SystemC IPs. Functional and cycle-based simulators have been synthesized starting from ArchC descriptions for the MIPS (implementing the MIPS-I ISA), Intel 8051 and SPARC processors. Functional models for the Texas TMS320C62x, Intel XScale, Motorola ColdFire, PowerPC, Altera NIOS, OpenCores OR1k, and Hitachi SH-4 were also developed in ArchC, some of them are still in the verification phase.

Besides their application and well known suitability for designing and experimenting with new architectures in the industry, architecture description languages can be very useful for academic purposes, like teaching/researching computer architecture at the undergraduate and graduate levels. At the undergraduate level, models of well known architectures are appropriate to teach how a pipelined architecture works, including interlocking, hazard detection and register forwarding. When allowed by the ADL, a processor model can be plugged to different memory hierarchies in order to illustrate how the performance of a given application can vary,

depending on the choice of cache size, policy, associativity, etc. On the other hand, at the graduate level, researchers can use ADLs to model modern architectures and experiment with new ISA and structures, with all the flexibility demanded by research projects. ArchC fits very well such context, since the ArchC parser, simulator and assembler tools, along with several architecture models, are published on public domain and can be freely downloaded from.[2]

The remaining sections of this paper are organized as follows. Section 2 discusses some previous work on architecture description languages. Section 3 introduces the ArchC syntax and semantics. Section 4 presents the design support provided by ArchC. Section 5 describes ArchC tools and simulators, Section 6 presents some experimental results, and finally, Section 7 summarizes our conclusions and describes the future work.

2. RELATED WORK

The nML[3] ADL is based on the information typically available in the programmer's manual of a processor, which consists of a list of instructions and corresponding register transfers, binary encoding and assembly mnemonics. The abstraction level of nML is the instruction set, which is described through an attributed grammar. Hartoog *et al.*[4] presented a set of tools generated from a nML description of a processor and concluded that nML does not support multi-cycle instructions and straightforward extensions of nML can only support very simple pipelines, i.e., in nML it is not possible to produce cycle-accurate simulators for processors with complex execution schemes, like many DSPs. The nML implicit program counter is inconvenient in some circumstances, as when multi-word and multi-cycle instructions are used. These limitations are not present in ArchC.

The LISA[5] ADL was primarily designed for automatic retargeting of fast compiled simulators that are cycle and bit-accurate. When introduced, the main contribution of LISA was its operation-level description of the pipeline and sequencing model. Hoffmann *et al.*[6] used LISA to create a framework for hardware/software co-simulation, by connecting LISA models to SystemC hardware models. Nohl *et al.*[7] presented a new technique to mix compiled and interpretive simulation. Efficient support of complex instruction formats seems to require extensive coding in LISA. Companies like Axys Design and LISATek have been able to deliver proprietary LISA based processor models for a number of architectures, but no public release compiler and/or toolkit is available for this language.

EXPRESSION[8] is an ADL focused on architecture design space exploration for SoCs and automatic generation of a compiler/simulator

toolkit. EXPRESSION supports specification of memory subsystems, which seems to be the key feature of the language. Reshadi *et al.*[9] presented a technique called instruction-set compiled simulation (IS-CS), designed to improve the flexibility of compiled simulators without slowing down the performance to the interpretive simulation level.

The Instruction Set Description Language for Retargetability (ISDL)[10] was designed to be an ADL for compiler retargetability, specially focused on VLIW machines. Like nML, ISDL is a purely behavioral language also based on an attributed grammar. It is not clear if ISDL tools are capable of extracting the correct behavior for pipelined architectures with complex execution schemes. Pipeline flushes does not seem to be possible. Another strong constraint: ISDL is not able to model multi-cycle instructions of variable length.[11]

ArchC has its syntax totally based on C++ and SystemC. It was designed focused on SystemC users, i.e., aiming to make these users more comfortable with the language, since its syntax is very similar to the one they are familiar with. The user has just to learn a small set of keywords. ArchC relies on both structural and behavioral information of the architecture. ArchC has many features that distinguish it from other ADLs, like: non-proprietary SystemC compatibility, behavior modeling at several abstraction levels, behavior hierarchy, etc, Its key features are a storage-based coverification mechanism that automatically checks the consistency of refined ArchC model against ac ArchC functional description, memory hierarchy modeling capability, the possibility of integration with other SystemC IPs. To the best of our knowledge, it is currently the only ADL totally committed to the SystemC effort, and thus it is published on public domain so that all source code and tools presented in this paper can be freely downloaded. As shown later in this paper, ArchC models can produce very fast simulators.

3. ARCHC SYNTAX AND SEMANTICS

An architecture description in ArchC is divided in two parts: the *Architecture Resources* (`AC_ARCH`) description and the *Instruction Set Architecture* (`AC_ISA`) description. In the `AC_ARCH` description, the designer provides ArchC with information about storage devices, pipeline structure, memory hierarchy and all the processor resource information available in the ISA manual. In the `AC_ISA` description, the designer provides ArchC with details about each instruction, like: (a) format, size and assembly language syntax; (b) opcode information required to decode it; and (c) instruction behavior. Based on these two descriptions, ArchC automatically generates a simulator and an assembler for the architecture.

```
AC_ARCH(mips){
  ac_mem    MEM:5M;
  ac_cache  l2cache("dm", 256, 32,  "lru", "wb", "wal");
  ac_icache icache("dm", 16, 4, "wt", "war");
  ac_dcache dcache("2w", 32, 4,  "lru", "wb", "wal");

  ac_regbank RB:34;
  ac_pipe    pipe = {IF, ID, EX, MEM, WB};
  ac_wordsize 32;

  ac_format IF_ID_Fmt = "%npc:32";
  ac_format ID_EX_Fmt = "%rd:5 %rs:32 %rt:32 %imm:32";

  ac_reg<IF_ID_Fmt> IF_ID;
  ac_reg<ID_EX_Fmt> ID_EX;
          ...
  ARCH_CTOR(mips){
    set_endian("big");

    icache.bindsTo( l2cache );
    dcache.bindsTo( l2cache );
    l2cache.bindsTo( MEM );
  }
};
```

Fig. 1. MIPS AC_ARCH resource declaration.

3.1. Architecture Resources (AC_ARCH)

Architecture resources are described in the AC_ARCH description. ArchC provides a set of data types that can be used to declare registers (ac_reg), register banks (ac_regbank), memories (ac_mem), caches (ac_cache), pipeline and pipeline stages (ac_pipe), and other (micro) architecture modules. We illustrate the basic resource data-types available in ArchC using a fragment of the MIPS, AC_ARCH file, shown in Fig. 1.

The MIPS processor has a five stage pipeline, 32 general purpose registers plus two special registers used for multiplication and division. In the MIPS example of Fig. 1, the keyword ac_regbank is used to define the processor register file.

Caches are declared using the ac_cache keyword with the following attributes: block placement strategy, number of words in each block, number of blocks in the cache, set associativity, replacement policy and the

writing scheme. Figure 1 shows the declarations of `icache`, `dcache` and `l2cache`, 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 16 lines of 4 words each and write-through scheme, named `icache`, will be instantiated. The declarations also show a 2-way cache, with four words per line, replaced by a LRU policy (`dcache`) and a second level cache of 8192 words (`l2cache`).

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 defined by the method `bindsTo`. This method takes as argument the cache device which is the next cache level in the hierarchy. In the example (Fig. 1), the `icache` and `dcache` modules are supplied by the higher level `l2cache`, which in turn is supplied by the main memory, as declared in the last three lines of the description in Fig. 1.

Microarchitecture designers frequently need to access instruction fields or control signals stored in registers. In order to enable that, ArchC allows the designer to declare register formats, exactly as he/she does for instructions (Section 3.2). In ArchC, a format is declared through the `ac_format` keyword. The designer must provide a name and a string, representing the format subdivision into fields. Take a look at the `ID_EX_Fmt` format declaration in the example. The declaration of a field starts by a % character, followed by an identifier that becomes the field's name. The number after the colon indicates the size of the field. So, the string "%rs:5" declares a 5-bit field named `rs`. The designer assigns a format to a register by using the keyword `ac_reg` and a syntax similar to C++ *templates*. In the example, format `ID_EX_Fmt` is associated to register `ID_EX`. This allows the designer to assign to (and read from) each register field individually, when describing instruction behaviors. The behavior description of the add instruction in Fig. 3, uses `ID_EX.rt` to read the value stored in the `rt` field of the `ID_EX` pipeline register.

Pipelines are created through the keyword `ac_pipe` and are composed by a name and a list of pipeline stages. Pipeline stages listed in an `ac_pipe` declaration are assumed to execute instructions in the same order that they were declared. Stages also carry methods that allow pipeline flush and stall operations.

3.2. Instruction Set Architecture (`AC_ISA`)

The `AC_ISA` description provides ArchC with all information it needs to automatically construct a decoder and the skeleton of a SystemC (C++)

```
AC_ISA(mips){

 ac_format Type_R="%op1:6 %rs:5 %rt:5 %rd:5 0x00:5 %op2:6";

 ac_format Type_I="%op1:6 %rs:5 %rt:5 %imm:16";

 ac_instr<Type_R> add;
 ac_instr<Type_I> load;

 ISA_CTOR(mips){
   add.set_asm("add %reg,%reg,%reg," rd,rs,rt);
   add.set_decoder(op1=0x00, op2=0x20);

   load.set_asm("lw %reg,%imm(%reg)",rt,imm,rs);
   load.set_decoder(op1=0x23);

 };
};
```

Fig. 2. MIPS ISA Description.

source file where the designer inserts the behavior of each instruction in the ISA, as shown in Fig. 2 from our MIPS description.

The MIPS architecture uses the so called `R` format (or type) for arithmetic instructions. The example in Fig. 2 starts with the declaration of this format (`Type_R`), by using the `ac_format` keyword, similarly as it was done before (Section 3.1) to assign a format to a register. Still following the example in Fig. 2, notice that instructions are declared using the keyword `ac_instr`. The designer must provide a name and assign a previously declared format to any new instruction. The example shows the declaration of the `add` and `load` instructions.

The `ISA_CTOR` section of an `AC_ISA` declaration is used to construct instructions from their pre-defined formats. Instructions carry methods `set_asm` and `set_decoder` that are, respectively, used to define the instruction assembly syntax and the contents of its opcode fields. The former information can be used to synthesize the processor assembler and the latter a decoder for the simulator.

3.2.1. Modeling Instruction Behavior in ArchC

Figure 3 shows how the designer describes the behavior of each instruction in ArchC. This example is a fragment extracted from a description of the MIPS `add` instruction. The body of an `ac_behavior`

```
void ac_behavior( add, stage ){
   switch(stage)
    {
    case IF: ...
    case ID: ...
      break;
    case EX:
      EX_MEM.alu_result = ID_EX.rs + ID_EX.rt;
             ...
      break;
    case MEM:
      MEM_WB.alu_result = EX_MEM.alu_result;
      MEM_WB.rd = EX_MEM.rd;
             ...
      break;
    case WB:
      RF.write(MEM_WB.rd, MEM_WB.alu_result);
    default:
      break;
    }
};
```

Fig. 3. Instruction `add` behavior description.

method is a piece of code that executes operations by accessing declared storage devices and instruction fields. In general, if the designer declares a pipeline, he/she has to provide the behavior of each instruction for each pipeline stage. For example, during the execution of stage `EX` the designer uses fields `ID_EX.rs` and `ID_EX.rt` from pipeline register `ID_EX` to compute the result of instruction `add` into the `EX_MEM.alu_result` field of pipeline register `EX_MEM`.

If an instruction does not execute anything in a given pipeline stage, the `case` clause in the `switch` statement of the instruction behavior should be empty for that stage. Moreover, if instead of a pipeline, a processor has multi-cycle instructions, the `ac_behavior` method takes as argument the cycle number, also using a `switch` statement to select and execute the appropriate cycle, and `case` clauses to divide the instruction behavior into several cycles.

The main point about behavior description in ArchC is its flexibility. The `ac_behavior` method is written in pure SystemC (C++) code. Designers' own C++ functions can be called from `ac_behavior` methods, providing them with a lot of expression power for debugging and extension features. Moreover, there is no assumption that the designer is

```
void ac_behavior( add ){
  RB[rd]=RB[rs]+RB[rt];
}

void ac_behavior( load ){
  RB[rt] = DM.read(RB[rs]+imm);
}
```

Fig. 4. Functional Behavior Description in ArchC.

writing code at any determined abstraction level. For example, one could have written a functional model of the MIPS without a pipeline if cycle-accuracy was not an important feature at the early stages of the design. In that case, the instruction behavior could be as simple as the piece of code in Fig. 4, i.e., just a sequence of C++ statements to execute the given MIPS instruction behavior, resulting in a simulator that runs at one instruction per cycle. This ability to express behaviors in several levels of abstraction is another strong feature in ArchC.

Often, there are many instructions in a particular architecture that execute exactly the same task as part of their behavior. In order to factor out this behavior, writing it once and using it for all instructions of the same type, ArchC provides the designer with the possibility of overloading the `ac_behavior` method, so that it can take an instruction format as argument. In the MIPS processor example, all instructions that are declared with the `Type_R` format associated to it execute a couple of tests (as shown in Figure 5) *before* running its own behavior, so that they can do register forwarding for both instruction operands, `rs` and `rt`. The code in Fig. 5 happens to be as simple and clear as it is presented in the Hennessy and Patterson's classical architecture book.[12] The same strategy can be used for coding data hazard detection. There is still another situation, where the designer wants all instructions in the architecture to execute a piece of code before running its own behavior method. This is also possible in ArchC by describing a generic instruction behavior, which is a behavior method that belongs to all instructions. The designer should follow the same style used above and pass the keyword `instruction` as the argument to the behavior method. This approach allows a behavior hierarchy, where generic instruction behavior is executed first, followed by the format behavior, and finally the specific instruction behavior. The same sequence is performed by each pipeline stage in the case of a pipelined architecture. Notice that, for pipelined or multi-cycle instructions, both format and generic instruction behaviors can be written in a cycle-accurate way by means of a C++ switch statement, just like it is done for

```
void ac_behavior( Type_R ){
  switch( stage ) {
                    ...
  case _EX:
    /* Checking forwarding for the rs register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rs) )
           operand1 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rs) &&
          (EX_MEM.rdest == ID_EX.rs) )
           operand1 = MEM_WB.wbdata.read();
    else
      operand1 = ID_EX.data1.read();

    /* Checking forwarding for the rt register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rt) )
           operand2 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rt) &&
          (EX_MEM.rdest == ID_EX.rt) )
           operand2 = MEM_WB.wbdata.read();
    else
      operand2 = ID_EX.data2.read();
    break;
                ...
  }
}
```

Fig. 5. `Type_R` format behavior description.

instructions. By using ArchC's behavior hierarchy, we were able to factor out many operations that would have been repeated into several instruction behaviors, thus saving a great amount of redundant code in our models. A similar effect would be possible in an attributed grammar based ADL, but never with the same flexibility, cycle-accuracy and expression power as in pure C++ code.

4. ARCHC DESIGN SUPPORT

The ArchC main toolset is described in Section 5. Besides that it also comes with a subset of tools for design support. There are implemented tools for architecture refinement, application development and operating system (OS) call emulation, Architecture refinement is given by a *Storage-based Co-verification.* This feature helps the designer to compare one version of the architecture to a golden model. Application refinement support is provided by the *GDB Support for Simulators.* It is a debug functionality for the target application. Finally, the OS call emulation allows simulators to emulate file-system and other Operating System tasks. The tools for design support are described in detail below.

4.1. Storage-Based Co-Verification

Due to the current growth on the complexity and reduced time to market in embedded system designs, verification tools are gaining more and more importance in the whole process. The natural flow of a project is to start with a model at a higher level of abstraction and gradually refine it toward synthesis. As we see it, a design of an architecture model using ArchC should start by the elaboration of a description at the functional level. After validating the whole ISA using this high-level model, the designer can refine it, including pipeline structure, or making it a multi-cycle architecture model, in order to get a cycle-based description to experiment with. By describing an architecture in ArchC, the designer automatically gets a SystemC behavioral simulator of the processor at each of these phases.

Since this is a manual process, it is clearly error prone, and thus a tool for checking the refined behaviors against a reference model is very useful for verification purpose. ArchC provides a co-simulation tool allowing the designer to couple these two different models of the architecture. Both models run the same application and the ArchC verifier checks if they are consistent. Figure 6 illustrates the ArchC co-verification methodology, where usually an ArchC functional model is used as the reference model and the refined model is the *Device Under Verification* (DUV), assuming that it contains all the storage elements the designer wants to verify. Here, the refined model may be generated from a refined ArchC description. The ArchC verification approach is a transaction-based verification methodology that we call *Storage-Based Co-Verification*. It is based on tracking down every update to the storage devices of both models, marking them with time-tamps to show when they have happened. By comparing the sequence of transactions generated throughout

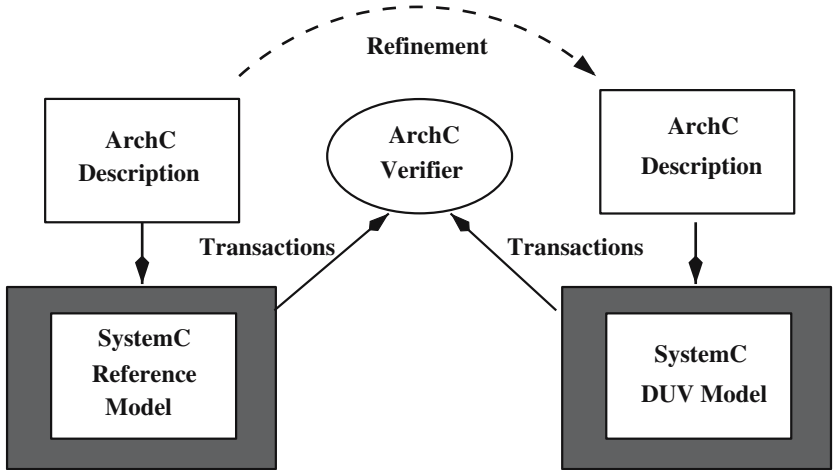

Fig. 6. ArchC Co-verification Methodology.

the execution, the ArchC verifier can tell if both models are consistent. Of course, this approach does not capture all possible design bugs, but it catches most of those that are expressed through storage elements.

When co-simulating two ArchC models, the designer does not have to make any change in the descriptions, it is only necessary to run the ArchC Simulator Generator (Section 5.1), passing the command-line option to generate a simulator prepared to run in the co-verification mode. When the designer simulates the refined model, coupled with a correct ArchC reference model, ArchC's co-verification mechanism issues an error message and saves a transaction log every time the values matched are inconsistent. Using this information, the designer rapidly identifies which instruction was executing at that time, and which values were stored into register banks, memories and pipeline registers. The whole process of identifying and correcting this kind of error is thus accelerated.

It is important to mention that both models run in parallel along with the `ac_verifier` tool, using three different Linux processes that communicate through IPC (Interprocess communication) mechanisms provided by the OS. We designed our co-verification algorithm to work in two situations. First, both the behavioral and the refined models are timing-accurate and the designer wants to assure that updates are executed at the same simulation (cycle) time on both models. Second, the ArchC reference model is not cycle-accurate and the designer just wants to verify the consistency of the update sequences generated by the models.

4.2. GDB Support for Simulators

Simulators built with ArchC can use the GDB protocol very easily. Just implement a few processor dependent methods for the interface and the simulator will be able to respond to GDB, allowing users to debug software inside the simulator. This is a new functionality and is implemented for simulators generated for *functional* models generated by `acsim` only. The SPARC-V8 and MIPS-I models available at the ArchC site[2] already count with the additional information to activate this functionality.

The developer should implement methods to interface the architecture and GDB, mapping registers and memory to the desired GDB format. These methods, explained in detail below, should be implemented in a file named after your processor: **PROC_gdb_funcs.cpp**, given that your processor is implemented in **PROC.ac**. Runnig `acsim` with `-gdb` will make your simulator class inherit from `AC_GDB_Interface` and use the functions defined in **PROC_gdb_funcs.cpp**.

When built, the simulator will handle the `-gdb` option, waiting for the GDB to connect at port 5000 (port defined at compile time, changeable with `-DPORT_NUM=<YOUR_PORT>`, or defined at run time, using `-gdb=<YOUR PORT>`).

4.2.1. Register Support

The designer must implement `nRegs()`, `reg_read ()` and `reg_write()` from `AC_GDB_Interface`, so the simulator can send the read and write register packets to GDB. You must check GDB documentation to learn the order the registers should be provided, and map them in `reg_read()` and `reg_write()`. Then, define the number of registers GDB expects to receive/send by using `nRegs()`. The order is defined by the `REGISTER_RAW_SIZE` and `REGISTER_NAME` macros. Please read the "`info gdb`", section "`Remote Protocol`", nodes "`Packets`" and "`Register Packet Format`" for more information.

Just to illustrate, if you have one bank with general purpose registers (`RB_GP`), one with floating point registers (`RB_FP`) and one with status register (`RB_S`). If GDB expects the packets in the order: 32 general purpose registers, 32 floating point registers and 8 status register and PC, your functions for processor PROC should be like those in Fig. 7.

4.2.2. Memory Support

The designer must implement `mem_read()` and `mem_write()` from `AC_GDB_Interface` to inform how to read and write memory regions.

```
int PROC::nRegs( void ) {
  return 73;
}

ac_word PROC::reg_read( int reg ) {
  // General Purpose
  if ( ( reg >= 0 ) && ( reg < 32 ) )
    return RB_GP.read( reg );

  // Floating Point
  else if ( ( reg >= 32 ) && ( reg < 64 ) )
    return RB_FP.read( reg - 32 );

  // Status
  else if ( ( reg >= 64 ) && ( reg < 72 ) )
    return RB_SR.read( reg - 64 );

  // Program Counter
  else if ( reg == 72 )
    return ac_resources::ac_pc;

  return 0; // unmaped register? return 0
}

void PROC::reg_write( int reg, ac_word value ) {
  // General Purpose
  if ( ( reg >= 0 ) && ( reg < 32 ) )
    RB_GP.write( reg, value );

  // Floating Point
  else if ( ( reg >= 32 ) && ( reg < 64 ) )
    RB_FP.write( reg - 32, value );

  // Status
  else if ( ( reg >= 64 ) && ( reg < 72 ) )
    RB_SR.write( reg - 64, value );

  // Program Counter
  else if ( reg == 72 )
    ac_resources::ac_pc = value;
}
```

Fig. 7. Register manipulation routines for GDB support.

```
unsigned char
PROC::mem_read( unsigned int address ) {
  return ac_resources::IM->read_byte( address );
}

void PROC::mem_write( unsigned int address,
                              unsigned char byte) {
  ac_resources::IM->write_byte( address, byte );
}
```

Fig. 8. Memory manipulation routines for GDB support.

Example: If you use just one memory bank (no separated data and instruction memory) for the processor `PROC`, it's easy as shown in Fig. 8.

4.3. OS Call Emulation

Programming in a high level language like C/C++ is mandatory for the current embedded software development flow. The majority of applications are designed using C/C++ together with some hand-tuned assembly code to improve performance of critical inner-loops. Any non-trivial C/C++ program uses the standard C library (`stdlib`), which provides the most essential routines for I/O, dynamic memory allocation and other utilities.

ArchC generates simulators capable of being instrumented with an OS call emulation mechanism, which enables ArchC models to simulate applications containing I/O operations. We have grouped the system call routines in a retargetable library written in the C language, so only a recompilation is needed to port it to any other target. The designer of a new architecture is able to tell from which storage elements (memory, register bank, etc.) the arguments to the system call will come from. This is done by writing interface functions that provide the required information to the ADL compiler. Typical examples of information that is required by most of the operating system calls are: how to get the first three arguments provided by a function call and how to save the return value as an integer or pointer. Functions in this group are normally very small, with less then five lines. The information required to implement them is taken from the processor Application Binary Interface manual. This feature allows ArchC to simulate programs extracted from real-world benchmarks, like Mediabench and Mibench, running them with realistic data samples.

5. ARCHC TOOLS

ArchC was idealized to facilitate description of novel architectures and automatically generate a SystemC model to simulate this new machine. The *(Interpreted) Simulator Generator* was the first tool created within the ArchC Project. Nowadays other tools were designed to use the same description. The *Compiled Simulator Generator* generates a faster simulator by using static scheduling. The *Assembler Generator* generates a port of GNU Assembler for the target architecture. Finally, the *Multiprocessor Simulator* put together some generated compiled simulators for heterogeneous architectures along with a communication mechanism.

5.1. The ArchC Simulator Generator and Simulators

We called as `ArchC Simulator Generator (acsim)` the tool that takes an ArchC description, composed by an instruction set architecture description (`AC_ISA`) and an architecture resource description (`AC_ARCH`), and generates a behavioral model of the architecture. `Acsim` uses two other tools that are not visible to the user: the Archc Pre-Processor (`acpp`), which is composed by a lexical analyzer and a parser for the language, and a decoder generator. The parser extracts information from the description files and stores it into data structures that are used by `acsim` to create all C++ classes and/or SystemC modules necessary to build the architecture simulator. The decoder generated by ArchC is capable of handling ISAs from simple RISC machines to multi-word variable length instructions, like in many DSPs.

ArchC can generate simulators using the interpreted technique. Interpreted simulators execute instruction decoding, schedule and behavior dynamically. Since the decoding process is too costly in terms of simulation performance, these interpreted simulators may use a cache to decoded instructions in order to speed-up simulation. Similar techniques are applied in some well known ISA simulators.[13] This technique can be disabled by command-line options passed to `acsim`, in order to enable the execution of self-modifying code.

ArchC SystemC simulators may be used for memory hierarchy exploration[14,15] and have even been successfully connected to other SystemC IPs to compose more complex digital systems. Depending on command-line options that can be passed to `acsim`, the generated simulator can be instrumented with several features to help on architecture exploration, like simulation statistics collection, trace generation, co-verification interface, etc.

5.2. Fast Static Compiled Simulation

ArchC also has a compiled simulator generator tool named `accsim`. It uses a static approach so-called *Fast Static Compiled Simulation* (FSCS) that will be explained in this section. We will also compare the FSCS technique with similar ones presented in the literature.

Figure 9 presents our workflow. The application is compiled using a cross compiler (`gcc` in our experiments) configured to generate a binary file for the target machine. This file, along with the architecture description, are the inputs to the static simulator generator `accsim`. The program is decoded one instruction at a time into a memory structure. At this point, the simulator generator can decide on some alternatives to build the simulator, using the optimizations described later in this paper if there is enough information to do it. We now describe the basic simulator that is generated, noting that just this base simulator, without any other optimization, is faster than comparable simulators in the literature.

Each instruction of the decoded application corresponds to the respective behavior function call in the generated simulator. These functions must be accessed randomly, as any of them can be a potential target of a jump instruction. We chose to label these functions using the C language `switch` statement, rather then goto labels, for better code structure and readability. Each one of them receives a `case` label. The switch is based on the Program Counter.

For any non-trivial program, this switch would be big enough to have a considerable impact on simulator compilation time, so we used one more switch level to select the region of code to be executed. A group of 512 instructions forms a region with their own switch. Each region is contained in its own function and the simulator returns from a region function only if the application jumps to an address outside the corresponding region. Figure 10 shows one region function in C language automatically generated from the SPARC V8 description. The master simulation loop selects the region from where the next instruction to be executed belongs to. This is presented in Fig. 11.

The simulator generated by this approach is static, based on the hypothesis that the application does not suffer any change in the binary

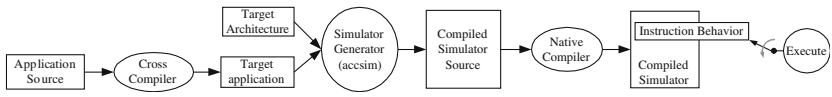

Fig. 9. Fast Static Compiled Simulation Flow.

```
void Region3() {

  while (1) {
    switch(ac_pc) {

    ...

    case 0x1954: // be PC-348
      ac_behavior_instruction(4);
      ac_behavior_Type_F2B(0, 0, 1, 2, -348);
      ac_behavior_be(0, 0, 1, 2, -348);
      ac_instr_counter++;
      break;

    case 0x1958: // andcc %28, 4, 0
      ac_behavior_instruction(4);
      ac_behavior_Type_F3B(2, 0, 17, 28, 1, 4);
      ac_behavior_andcc_imm(2, 0, 17, 28, 1, 4);
      ac_instr_counter++;
      break;

    ...

    default:
      if ((ac_pc >= 4096) && (ac_pc < 6144)) {
        AC_ERROR(... non-decoded memory location ...);
        ac_stop(EXIT_FAILURE);
      }
      return;
    }
  }
}
```

Fig. 10. One of the region functions automatically generated for SPARC V8 architecture.

code during the simulation. This hypothesis is used to generate a very fast simulator that will have the application hard-coded into it.

The function call, that corresponds to the behavior of an instruction in the application, receives the decoded instruction fields as arguments.

```
void Execute(int argc, char *argv[]) {

  model_syscall.set_prog_args(argc, argv);

  while (!ac_stop_flag) {
    switch(ac_pc >> 11) {

    case 0: Region0(); break;

    case 1: Region1(); break;

    ...

    case 14: Region14(); break;

    default:
      AC_ERROR(... print error and debug info ...);
      ac_stop(EXIT_FAILURE);
      break;
    }
  }
}
```

Fig. 11. The main simulation routine.

Also, all behavior functions in the ISA are marked as *inline* functions. This allows the optimizing compiler to generate faster code for the simulator.

The just described simulation technique is in the core of our base compiled simulator, and can be compared to similar retargetable simulators present in the literature. The EXPRESSION group, from the University of California, in a recent work,[9] claims to have the fastest retargetable simulator at the instruction level. In their work, the *Instruction Set Compiled Simulation* (IS-CS), the simulator uses dynamic scheduling, so the application can be modified during the simulation. As stated in their work, they archive up to 9% performance improvement by disabling the instruction modification detection. We can compare our work with this last case as their simulator becomes static as ours.

In Ref. 16 we could find performance information for the SPARC architecture using the IS-CS approach. Figure 12 compares the performance numbers from both proposals. Our data was taken using a compatible machine, an Athlon running at 1 GHz and 1 GB RAM versus a

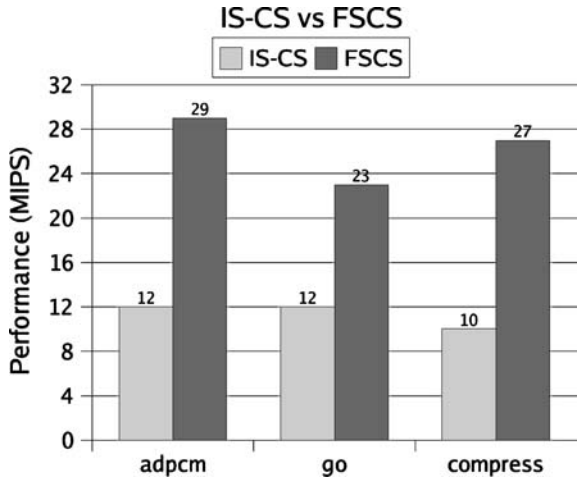

Fig. 12. Comparison between IS-CS and FSCS.

Pentium 3 at 1 GHz with 512 MB RAM. The difference in memory configuration is not important as the biggest program, 129.compress, needs only 50 MB of memory to run (and the simulator itself has less then 2 MB overhead). Our base compiled simulator performs from 90% to 170% better than IS-CS technique. The next sections contain optimizations for our approach that increase even more this speedup difference.

The LISA project, started in Aarchen University, but now a commercial product, states on[7] that their *Just-In-Time Cache Compiled Simulation* technique achieves 95% of the compiled simulation performance. So we can also compare our approach to theirs. Their site[4] informs that their technique can perform up to 30 MIPS in an Athlon 2.5 GHz. Our results, presented later in this paper in Section 6.1 shows that our simulator can achieve a minimum of 45 MIPS for the MIPS architecture model on this platform.

The main challenge in ADL research is to have a model that is efficient in terms of both quality of description and performance of the simulator. To have a high quality description, the model must easily capture the architectural information in a natural, compact and manageable form for a wide range of architectures. On the other hand, to generate a high performance simulator and reduce the operations that the simulator must

[4]http://www.coware.com/products/lisatek_description.php

perform dynamically at run time, the model should provide as much static information as possible about the architecture and its instruction set.

In order to achieve a better performance for the simulator, we included additional constructs into the ArchC description language. They are optional but, if provided, allow the generation of a better compiled simulator. This work presents two levels of optimizations based on more detailed information about the architecture provided by the designer.

5.2.1. Optimization 1

The first optimization requires the identification of the instructions that can change the execution flow. By using this information, `accsim` creates a simulator that saves execution time by testing for control flow changes only after those instructions.

The delay slot feature of some architectures has to be treated carefully in instruction level simulation. The program flow may change not after the control instruction, but after the instruction in the delay slot. For this reason, the number of delay slots must also be provided in the ISA description.

Some architectures, like SPARC, can annulate the delay slot instruction. A simple solution was used to allow similar cases to work without any further architecture information. The test for changes in the control flow is done right after control-flow instructions *and* their delay slot instructions.

The additional information needed for the first optimization approach could be integrated easily into ArchC descriptions following the same syntax rules, as shown in Fig. 13 for the SPARC V8 instructions `call` and `be`. The new constructs are in bold face.

```
...
call.set_asm("call %disp30");
call.set_decoder(op=0x01);
call.jump();
call.delay(1);

be.set_asm("be %an, label");
be.set_decoder(op=0x00, cond=0x01, op2=0x02);
be.branch();
be.delay(1);
...
```

Fig. 13. Additional information for optimization 1 in ArchC description.

Three instruction properties were added to the description. Two of them, namely `jump` and `branch`, are used to declare control instruction to ArchC. These two keywords have the same meaning for now, but they were designed to represent unconditional or conditional control flow instructions, respectively, The third property, named delay, has one parameter to inform the number of delay slot instructions before the control flow changes.

The existing ArchC descriptions for SPARC V8 and MIPS I were incremented with the new information and the simulator was automatically built using this first optimization. This optimization produces a simulator similar to the one presented in Fig. 10, but with break statements only after control instructions and their delay slots, so program flow is evaluated less frequently.

5.2.2. Optimization 2

The second experiment was to include sufficient information into the architecture description to make the simulator capable of controlling all the execution flow of the application. This ambitious objective made it possible to remove the Program Counter manipulation from the instruction behaviors, allowing the simulator to take care of it. Much more information about the control instructions is needed though, but the excellent results are worth the price.

The information given for optimization 1—declaration of control flow instructions and how many delay slots they have—is also used here. The simulator still needs to know how control instructions calculate their target, the condition to branch to the target, the condition to execute the delay slots (so they can be annulled), and the additional behavior executed by the control instruction, if there is one (for example call instructions need to save the return address).

At this point, the keywords for the properties `jump` and `branch` are used to represent unconditional and conditional jumps, respectively. The expression for the target calculation was included into the ArchC description as an argument for the `jump` and `branch` properties.

We also need to capture the condition for conditional jumps, and thus it became a new property `cond`. The condition for the execution of delay slots became the second argument to the `delay` property. Finally, a new property, so called `behavior`, was added to permit the instruction to execute any other action besides the flow of control. This behavior substitutes the original one in the ISA description for the instruction. Figure 14 shows the added syntax for the same `call` and `be` SPARC V8 instructions used in the optimization 1 example.

```
...
call.set_asm("call %disp30");
call.set_decoder(op=0x01);
call.jump(ac_pc+(disp30<<2));
call.delay(1, true);
call.behavior(writeReg(15, ac_pc));

be.set_asm("be %an, label");
be.set_decoder(op=0x00, cond=0x01, op2=0x02);
be.branch(ac_pc+(disp22<<2));
be.cond(PSR_icc_z)
be.delay(1, PSR_icc_z || !an);
...
```

Fig. 14. Additional information for optimization 2 in ArchC description.

6. EXPERIMENTAL RESULTS

We evaluated the ArchC generated simulators performance for both interpreted simulators and optimized compiled simulators with the FSCS technique. Table I shows some performance measurements of our SPARC-V8 and MIPS-I models running programs extracted from the Mediabench[17] suite, including system calls emulation. Table II shows the performance measurements for the Mibench[18] suite, which has two workloads imposed by small and large input data samples, but only the large work-load is presented. We have included the number of running instructions for each application to show the large coverage of these tests. We can see that the LAME MP3 coder executes nearly 95 billion instructions in the MIPS architecture model. The output files provided by the simulators were checked against files generated by a real SPARC machine running the same benchmarks. The MIPS-I model reached 550.91 KIPS (thousands of instructions per second) in average for interpreted simulator and 60.21 MIPS (millions of instructions per second) for optimized FSCS, while the SPARC-V8 model reached 633.47 KIPS and 138.83 MIPS, respectively. The SPARC model is faster because this model does not need the ArchC internal delayed assignment control, which is used to simulate delay slots on the MIPS-I model. This functionality brings an additional burden for the MIPS simulator. The use of a cache associated to the decoder boosted the performance of our interpreted simulators in approximately 80% for our MIPS-I model and more than 200% for the SPARC-V8 model. The greater impact on the SPARC-V8 model performance is due to the higher complexity of its ISA, when compared with MIPS-I, demanding more processing power to decode it. These simulators ran on a P4 2.8 Ghz 1 GB

Table I. ArchC Functional Simulator Performance Evaluation for MediaBench

	MIPS-I			SPARC-V8		
Program	#Instr.	Interpreted (KIPS)	FSCS (MIPS)	#Instr.	Interpreted (KIPS)	FSCS (MIPS)
ADPCM Coder	7,491,407	542.91	74.91	7,256,633	636.22	139.44
ADPCM Decoder	5,902,038	553.76	73.78	6,261,169	652.16	145.13
JPEG Coder	16,776,932	568.70	67.11	14,288,757	606.85	129.90
JPEG Decoder	5,205,441	559.48	65.07	4,187,863	625.48	149.57
MPEG Coder	11,699,577,710	540.59	48.61	10,834,240,031	637.89	112.13
MPEG Decoder	3,858,003,816	569.94	50.62	3,451,838,509	640.80	79.17
PEGWIT Encode	31,167,432	546.36	52.10	30,823,606	616.18	151.95
PEGWIT Decode	17,495,609	565.64	51.47	16,865,081	625.39	153.32
PEGWIT Generate	12,611,316	551.89	66.38	12,790,067	642.19	159.88
GSM Encode	220,916,822	575.00	37.07	157,404,486	639.92	162.27
GSM Decode	64,216,961	570.25	60.02	88,286,413	653.43	149.64

Table II. ArchC Functional Simulator Performance Evaluation for MiBench

	MIPS-I			SPARC-V8		
Program	#Instr.	Interpreted (KIPS)	FSCS (MIPS)	#Instr.	Interpreted (KIPS)	FSCS (MIPS)
Basicmath	22,469,864,764	553.76	48.37	21,365,365,698	619.22	101.80
Bitcount	684,250,120	550.90	78.61	748,368,499	636.14	121.62
Qsort	991,774,388	561.43	49.51	792,301,299	630.78	94.25
Susan (smooth)	423,335,581	545.31	63.07	349,096,326	658.56	143.26
Susan (corners)	44,558,848	565.41	49.41	42,298,235	659.97	98.91
Susan (edges)	177,394,682	545.58	45.92	174,339,811	620.78	111.76
Dijkstra	285,280,151	549.00	71.51	244,328,854	636.40	149.79
Patricia	1,828,671,354	557.35	45.72	1,760,759,794	643.82	106.07
ADPCM Coder	688,959,463	533.88	70.67	678,637,897	649.54	126.98
ADPCM Decoder	538,659,721	530.21	75.71	584,732,032	646.23	141.37
FFT	15,244,218,349	560.88	44.69	12,936,715,454	642.34	103.80
FFT INV	14,750,402,897	561.03	44.36	12,559,257,170	626.07	103.29
CRC 32	615,051,948	547.36	75.34	587,712,330	630.98	167.98
JPEG Coder	118,600,038	535.97	65.65	93,002,355	630.04	132.82
JPEG Decoder	32,333,052	562.32	63.16	24,773,993	648.63	161.78
LAME	94,314,747,771	564.87	45.87	89,899,090,464	629.76	103.30
SHA	135,696,013	561.65	81.48	137,975,709	626.74	220.92
RIJNDAEL Encode	350,251,921	542.67	30.30	339,042,970	613.42	148.00
RIJNDAEL Decode	361,155,494	541.36	31.53	338,231,604	613.54	154.68

machine These results are from our functional model. Applications are compiled for the target architecture using gcc and loaded into the simulator. Our experiments also showed that, when running on co-verification mode, the simulators reach about 36% of their usual performance.

Up to this point, our group and collaborators have modeled the following architectures in ArchC: MIPS, SPARC-V8, Intel 8051, Intel XScale, Motorola ColdFire, PowerPC, Texas TMS320C62x, Altera NIOS, OpenCores OR1k, and Hitachi SH-4. For two of these architecture we have both cycle-accurate and functional models: MIPS and the Intel 8051 microcontroller. The development stage reached by each model can be checked at Ref. 2. The first two are well-known general purpose architectures, the third is one of the most used processor in embedded control applications, and the following are modern architectures largely used in embedded systems. The SPARC-V8 functional model has been used to develop and explore memory hierarchy simulation in ArchC,[14,15] by performing experiments to evaluate several cache and hierarchy configurations using multimedia applications. This was our first step toward extending ArchC modeling capabilities beyond processor architecture boundaries, enabling designers to simulate complete systems, like SoCs, based on ArchC descriptions of processors and memory hierarchies. Our partners are currently working on connecting ArchC generated models to other SystemC IP through buses, like OCP/IP. There is a first prototype which uses the SPARC-V8 functional model, simulating other SystemC IPs, like memories or application specific modules and communicating through an OCP/IP interface. By adding this communication feature, we intend to provide ArchC users with a set of tools that are suitable for supporting platform-based design.

We are also applying ArchC as a tool to support computer architecture education.[19] It has been applied in computer architecture courses at IC-UNICAMP during the last year. In one of theses courses, students were divided into several groups, and each one of those groups was assigned to a different project. The project was basically to develop a new functional model of a real-world architecture, most of them largely used in the industry as part of SoCs and embedded systems. This experience was very useful and tested the language's expression power, and gave the opportunity to students to contribute with suggestions and improvements to the ArchC tools.

6.1. Optimizations for Compiled Simulation

The optimizations for the compiled simulation require additional information from the model description. This information were included

in the SPARC V8 and MIPS I ArchC models in less than a day for each architecture by one designer. Remember that we require additional information only for the control instructions. The SPARC V8 ISA has 19 control instructions, while the MIPS I ISA has 12.

The performance numbers for the generated simulators execution were taken from a 2.8 GHz Pentium 4 host machine running Linux. The generated simulator code was compiled using gcc 2.96.

Figure 15 shows the benchmarks results for the SPARC V8 architecture. The best results, from optimization 2, reaches a speedup range from 147% (for the `adpcm encoder` application) to 244% (for `sha`) when comparing to the already fast FSCS base technique. The generated compiled

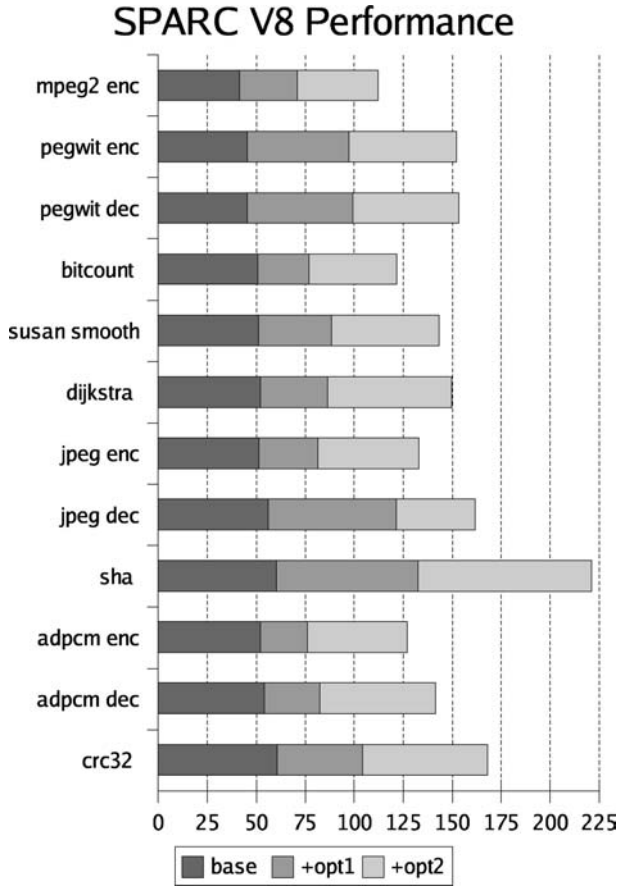

Fig. 15. SPARC V8 Performance.

simulator for the `sha` program presents the best performance, simulating 221 millions of instructions per second (MIPS). The average performance is 134 MIPS.

Figure 16 shows the results for the MIPS I architecture. The numbers are lower than those for the SPARC V8 architecture, because the MIPS I ISA implementation for the delay slots uses a feature from the ArchC description called *delayed assignment,* where an assignment to any variable (in this case, the program counter) can be delayed any number of instructions. When this feature is turned on, the simulator has to do additional control before each instruction completion, to update the assigned values

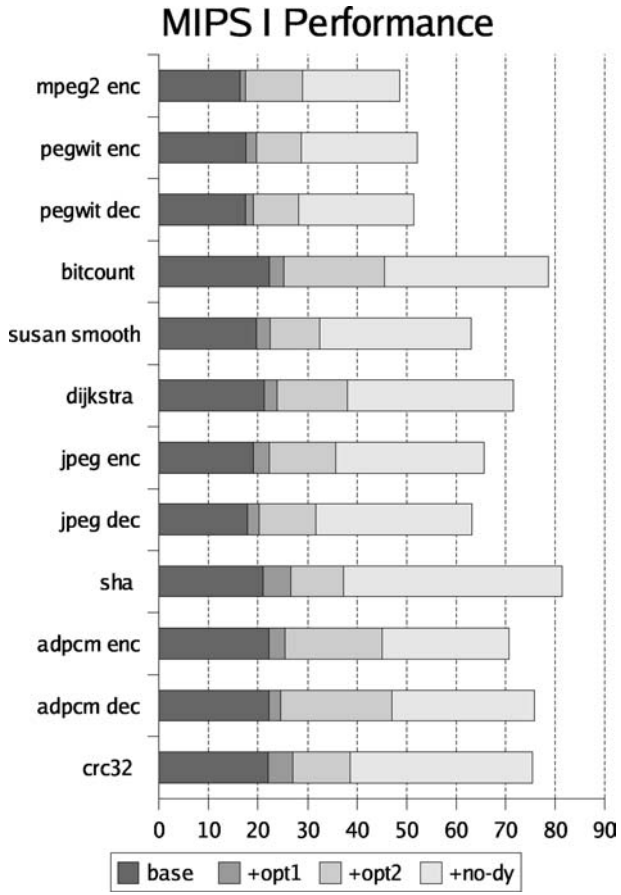

Fig. 16. MIPS I Performance.

in the correct cycle. The *delayed assignment* feature facilitates the architecture design, but has a noticeable impact in simulation speed. The SPARC V8 architecture implements delay slots in a different way, maintaining two registers for the program counter, one for the present instruction and the other for the next one, as indicated in the SPARC V8 ISA specification.

Note that optimization 2 can give a burst in performance for the MIPS I architecture. This optimization transfers the program flow control from the instructions behaviors to the simulation kernel, so there is no need to update the program counter register. The *delayed assignment* feature for the MIPS I model was turned on just for program counter manipulation and is not necessary anymore. Without the overhead caused by this mechanism, the compiler was allowed to perform more optimizations and the model could be simulated at least 55% faster when compared to the optimization 2 results with the feature turned on. This performance gain is represented in Fig. 16 with the label `+no-dy`.

For both architectures, the optimization 2 makes the generated simulator code simpler for the compiler. That is another good advantage, as the compilation time is reduced by 20% comparing to the base compiled simulator. The compilation time increases linearly with the target application size, taking 85 s to compile the simulator generated for an average-sized application (20,000 instructions).

We also ran experiments to compare the generated simulators performance to the native performance. This method resulted in numbers that are much more independent from the host machine speed. Table III shows the execution time in seconds obtained running the SPARC V8 simulator in a Pentium host. In Table IV we also present the timing for the same SPARC V8 simulator compiled and executed in a native SPARC machine. The difference in slow-down factors for the two host architectures can be attributed to some reasons: (a) SPARC is a RISC architecture, while Pentium is a CISC, a huge difference in the instruction sets: note the relative difference in native performance among the tree programs; (b) the Pentium class processors have more internal cache memory and thus the simulators perform better.

Table III. Native versus Simulation Execution Times for Intel Host

Application	Intel native	SPARC simulator	Slow-down factor
adpcm	0.90	12.69	14.10×
go	45.87	1322.44	28.83×
compress	121.73	2233.33	18.35×

Table IV. Native versus Simulation Execution Times for SPARC Host.

Application	SPARC native	SPARC simulator	Slow-down factor
adpcm	1.96	45.33	23.13×
go	112.03	3226.21	28.80×
compress	132.90	7216.13	54.30×

7. CONCLUSIONS AND FUTURE WORK

This paper presents ArchC, a young architecture description language. ArchC was created to provide designers with the possibility of using automatically generated SystemC executable models from the very early stages of the architecture design process, combined with the power of automatically generating tools for architecture exploration and verification. Designers can choose between interpreted and compiled simulators, they can also use ArchC models to simulate/evaluate more sophisticated memory hierarchies with several cache and memory levels. ArchC is already capable of generating assemblers for RISC-like architecture models.

We are currently working on improving and tuning our co-verification algorithm and interfaces. As stated above, models of real-world architectures, like TMS320C62x, IA32, Motorola ColdFire, and ARM are under development. Some of these models has CISC-style variable-length instructions, which can be described in ArchC as easily as RISC-like instructions. ArchC is able to describe complex architectures like VLIW and superscalar, as functional models. Cycle-accurate simulators capable of treating superscalar features, like out-of-order execution, are under development. Finally, we are working to add a new TLM SystemC interface to ArchC simulators, making it easier to connect them to other SystemC IPs.

ACKNOWLEDGMENTS

We would like to thank FAPESP (Grants 00/14376-2, 01/09424-1, and 2000/15083-9), CNPq (Grant 55.2117/2002-1) and CAPES (PROCAD, PROC NR. 0073/01-6) for the financial support to this work. We are also very grateful to all the students and teachers that are using ArchC, for education and/or research, whose contributions and feedback have been extremely valuable to the continuous improvement of the ArchC tools.

REFERENCES

1. OSCI, *SystemC Version 2.0 User's Guide* (2001).
2. http://www.archc.org, The ArchC Resource Center.

3. A. Fauth, J. Van Praete, and M. Freericks, Describing Instruction Set Processors using nML, In *Proc. European Design and Test Conf. Paris,* pp. 503–507 (March 1995), URL citeseer.nj.nec.com/fauth95describing.html.
4. M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign, In *Proc. Design Automation Conference,* pp. 303–306 (1997), URL citeseer.nj.nec.com/hartoog97generation.html.
5. V. Zivojnovic, S. Pees, and H. Meyr, LISA — Machine Description Language and Generic Machine Model for HW/SW Co-Design, *Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco* (1996), URL citeseer.nj.nec.com/zivojnovic96dsp.html.
6. A. Hoffmann, T. Kogel, and H. Meyr, A Framework for Fast Hardware-Software Cosimulation, *Proceedings of Design, Automation and Test in Europe Conference (DATE)* (March 2001).
7. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation, *Proceedings of Design and Automation Conference (DAC)* (June 2002).
8. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability, In *Proc. European Conference on Design, Automation and Test (DATE)* (March 1999).
9. M. Reshadi, P. Mishra, and N. Dutt, Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation, *Proceedings of Design and Automation Conference (DAC)* (2003).
10. G. Hadjiyiannis, S. Hanono, and S. Devadas, ISDL: An Instruction Set Description Language for Retargetability, *Design Automation Conference (DAC)*, pp. 299–302 (1997), URL citeseer.nj.nec.com/hadjiyiannis97isdl.html.
11. G. Hadjiyiannis and S. Devadas, Techniques for Accurate Performance Evaluation in Architecture Exploration, *IEEE Transactions on VLSI Systems* (2002), URL citeseer.nj.nec.com/hadjiyiannis02techniques.html.
12. J. Hennessy and D. Patterson, *Computer Organization & Design: The Hardware/Software Interface,* Morgan Kaufmann (1998).
13. B. Cmelik and D. Keppel, Shade: A Fast Instruction-Set Simulator for Execution Profiling, *ACM SIGMETRICS Performance Evaluation Review,* 22(1):128–137 (May 1994), URL citeseer.nj.nec.com/article/cmelik93shade.html.
14. P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araújo, Exploring Memory Hierarchy with ArchC, *Proc. of the 15th Symp. on Computer Architecture and High Performance Computing, São Paulo,* (*SBAC-PAD'03*) (November 2003).
15. P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araújo, Modeling and Simulating Memory Hierarchies in a Platform-Based Design Methodology, *Proc. of the Design, Automation and Test in Europe (DATE'04), Paris* (February 2004).
16. M. Reshadi, N. Bansal, P. Mishra, and N. Dutt, An Efficient Retargetable Framework for Instruction-Set Simulation, *Proceedings of the 2003 International Symposium on System Synthesis (ISSS-2003),* Newport Beach, California (October 2003).
17. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, Mediabench; A Tool fo Evaluating and Synthesizing Multimedia and Communications Systems, *Proc. of 30th Annual International Symposium on Microarchitecture* (December 1997).
18. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, *IEEE 4th Annual Workshop on Workload Characterization,* Austin, TX (December 2001).

19. S. Rigo, M. Juliato, R. Azevedo, G. Araujo and P. Centoducatte. Teaching Computer Architecture Using an Architecture Description Language, *Proceedings of the Workshop on Computer Architecture Education (WCAE), held in conjunction with the International Symposium on Computer Architecture (INCA), Munich* (June 2004).