Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language

Stefan Pees, Andreas Hoffmann, Heinrich Meyr
Integrated Signal Processing Systems, RWTH Aachen
pees[hoffmann,meyr]@ert.rwth-aachen.de

Abstract

This paper presents a methodology to retarget the technique of compiled simulation for Digital Signal Processors (DSPs) using the modeling language LISA. In the past, the principle of compiled simulation as means for speeding up simulators has only been implemented for specific DSP architectures. The new approach presented here discusses methods of integrating compiled simulation techniques to retargetable simulation tools. The principle and the implementation are discussed in this paper and results for the TI TMS320C6201 DSP are presented.

1 Introduction

Integrating complete systems consisting of hardware and software components on a single chip raises new challenges in the area of verification. Because target hardware is typically available only late in the design cycle, the complete system must be verified by means of *cycle-accurate* simulation. At the same time, *simulation speed* is critical for the verification of such systems and thus an important issue in simulator design [1, 2].

The principle of compiled simulation is to take advantage of a priori knowledge and move frequent operations from simulation run-time to compile-time with the goal of providing the highest possible simulation speed. In contrast to interpretive simulators, this approach requires a translation step to be performed before the simulation can be run. Signal processing algorithms typically consist of many loops and DSP code has a high locality. Therefore, the additional effort of translating the application code to a compiled simulation pays off. This is because frequent operations such as fetching, dispatching and decoding instruction words are performed only once at compile-time instead of every time the respective instruction is executed at run-time of the simulation.

Compiled simulation of programmable DSP architectures was introduced to speed up the instruction set simulation of programmable DSP architectures [3] and was extended to cycle-accurate models of pipelined processors [4]. So far, the approaches addressing the particular requirements of compiled simulation of DSPs are targeted to a specific processor architecture using a handwritten simulation compiler. However, the task of building a custom simulator for new architectures is extremely error-prone and tedious. It is a very lengthy process of matching the simulator to an abstract model of the processor architecture. These efforts can be reduced significantly by using a retargetable simulator which is generated from machine descriptions [5, 6]. This paper explores the general principles of compiled simulation that can be applied when using a language-based approach. Furthermore, an implementation based on the machine description language LISA [7, 8] is presented. From this description, efficient simulation tools are generated.

2 Related Work

Hardware description languages (HDLs) like VHDL or Verilog are widely used to model and simulate processors, but mainly with the goal of developing hardware. Using these models for instruction-level processor simulation has a number of disadvantages. They cover hardware implementation details which are not needed for performance evaluation and software verification. Moreover, the description of detailed hardware structures has a significant impact on simulation speed [2].

Many publications on machine description languages are focused on retargetable compilation for embedded processors. The approaches of Maril [9] as part of the Marion environment and a system for VLIW compilation [10] are both using reservation tables for code generation. Nevertheless, reservation tables cannot be used

to accurately describe pipeline operations like flushes or to model pipeline hazards.

Several publications address retargetable compilation and simulation. The language nML [11] was developed at TU Berlin [12, 13] and adopted in several projects [1, 14]. While retargetable assemblers and disassemblers can be generated for some DSP processors, it is not possible to produce cycle-accurate simulators for pipelined processor architectures. The main reason is the simple underlying instruction sequencer which does not support pipeline operations like e.g. flushes. Processors with more complex execution schemes like the Texas Instruments TMS320C6x [15] cannot be described, even at the instruction-set level, because of the numerous combinations of parallel and sequential instructions within a fetch packet. These restrictions also apply to the approach of ISDL [16] which is very similar to nML. However, cycle-accurate models of pipelined processor architectures require a pipeline-accurate behavioral description beyond pure semantics. The approach based on the language EXPRESSION [17] incorporates particular mechanisms for the description of memory hierarchies. However, no results are published that indicate the applicability for cycle-accurate simulation purposes.

The tool set of the SimpleScalar architecture provides five fast simulators with different accuracy levels [18]. But the retargetability of this tool set is restricted to derivatives of the MIPS architecture.

The language RADL [19] is derived from earlier work on LISA [7] and extended to support multiple pipelines. But no results are provided on realized simulators based on this language.

To summarize the review, none of the approaches above does support cycle-accurate simulation or fast processor simulators that are based on compiled techniques [4]. Our interest in supporting this technique and the issue of realizing cycle-accurate processor models motivated the introduction of the language LISA which is used in our approach [7, 8].

3 Compiled Simulation

The objective of compiled simulation is to reduce the simulation time. In general, efficient run-time reduction is achieved by accelerating frequent operations. Here, the technique for accelerating operations is to use a priori knowledge during the translation of target program code into simulation code for the host.

The principle of compiled simulation for DSPs corresponds to the ideas that are already successfully implemented in the simulation of synchronous VLSI circuits [20], constant propagation in high-level language compilers [21], and that are used for static multi-processor scheduling [22]. Such compiled simulators for DSPs have been realized for specific processor architectures [4]. Re-using the efforts for the implementation of the compiled techniques is extremely difficult since the techniques are implemented in the so-called *simulation compiler* which is highly architecture dependent.

The processing of the simulation compiler can be split into three major steps which are depicted in figure 1.

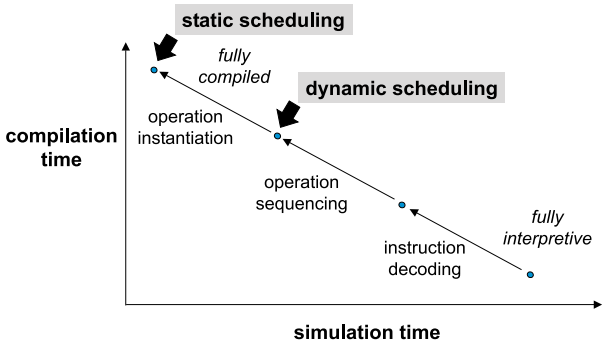

Figure 1: Levels of compiled simulation.

- The step of **instruction decoding** determines the instructions, operands and modes from the respective instruction word. The pipeline structures found in modern DSPs make it obvious that the simulation of these operations consumes a significant amount of simulation time. If we take for example the Texas Instruments TMS320C62x DSP, most instructions actually execute within only one pipeline stage (or cycle), whereas fetching, dispatching, and decoding require six pipeline stages (or cycles). In case of self-modifying code, this step has to be performed again for the affected part of the DSP program. However, self-modifying code is usually not used in signal processing applications.
- The step of **operation sequencing** determines the sequence of operations to be executed for each instruction of the application program. This step can be implemented in a compiled simulator by generating a two dimensional table (see figure 2). One dimension of this table represents the instructions of the DSP program, the other contains pointers to functions that contribute to the transition function which drives the simulator into the next control step.

address	simulator function	simulator function	simulator function	simulator function
80561	&sim_func_11	&sim_func_12	&sim_func_13	...
80562	&sim_func_21	&sim_func_22	&sim_func_23	...
80563	&sim_func_31	&sim_func_32	&sim_func_33	...
...	...	...	...	

Figure 2: Simulation Table.

- **Operation instantiation and simulation loop unfolding** unfolds the simulation loop that drives the simulation into the next state and instantiates the respective simulation code for each instruction of the application program. This is implemented in the compiled simulator by generating individual behavioral code for each instruction of the DSP program.

Between the two extremes of fully compiled and fully interpretive simulation, partial implementation of the compiled principle is possible by implementing only some of these steps. Higher levels of compiled simulation can be achieved by investing substantially more design effort and exploiting highly architecture-specific properties. There are two levels of compiled simulation which are of particular interest – the levels which we call *static scheduling* and *dynamic scheduling* of the simulation. In case of the dynamic scheduling, the task of selecting operations from overlapping instructions in the pipeline are scheduled at run-time of the simulation. The static scheduling already schedules the operations at compile-time.

4 Model Requirements of the Simulation Compiler

Beyond the general requirements of retargetable simulators that are generated from machine descriptions, compiled simulation requires specific information on the target processor architecture to perform the above steps of the simulation compiler.

4.1 Decoding

During the decoding step, the instruction type, the operands and execution modes are determined. The operands may come from different sources (registers file, immediate, indirect) and they can have different types (signed, unsigned, fixed-point, floating point) and word lengths. Execution modes and condition codes may further specify the operation. Decoding is performed by extracting this information from the respective instruction word.

DSPs typically feature extensive non-orthogonal instruction set coding which makes decoding complex and rises considerable issues in the formal capture of the decoding mechanisms using a machine description language. Most approaches such as nML avoid this problem by capturing the non-orthogonal coding in the behavioral model. However, this is no representation which hardly allows to distinguish (simulation) *run-time* operations from *compile-time* operations – those operations that already can be performed during simulation compilation. In LISA, the distinction between these two types of operations is made by means of particular IF-ELSE and SWITCH-CASE statements which are discussed later.

4.2 Operation Sequencing

In order to perform operation sequencing, the precedence of operations composing one instruction and the inter-instruction precedence must be determined. The complexity of this task rapidly grows with the depth and mechanisms of the instruction pipeline.

The processor model must provide detailed information on the pipeline structure and its mechanisms in order to enable this step. The LISA language with its detailed pipeline model enables the description of all pipeline structures and the intra-instruction precedence of operations. Figure 3 shows the intra-instruction precedence relations of operations for a simple four-stage pipeline (with the stages IF, ID, EX, WB).

Figure 3: Intra-instruction precedence.

4.3 Operation Instantiation

The inter-instruction precedence of operations can be derived from the overlapping of instructions in the pipeline for the case that no control hazards occur. Figure 4 depicts both, the intra- and inter-instruction precedence relations of operations. The simulation compiler has to compose operations from overlapping instructions to form the transition function that drives the simulation into the next state. Such operations are shown in vertical columns in figure 4.

Due to control hazards such as jumps, branches and exceptions, the program execution may follow different

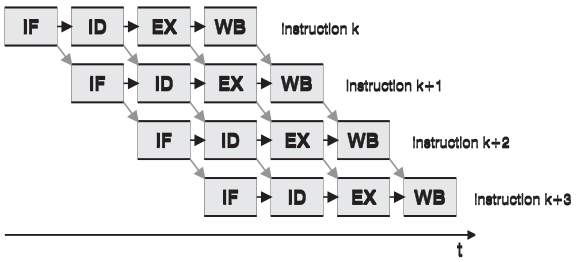

Figure 4: Precedence of instructions in the pipeline.

paths which causes multiple possible combinations of operations in the pipeline. For this reason, the simulation compiler has to generate code for all these possible combinations. During run-time of the simulation, the appropriate path is selected. This means that the simulation compiler must be able to identify control flow instructions – those instructions modifying the program counter.

In LISA processor models, the program counter is a distinct resource and accesses to this resource can easily be identified, such identifying control flow instructions and control hazards. Furthermore, structural hazards can be identified through the predefined mechanisms of the underlying generic pipeline model used in LISA. Predefined pipeline operations comprise flushes, stalls, and instruction injection.

5 LISA Language

LISA descriptions are composed of *resource declarations* on the one hand and of *operations* on the other hand. The declared resources build the storage objects of the hardware architecture (e.g. registers, memories, pipelines) which capture the state of the system and which can be used to model the limited availability of resources for operation access.

Operations are the basic objects in LISA. They represent the designer's view of the behavior, the structure, and the instruction set of the programmable architecture. Operation definitions collect the description of different properties of the system, i.e. operation behavior, instruction set information, and timing. These operation attributes are defined in several *sections*.

- The CODING section describes the binary image of the instruction word which is part of the instruction set model.
- The SYNTAX section describes the assembly syntax of instructions and their operands which is part of the instruction set model.
- The BEHAVIOR and EXPRESSION sections describe components of the behavioral model in C or C++. During simulation, the operation behavior is executed and modifies the values of resources which drives the system into a new state.
- The ACTIVATION section describes the timing of other operations relative to the current operation.
- The SEMANTICS section specifies the instruction semantics.
- The DECLARE section contains local declarations of identifiers and groups of alternative elements.

Operations are formed by a header line and the operation body. The header line consists of the keyword OPERATION and its identifying name:

```
OPERATION name_of_operation
{
  sections...
}
```

Enclosed in curly braces, the operation body contains the different sections which describe the properties of the instruction set model, the behavioral model, the timing model, and required declarations. For more details on the LISA language, please refer to [8].

5.1 Formal Description of Non-orthogonal Coding Fields

In LISA, non-orthogonal coding is expressed by additional conditional statements that can be used to structure the processor model. The purpose of these new conditional statements is to express the coding dependencies between different operations. Following the syntax of programming languages, they have the form of IF-ELSE and SWITCH-CASE statements.

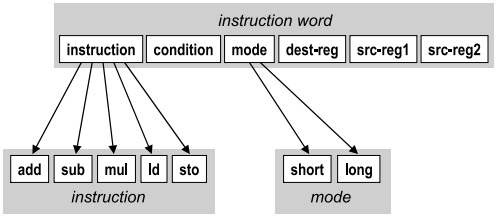

Figure 5: Non-orthogonal coding fields.

We will now discuss an example. Figure 5 displays the coding of a simplified instruction word. There are three instructions `add`, `sub`, and `mul` whose execution is also

controlled by the coding field `mode` which selects between `short` and `long` operands and their specific arithmetic. However, the other instructions `ld` and `sto` use the `mode` field for a different purpose. Possible LISA code for the `add` instruction is shown in example 1.

```
OPERATION add
{
  DECLARE { REFERENCE mode; }

  IF (mode == short)
    {
      BEHAVIOR
        {
          dest_lo = src1_lo + src2_lo;
        }
    }
  ELSE
    {
      BEHAVIOR
        {
          dest_lo = src1_lo + src2_lo;
          carry = dest_lo >> 16;
          dest_lo &= 0xFFFF;
          dest_hi = src1_hi + src2_hi + carry;
        }
    }
}
```

Example 1: Formal expression of non-orthogonality.

Here, the IF-THEN-ELSE statement encloses two alternative sections with their respective behavioral description of the operation `add`. This formal representation lets the simulation compiler distinguish these two cases and generate specific simulation code.

6 Implementation Results

In order to evaluate the applicability and efficiency of compiled simulation in a retargetable environment, we implemented the first two steps of section 3 – compile-time decoding and operation sequencing – in our experimental tool suite. As shown in Figure 6, a LISA compiler takes the processor model and translates in into a data base. The information in this data base is accessible for the simulation compiler generator which produces a processor-specific simulation-compiler. This simulation compiler generates the table

We have chosen the Texas Instruments TMS320C6201 DSP as reference processor for our experimental analysis of the obtainable simulation speed. The TMS320C6201 was described in LISA as a cycle-based model. Although the architecture of this processor with two pipelines consisting of eleven pipeline stages is very complex, the LISA description including the memory interface was realized by one designer in 6 weeks. The complete translation of this model with the LISA compiler and the simulation compiler generator takes less

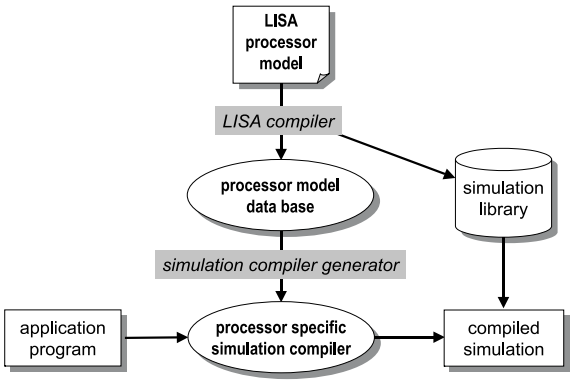

Figure 6: Retargetable, compiled simulation tools.

than 35 seconds on a Sparc Ultra 10 workstation. As a comparison, a custom compiled simulator for the less complex TMS320C54x (six-stage pipeline) the same designer has spent more than 12 months.

6.1 Simulator Benchmarks

In order to evaluate the simulation speed of our generated, compiled simulator of the TI C6201 we used the `sim62x`, version 2.0 which is part of the TI's software development tools for our reference. The benchmarks are based on three typical DSP algorithms, a FIR filter, the ADPCM G.721 codec, and the GSM speech encoder. All measurements were made on a Sparc Ultra 10.

Compilation time of object code into a compiled simulation was measured on three reference applications. The required time and respective application size is shown in figure 7. The compilation speed is calculated by relating the number of compiled instructions to compilation time.

Application	FIR	ADPCM	GSM
code size [kByte]	2112	3072	58592
compilation time [s]	0.97	1.38	27.31
compilation speed [instructions/s]	544	557	536

Figure 7: Simulation Compilation Speed.

For all applications measured, the compilation speed ranges between 530 and 560 instructions/s, even for our GSM coder that nearly requires the whole internal memory space of the DSP.

Simulation speed was quantified by running an application on the respective simulator and relating the simulation time to the processed number of cycles. The measurement results are depicted in figure 8. The reference simulator from TI achieved between 2k and 9k

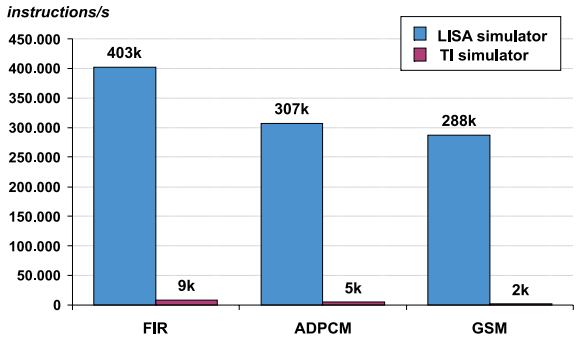

Figure 8: Simulation Speed.

cycles/s whereas our generated simulator runs at speeds between 288k and 403k cycles/s at the same accuracy level. This corresponds to factors of 47x to 170x faster simulation as shown in figure 9.

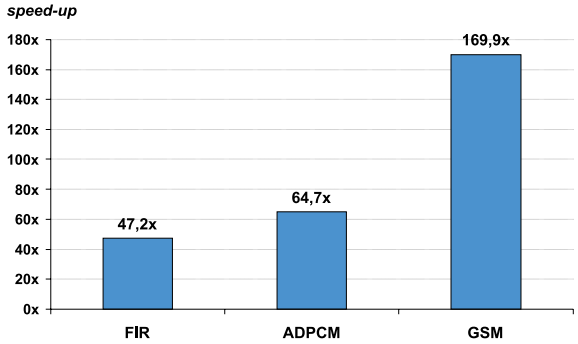

Figure 9: Speed-up: LISA simulator vs. TI sim62x.

7 Conclusion and Future Work

In this paper, we presented the new approach of applying compiled simulation to retargetable processor simulation environments. The compiled technique is a technology that enables fast simulation of programmable DSP architectures. Up to now, compiled simulation has only been implemented for specific processor architectures. Retargetable, compiled simulation based on a machine description language puts specific requirements on the instruction set model. The complete instruction coding must be described formally to enable a high degree of compiled simulation. The processor description language LISA is able to provide such models. In a case study, compiled simulation techniques are implemented for a model of the Texas Instruments TMS320C6201 DSP. Our generated, compiled simulator based on the LISA description runs at 47-170 times higher simulation speed than the commercially available instruction set simulator from TI without any loss in accuracy.

Our future work will focus on modeling further real-life processor architectures and retargetable compiled simulators that provide the third step of compilation – operation instantiation. Another issue is the integration of software simulators into HW/SW co-simulation environments. Furthermore, the goal of the ongoing language design is to address retargetable compiler backends as well.

References

[1] M. Hartoog, J. Rowson, *et al.*, "Generation of software tools from processor descriptions for hardware/software codesign," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, Jun. 1997.

[2] J. Rowson, "Hardware/Software co-simulation," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1994.

[3] V. Živojnović, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in *Proc. of 1995 IEEE Workshop on VLSI Signal Processing – Sakai, Osaka*, pp. 187–196, Oct. 1995.

[4] S. Pees, V. Živojnović, A. Ropers, and H. Meyr, "Fast Simulation of the TI TMS 320C54x DSP," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, (San Diego), pp. 995–999, Sep. 1997.

[5] M. Barbacci, "Instruction set processor specifications (ISPS): The notation and its application," *IEEE Transactions on Computers*, vol. C-30, pp. 24–40, Jan. 1981.

[6] A. Fauth and A. Knoll, "Automatic generation of DSP program development tools using a machine description formalism," in *Proc. of the ICASSP - Minneapolis, Minn.*, 1993.

[7] V. Živojnović, S. Pees, and H. Meyr, "LISA — machine description language and generic machine model for HW/SW co-design," in *Proceedings of the IEEE Workshop on VLSI Signal Processing*, (San Francisco), Oct. 1996.

[8] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," in *Proceedings of the Design Automation Conference (DAC)*, (New Orleans), pp. 933–938, June 1999.

[9] D. Bradlee, R. Henry, and S. Eggers, "The Marion system for retargetable instruction scheduling," in *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada*, pp. 229–240, 1991.

[10] B. Rau, "VLIW compilation driven by a machine description database," in *Proc. 2nd Code Generation Workshop, Leuven, Belgium*, 1996.

[11] M. Freericks, "The nML machine description formalism," Tech. Rep. 1991/15, Technische Universität Berlin, Fachbereich Informatik, Berlin, 1991.

[12] A. Fauth, M. Freericks, and A. Knoll, "Generation of hardware machine models from instruction set descriptions," in *Proceedings of the IEEE Workshop on VLSI Signal Processing*, 1993.

[13] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. European Design and Test Conf., Paris*, Mar. 1995.

[14] J. Van Praet, D. Lanneer, et *al.*, "A graph based processor model for retargetable code generation," in *Proceedings of the European Design and Test Conference (ED&TC)*, Mar. 1996.

[15] Texas Instruments, *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Mar. 1998.

[16] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, Jun. 1997.

[17] A. Halambi, P. Grun, et *al.*, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proceedings of the European Conference on Design, Automation and Test (DATE)*, Mar. 1999.

[18] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," *Computer Architecture News*, vol. 25, pp. 13–25, June 1997.

[19] C. Siska, "A processor description language supporting retargetable multi-pipeline dsp program development tools," in *Proceedings of the International Symposium on System Synthesis (ISSS)*, Dec. 1998.

[20] Z. Barzilai, *et al.*, "HSS - A high speed simulator," *IEEE Trans. on CAD*, vol. CAD-6, pp. 601–616, July 1987. 1987.

[21] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[22] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. C-36, pp. 24–35, Jan. 1987.