ADMS – A Fully Customizable Compact Model Compiler

Ben Gu and Laurent Lemaitre
{benjamin.gu,laurent.lemaitre}@freescale.com

Freescale Semiconductor

Abstract

This paper presents an overview of ADMS, a fully customizable compact model compiler. The architecture of ADMS and syntax of ADMST language are introduced. A few optimization techniques which can be employed to build an efficient customized model compiler using ADMS are briefly reviewed, and effects of these techniques were demonstrated using the PSP model.

Keywords: model compiler, compact model, model optiimzation

Introduction

Compact device models are an important element of circuit simulators like SPICE. The accuracy and performance of models to a large extent determines the quality and reliability of simulations. Conventionally, compact models are plugged into a simulator through an application programming interface (API). Using this approach, model developers must code the compact model using the data structures defined in the API and write several interface routines to communicate with the simulator [1]. This approach has proved to be very inefficient. Coding the model equations and the partial derivatives of charges and currents in a programming language can be very tedious and error prone, which makes implementing a model into a simulator often take months [2]. In addition, the approach is quite inflexible: The code developed for simulator A can't be used in simulator B without extensive changes, and any modification to device nodes, model parameters, or equations will result in substantial changes to the code.

Over the last few years, the Verilog-A language has arisen as a new standard for compact model development [3-6]. Developing and delivering models as Verilog-A modules liberates model developers from the tedium of coding equations in a programming language. Further, the error-prone writing of analytical partial derivatives, needed for solving equations using Newton iterations, is handled automatically, which accelerates the development process and improves the quality of compact models. More importantly, models developed in Verilog-A format are ready to run on any circuit simulator which supports the Verilog-A language. Because of these advantages, the Verilog-A language has been adopted by several leading compact model developers, such as the Berkeley BSIM group [7], Arizona State's PSP group [8], and so on.

On the other hand, as more and more models are developed and delivered in Verilog-A format, there comes a new challenge for simulator developers: How should they implement these models into simulators automatically and efficiently? They could just leave the model in Verilog-A, relying on the simulator's Verilog-A implementation. Often, this is an interpretive evaluation, in which the Verilog-A code describing the model equations is evaluated on the fly. This approach is not optimal for large simulations, as interpretation is slow and not competitive with compiled built-in models. Even if the simulator handles Verilog-A through compilation, there will be overhead with the more general Verilog-A language which will not exist for built-in models.

Another approach, which we take here, is a custom-compiled approach. The Verilog-A code is compiled into a common programming language, i.e. C, by some specialized software (compact model compiler), respecting the simulator's compact model API and effectively making the Verilog-A module into a built-in model. Using this approach, the computational efficiency of the models can be brought up to a level which is comparable to built-in models implemented by experienced engineers. Recently there has been significant interest in developing compilers which convert compact models defined in Verilog-A into a programming language (typically C) for implementation into simulators [9-11]. Among those compilers, the Advanced Device Model Synthesizer [12] (ADMS), an open source program developed at Freescale Semiconductor, is becoming increasingly popular. Because of its open source availability and agile architecture, it has been used by several EDA vendors and semiconductor companies for different applications. In this paper, we present an overview of the architecture of ADMS, its data structure, and the syntax of the ADMST language. We also review a few optimization techniques which can be employed to build an efficient customized model compiler using ADMS.

Architecture of ADMS

ADMS is a compact model compiler which users can customize according to their own specifications. The software strategy employed by ADMS is somewhat similar to the compiler-compiler techniques used in the GNU bison program. As illustrated in Fig.1, The ADMS software has two major components: a Verilog-A parser and an ADMST interpreter. The Verilog-A parser reads in a Verilog-A source file describing the compact model and stores information of the model in a data structure which is similar

to a XML tree. The ADMST interpreter parses and executes users' definition of compilers given in a ADMS-specific format, ADMST. The ADMST language is very similar to the XSLT and XPATH languages used in XML technologies, and this language can be used to perform operations on the XML data of the model so as to build a fully customized compiler. The ADMST interpreter reads the instructions written in ADMS scripting languages and transforms the XML tree into any other form of presentation (e.g. C, Matlab, html). This software architecture allows users to build their own applications flexibly. For instance, a user can develop different sets of ADMST scripts to implement Verilog-A models into different simulators, i.e. SPICE or any commercial simulator having an exposed model API. Using ADMST scripts, ADMS can also be extended to other applications such as automatically creating test-bench circuits or documentation for models.

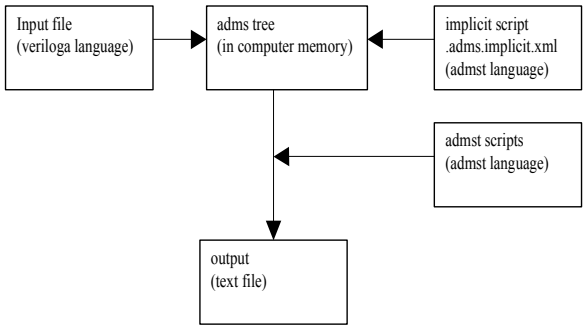

Fig. 1. Architecture of the ADMS system

ADMS Data Tree

After a Verilog-A source file is parsed, a data tree is built in memory. This XML tree is a representation of the model data stored in the Verilog-A file. Figure 2 shows a visual representation of the ADMS data tree that has been created for a simple module. The picture has been simplified; the actual XML tree has many more branches and nodes.

```
//Verilog-AMS code
module myR (p,n);
  parameter real G=1;
  analog begin
    I(p,n)<+G*V(p,n);
  end
endmodule
```

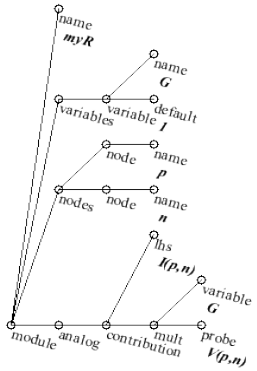

Fig. 2. ADMS Data Tree (Simplified View)

ADMST and ADMSTPATH

The language used to describe the behavior of the compiler is very similar to the XSLT and XPATH languages used in XML technologies. Missing features in the XSLT language for this particular application and the lack of available high-speed open-source XPATH interpreters required us to abandon a fully-XML based approach. We therefore defined an ADMS-specific system. However, a significant level of backward compatibility has been maintained. At some time it will be still possible to move back to the XML system with minimal effort when it is extended to handle capabilities necessary for ADMS.

The "Hello World" routine (German version) is a starting point to illustrate the syntax of the ADMST – the language tells ADMS how to transform the ADMS data tree. Figure 3 gives an implementation of a generic "Hello World" routine (lines #2 to #4).

```
1. $ cat hello.xml
2. <admst version="2.2.0">
3.   <admst:text format="Guten Morgen Welt\n"/>
4. </admst>
5. $ admsXml -e hello.xml
6. Guten Morgen Welt
```

Fig. 3. "Hello World" in the ADMST language.

This example illustrates how close the syntax of the ADMST language is to the HTML language – a subset of the XML system. The ADMST language can be defined using following recursive definition:

```
1. <admst:instruction attr="val">
2.   …other admst:instructions (children like)
3. </admst:instruction>
```

ADMST scripts are built by using the above definition recursively. The structure of ADMST scripts is a tree. Each node of the tree is an **admst:instruction**. The XML interpreter of ADMS traverses all instructions. When it enters (or leaves) an **admst:instruction** it executes some procedures. The keyword **instruction** is a placeholder. It can take one of the following self-explanatory values: *text, for-each, if, choose, template, apply-templates, open, when, join, value-of, value-to, push, reverse, message, fatal*, and so on. When the XML interpreter traverses node **adms:text** as shown in Fig. 3 it calls a routine that prints the text of line #6.

The meaning of pairs **attr="val"** is trickier. To keep it simple, each **attr="val"** pair acts as an argument to every **admst: instruction**. At line #3 attribute **format** passes to the ADMST instruction **admst:text** the actual value of the string that will be printed. An almost ubiquitous attribute is the **select** attribute. This attribute specifies the position of the XML interpreter inside the ADMS data tree. The language used to specify the location inside each **admst:instruction** is the ADMSTPATH language.

The script given in Fig. 4 illustrates the concepts introduced so far. Here is the sequence of events that will occur in the course of compilation of the script: file **myfile** is opened for writing. Then all nodes **module** of the ADMS data tree are traversed. Starting from each **module** all nodes

node are in turn traversed. Finally the attribute name of each **node** is printed in file **myfile**.

```
1.  <admst:open file="myfile">
2.    <admst:for-each select="module">
3.      <admst:for-each select="node">
4.        <admst:value-of select="name">
5.        <admst:text format="node is: %s\n">
6.      </admst:for-each>
7.    </admst:for-each>
8.  </admst:open>
```

Fig. 4. Simple script illustrating the ADMST syntax.

Details and a pragmatic introduction to the ADMS system can be found at the home page of ADMS [12].

Optimizing Performance of Generated Code

The ADMST language provides a framework on which users can build their own model compilers. One of the most essential requirements for a model compiler is the capability of generating computationally efficient computer code to evaluate the model described in Verilog-A format. In this section, we review a few techniques users of ADMS can employ to enhance the computational efficiency of the generated code. These techniques has been implemented in the ADMST language in the compact model compiler developed for the Freescale in-house circuit simulator, Mica, and have been used to significantly improve the computational efficiency of the the PSP model in Mica. The PSP model is a surface potential based compact model, which was chosen by the compact model council (CMC) as a next-generation standard model for MOSFET devices [5,8]. The model has always been developed and distributed in the Verilog-A format, and there has been an active discussion in CMC about the model's computational efficiency since it was selected as a standard, which makes it particularly apropos for our discussions in this paper.

Code Partitioning

A compact model typically consists of a large number of equations evaluating currents and charges of a physical device described by the model. These equations are partitioned into three sections: a process-dependent section, a geometry-dependent section and a bias-dependent section. The process-dependent section deals with the computations based only on model parameters that reflect characteristics of the fabrication process. This section is often alternatively referred to as the model initialization section and it is evaluated only once for each individual model in a given circuit during a simulation. The second section does computations involving geometry-dependent quantities, such as width, length, area, multiplicity, and so on. This section is often referred to as instance/geometry initializations and it is evaluated once for every device instance that uses a particular model in the circuit. The last section groups the computation of all quantities that depend on the electrical bias conditions. Evaluation of this block is repeated at every iteration during the simulation process. In the PSP model, the Verilog-A code has been partitioned by the model developers, however an intelligent compiler should efficiently determine an optimized partitioning of the physics-based equations that avoids overloading of the bias-dependent section. After Verilog-A code of the model is carefully partitioned, the primary candidate for code optimization is the bias-dependent partition as this part of the code will be evaluated at every iteration of the simulation.

Bias Dependency Tracking

In order to make the model work with simulators, a compact model compiler should not only generate code to evaluate all the equations presented in the Verilog-A code, but also generate code to analytically compute partial derivatives of those equations with respect to (w.r.t) the solution variables (typically the device bias voltages). Even so, many expressions in the model do not require differentiation, only those which are directly dependent on solution variables. Thus ADMST scripts should track these dependencies and generate the appropriate partial derivatives only when needed, via application of the chain rule.

Reuse of Expensive Math Computations

The quality of the partial derivative code generation is essential to the efficiency of the compiler. Below is an equation in the section evaluating mobility reduction in the Verilog-A code of the PSP model,

```
Mutmp = pow(Eeffm * MUE_i, THEMU_i) + CS_i *
        (Pm / (Pm + Dm + 1.0e-14));
```

For this equation, an unoptimized compiler will generate C code as follows:

```
Mutmp_Vs_bp
  = pow(Eeffm * MUE_i, THEMU_i) *
    MUE_i * Eeffm_Vs_bp / (Eeffm * MUE_i) +
    CS_i * (Pm_Vs_bp * (Pm + Dm + 1.0e-14) -
    Pm * (Pm_Vs_bp + Dm_Vs_bp)) /
    ((Pm+Dm + 1.0e-14) * (Pm + Dm + 1.0e-14));
Mutmp_Vd_s
  = pow(Eeffm * MUE_i, THEMU_i) *
    MUE_i * Eeffm_Vd_s / (Eeffm * MUE_i) +
    CS_i * (Pm_Vd_s * (Pm + Dm + 1.0e-14) -
    Pm * (Pm_Vd_s + Dm_Vd_s)) /
    ((Pm + Dm + 1.0e-14) * (Pm + Dm + 1.0e-14));
Mutmp_Vgp_s
  = pow(Eeffm * MUE_i, THEMU_i) *
    MUE_i * Eeffm_Vgp_s / (Eeffm * MUE_i) +
    CS_i * (Pm_Vgp_s * (Pm + Dm + 1.0e-14) -
    Pm * (Pm_Vgp_s + Dm_Vgp_s)) /
    ((Pm + Dm + 1.0e-14) * (Pm + Dm + 1.0e-14));
Mutmp = pow(Eeffm * MUE_i, THEMU_i)
  + CS_i * Pm / (Pm + Dm + 1.0e-14);
```

The first three lines of code compute the partial derivatives of Mutmp w.r.t to three branch voltages, V(S,BP), V(D,S), and V(GP,S) respectively, and the last line computes Mutmp itself. Obviously the code is far from being efficient. The first thing we notice is that evaluation of the same pow() function is repeated four times in the generated code. This will make the code unnecessarily slow given that evaluating the pow() function is expensive. So one optimization technique could be to evaluate the

expensive functions first, such as log, exp, pow, sqrt, and so on, then reuse the computations in the generated code. In addition to these functions, the divide operation is also expensive. On many CPU architectures, a divide operation is as expensive as evaluating a sqrt() function, so opportunities to reuse divide operations can be exploited as well. With these techniques implemented, an optimized compiler could generate code as follows:

```
__pow_0 = pow(Eeffm * MUE_i, THEMU_i - 1.0);
__dF1_pow_0 = (THEMU_i) * __pow_0;
__pow_0 = (Eeffm * MUE_i) * __pow_0;
__dF1_div_1 = 1.0 / (Pm + Dm + 1.0e-14);
__div_1 = (Pm) * __dF1_div_1;
__dF2_div_1 = -__div_1 * __dF1_div_1;

Mutmp_Vs_bp
  = (__dF1_pow_0 * MUE_i * Eeffm_Vs_bp) +
    CS_i * (__dF1_div_1 * Pm_Vs_bp
     + __dF2_div_1 * (Pm_Vs_bp + Dm_Vs_bp));
Mutmp_Vd_s
  = (__dF1_pow_0 * MUE_i * Eeffm_Vd_s) +
    CS_i * (__dF1_div_1 * Pm_Vd_s
    + __dF2_div_1 * (Pm_Vd_s + Dm_Vd_s));
Mutmp_Vgp_s
  = (__dF1_pow_0 * MUE_i * Eeffm_Vgp_s) +
    CS_i * (__dF1_div_1 * Pm_Vgp_s
    + __dF2_div_1 * (Pm_Vgp_s + Dm_Vgp_s));
Mutmp = __pow_0 + CS_i * __div_1;
```

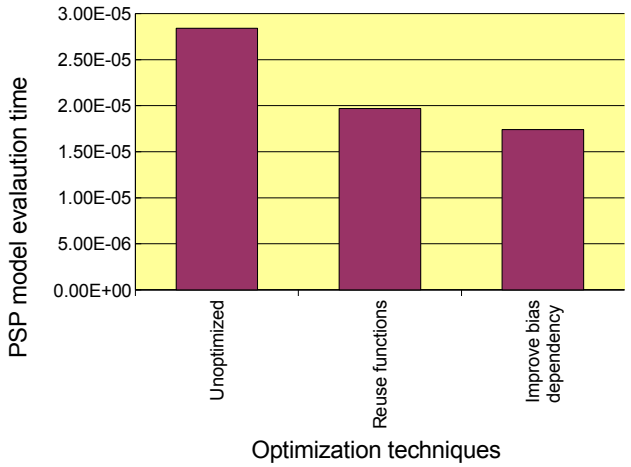

Fig. 5. Performance improvements of the PSP model

Figure 5 summarizes performance improvements of the PSP model achieved by cumulatively using the techniques we have reviewed in this section. The result presented is an average of one million evaluations of the code generated by the customized model compiler for Mica, using ADMS. The .model statement used in the experiment is from one of bulk technologies developed by Freescale, and node voltages at each evaluation are randomly generated from 0 to 3V. As shown in Fig. 5, using these techniques collectively can lead to a respectable result, speeding up the computational performance of the PSP model by nearly 1.6x.

Conclusions

This paper reviews architecture of ADMS, a fully customizable a compact compiler and syntax of ADMST language with which users define their specifications of device compilers. Several common optimization techniques to improve the computational efficiency of compilers built by ADMS are introduced by using the PSP model as an example.

Acknowledgments

Authors are sincerely grateful to Steve Hamm in Freescale for his extremely useful suggestions on improving the manuscript. Also they would like to thank Colin McAndrew and every member of Mica group in Freescale for being wonderful to work with.

References

[1] T. Quarles, “Adding Devices to SPICE3” (UCB/ERL M89/45, April 1989)
[2] K. Kundert, “Automatic Model Compilation, An idea whose time has come”, http://www.designers-guide.org/Perspective/modcomp.pdf
[3] L. Lemaitre, et. al. , “Extensions to Verilog-AMS to Support Compact Device Modeling”, IEEE BMAS 2003
[4] G. Coram, “How to (and how not to) write a compact model in Verilog-A”, IEEE BMAS 2004
[5] Compact Modeling Council meeting minutes, http://www.geia.org/index.asp?bid=597
[6] Verilog-AMS language reference manual http://www.eda-stds.org/verilog-ams/.
[7] http://www-device.eecs.berkeley.edu/
[8] http://pspmodel.asu.edu/
[9] B. Wan, B. Hu, L. Zhou, C. Shi, “MCAST: an abstract-syntax-tree based model compiler for circuit simulation,” Proc. IEEE CICC, pp. 249-252, Sept. 2003.
[10] B. Troyanovsky et al., “Analog RF Model Development With Verilog-A,” MTT-S 2005
[11] L. Lemaitre et. al. IEEE CICC Proc. pp. 27-30, 2002
[12] http://sourceforge.net/projects/mot-adms/
[13] http://www.gnu.org/software/bison/manual/