Automatic Generation of Functional Models for Embedded Processor Extensions

Fei Sun
Tensilica Inc. 3255-6 Scott Boulevard, Santa Clara, CA, 95054

***Abstract*—Early architectural exploration and design validation are becoming increasingly important for multi-processor systems-on-chip (MPSoC) designs. Native functional simulations can provide orders of magnitude in speedup over cycle or instruction level simulations but often require dedicated maintenance.**

In this work, we present a tool called NATIVESIM to automatically generate the functional models for embedded processor extensions. We provide a mechanism to address the challenge of modeling a subset of the processor architecture, with no visibility to the rest of the processor. We illustrate the problem of modeling the processor extensions when the endianness of the target processor is different from the host system and provide a solution to it. Experiments on several benchmark programs indicate that native execution of the target application with the functional models of the processor extensions can achieve large simulation run-time speedup over simulations based on either cycle accurate models (up to 14102× with an average of 3924×) or compiled functional models of an entire processor (up to 103× with an average of 31.6×).

I. Introduction

Native simulation of the functional models of the MPSoC system and native validation of the target applications become popular in the early stage of the design cycle. However, as implementation details are added to the design, the native models quickly become obsolete. Keeping them synchronized with the design is a very tedious and error prone process.

This problem is more challenging on configurable and extensible processors [1], [2], whose instruction set architecture (ISA) can be extended during the design process. Because the processor extensions such as custom instruction prototypes and custom data types are referenced directly in the target applications (*i.e.* applications ported on the configurable and extensible processors) and the native compiler does not have any knowledge of such extensions, the target applications become target dependent and cannot be compiled by the the native compiler and executed on the host system.

In this paper, we present a tool, NATIVESIM, to automatically generate the functional models for the embedded processor extensions in the form of C or C++ libraries. The functional models are called Cstubs. The target applications, linked with Cstubs, can be modeled natively with little or no modification. The designers have the freedom to integrate Cstubs in their MPSoC functional models, which significantly reduces the workload of maintaining the functional models.

Because Cstubs accurately model the processor extensions during application execution, they may also be used as validation tools to validate the embedded processor extensions or the target applications ported to the embedded processor.

Automatically generating the functional models of embedded processor extensions fills a gap in system exploration, eliminating the need to rewrite models as implementations are fine tuned. It also provides a fast validation technique for the embedded processor extensions and the target applications. To the best of our knowledge, this tool is the first one to extract the functional models from embedded processor extensions.

978-3-9810801-8-6/DATE12/©2012 EDA

II. Related work

Native simulation, untimed or loosely timed modeling has been the focus of research for years. A good overview of the transactional level modeling (TLM) from instruction-accurate interpretation to native simulation is contained in [3].

Hybrid simulation techniques are proposed in [4], [5] to execute the target independent portion of the application on the host system and execute the target dependent portion on an ISS. On extensible processors, however, the target dependent portion is usually the timing critical part of the application. The Cstubs generated by our tool enable simulating the target dependent portion natively, which helps to improve the simulation speed further.

Native simulation techniques annotated with timing information to estimate the performance of the target application are presented in [6], [7]. The functional models presented as Cstubs could possibly be integrated into those native simulators to extend their capabilities to handle configurable and extensible processors.

III. Methodology

In this section, we describe our tool, NATIVESIM, to generate the functional models from the embedded processor extensions. We first outline each part of NATIVESIM, and then present the details of several modules.

An embedded processor can be extended by adding custom instructions, custom register files, custom states, *etc.* in the form of a processor description language. The extensions are referenced either explicitly or implicitly in the target application. NATIVESIM generates the C/C++ models of such extensions so that the target applications utilizing the extensions can be compiled and executed natively on the host system.

Fig. 1 describes the flow of generating the functional models of the processor extensions. The tool first reads in a description of the processor extension and analyzes the characteristics of the extension (module 1). It sends different extensions to different functional model generation modules.

Similar to the general-purpose register file and its built-in types (*e.g.* `int`, `short`, `char`), processors can be extended with custom register files and custom types. The custom types can be referenced in the target application just like the built-in types. The custom register file analyzer (module 2) reads in the description of the custom register files. The custom type generator (module 3) reads in the description of the custom data types and generates a model for each custom type. They are described in Section III-A. Unlike the custom register files, the custom states are used implicitly by the custom

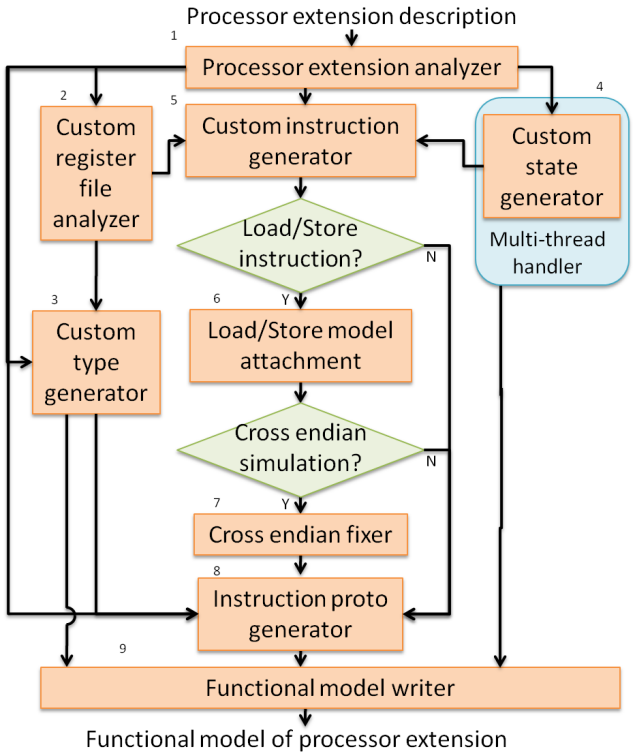

Fig. 1. Functional model generation flow of processor extensions

instructions. The custom state generator (module 4) generates the functional model for all architectural visible states of the processor. It is described in Section III-B.

A custom instruction is not included in the instruction-set of the base processor. It is implemented by the designers to speed up a specific application. The custom instruction generator (module 5) generates the functional models of the computation part of the custom instructions. The module is described in detail in Section III-C. If the custom instruction is a load or store instruction, aside from the computation portion, the instruction needs to access the memory. Thus, the load/store model needs to be attached to the instruction model (module 6), which is described in Section III-D. If the endianness of the host system is different from the endianness of the target processor, the load or store instruction may not retrieve or save the desired custom type values in the correct order. A new processor extension construct is introduced to assist the tool in identifying such situations and fix the cross endian simulation issue (module 7). A detailed description of the construct can be found at Section III-E.

The custom instructions are inserted into the application code through the means of custom instruction prototypes, which hide the architecture invisible portions of the custom instructions. An instruction prototype may contain multiple custom instructions, which complicates the functional model generation (module 8). Section III-F describes the functional model generation of instruction prototypes in detail.

Finally, the functional model writer (module 9) combines the functional models of all the processor extensions into a unified library in C or C++.

A. Custom register file analyzer and custom type generator

An embedded processor may contain multiple register files, with one or multiple data types mapped to each register file. Similar to the built-in data types `short` and `int`, a custom type variable can be used directly in the application code. The custom type generator (module 3 in Fig. 1) evaluates each custom type and the corresponding register file, and generates the custom type model, which is responsible for storing the variable values in their memory form.

Because only custom instructions read from and write to custom register files, the functional models do not model custom register files directly. Instead, they are modeled implicitly in the custom instruction models and their interaction with the custom type models.

A custom type variable must be aligned to the same boundary as on the target architecture. Its size in memory may not be the same as the size of its corresponding custom register file. When the value of a custom type is loaded from memory, the data may need to be expanded to the size of the custom register file (*e.g.* sign extension) by the load instruction specified by the designer. It can be easily done in the compiler for the target architecture. However, the functional models of the processor extension have no visibility into the application code utilizing the processor extensions. Thus, it is modeled conservatively that any instruction prototype that references a custom type value automatically converts the value to a register file value before the computation and converts it back to a custom type value after the computation. It is described in detail in Section III-F.

B. Custom state generator

A custom state behaves like a single entry custom register file in hardware, but their software implications are very different. Because a custom state contains one copy of data, the custom instruction can infer the accessed custom states from its opcode. The custom state generator (module 4 in Fig. 1) creates a model for all states of the processor. In a single processor system, the processor states are unique throughout the system. Thus, only one copy of the model is needed. In a multi-processor system, the same processor may be used in different parts of the system. To keep a separate copy of the processor states for different processors, the models are marked as thread local variables and are created and initialized in the system simulation environment dynamically.

C. Custom instruction computation generator

The instruction set of an embedded processor may be extended with custom instructions. The computation portion of a custom instruction is usually written in a language similar to the hardware description language (HDL), which is a parallel processing language. In functional modeling (module 5 in Fig. 1), a bit accurate data flow graph (DFG) is created to capture the parallel computation. DFG is a unidirectional graph composed of nodes and edges. The nodes are computation operators such as addition, subtraction, and bit-wise operators. The edges are wires connecting the computation operators to represent the data flow between them. The flow to generate an optimized DFG is illustrated in Fig. 2. NATIVESIM reads in the description of a custom instruction and builds an initial DFG (block 1 in Fig. 2). It then rewrites the DFG based on the states, register files, and interfaces the operation accesses (block 2 in Fig. 2).

When several custom instructions perform similar computation, it is desirable to share hardware for them. Thus, a hardware block may be used to compute the result of several custom instructions, each activates a portion of the hardware block. The hardware block is denoted as a semantic. However, in functional modeling, each custom instruction is modeled separately. It is preferrable to duplicate the hardware block for each custom instruction and specialize the hardware block

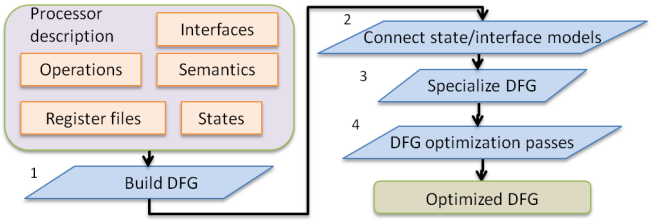

Fig. 2. Data flow graph generation for custom instructions.

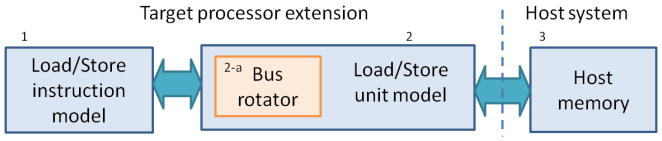

Fig. 3. Block diagram of the interface between the functional models of the load/store custom instructions and the host memory.

for it, *i.e.* only the logic activated by the custom instruction remain. Step 3 in Fig. 2 specializes the DFG for one custom instruction.

After that, the DFG is simplified by dozens of optimization passes (Step 4 in Fig. 2). Each node in the DFG is evaluated for optimization opportunities. Because one optimization may trigger other optimization opportunities, the optimization techniques are performed iteratively. The optimization is complete when no simplification opportunity is identified, or the target number of iterations is met. The optimized DFG is used in the instruction prototype generation, which is described in Section III-F.

D. Load/store model attachment

If a custom instruction is a load or store instruction, besides accessing the register files and states, it reads data from memory or writes data to memory. In the target program, the same memory may be accessed through reads and writes of standard C variables, through reads and writes of custom type variables or through instruction prototypes. The native simulation has no control of the standard C variables so must insure that the instruction prototypes and the custom type variables access the memory in a manner that is consistent with the standard C portions of the application. Thus, a model of the load/store unit is attached to the load/store instruction model, as shown in block 2 of Fig. 3.

The load/store unit model is divided into two parts. The first part retrieves data from the host memory for load instructions and writes data to the host memory for store instructions. The second part performs data manipulation usually done by the the load/store unit, such as rotation, sign extension, raising exception on unaligned address. The DFG of such data manipulation is built and merged into the DFG of the computation portion of the load/store instructions. The second part also handles the cross endian simulation issues, as described in Section III-E.

E. Cross endian fixer

If the target processor is big endian and the host system is little endian, the full fledged ISS builds up a target memory system layer and models the processor on top of it. It keeps track of the endian difference and swaps the memory data order at byte boundaries. However, in functional modeling, even though the application is written for the big endian target processor, it is compiled and executed on the little endian host system directly. This memory model mechanism is completely different from the one in ISS. If the standard C application code is endian dependent, it must be refactored to be endian independent. However, even if the C application is endian independent, the custom instruction may be endian dependent. When a custom instruction loads data from memory, it expects that the data in memory are loaded in the order of the target processor, which may not be the same as the actual order on the host system.

The Cstubs model the processor extensions implemented on the target processor, while the rest of the processor and the memory follow the convention of the host simulation environment. The order of the data values needs to be swapped between the host environment and the target processor extension models, so that they appear in the same order as in the target processor to the custom instructions, as shown in block 2-a of Fig. 3.

However, the functional models of the custom instructions in C/C++ code are still compiled and executed on the host simulation environment. The swapping of bytes is not performed within one data value. Instead, the swapping is performed across multiple data values. Thus, the swapping is not needed when an instruction loads or stores one data element. It is required if the instruction loads or stores multiple data elements. This is different from the cross endian modeling performed by ISS.

In order to make the functional models aware of the necessary swapping patterns, a new construct is added to the processor extension description. NATIVESIM reads in the construct and automatically swaps the data between the custom instructions and the memory when the endianness of the host system and the target processor are different, as shown in block 2-a of Fig. 3. The construct has no effect otherwise. The construct is defined as follows. It associates a swapping pattern with one or more custom instructions. `op-name` is the opcode name of a custom instruction. `swap-pattern` describes the swapping pattern on the memory interface.

cstub_swap {op-name [, op-name]+} {swap-pattern}

F. Instruction prototype generator

To utilize the custom instructions in an application, the designer needs to reference the instruction prototypes in the application, which are also called as instruction intrinsics. An instruction prototype consists one or several instructions that perform one computation task. Instruction prototypes hide the detailed hardware implementation from the software application designer. Thus, the same software source code can be executed on processors with different extensions.

Because the instructions in an instruction prototype are closely related, the instruction prototype generator creates one DFG for each instruction prototype by inlining the DFGs of its composing custom instructions generated in module 5 of Fig. 1. It may reveal further optimization opportunities to reduce the size of the DFG of the instruction prototype.

As described in Section III-A, the custom type variables reside in memory while the custom register file variables reside in the register files. Their values may be different. For example, the value `-1` of type `short` is `0xffff` when it is in memory, but is `0xffffffff` when it is in a 32-bit register file. The application code may modify the values residing in memory without going through custom instructions. However, the values in custom register files are only accessed by custom instructions. To avoid the confusion of modeling two representations of the same value, only the custom type value in memory is modeled.

However, the custom instructions expect the register file inputs and outputs to be the values in the register files. Thus,

TABLE I
RUN TIME COMPARISON BETWEEN CSTUBS, ISS, AND TURBOXIM IN SINGLE PROCESSOR SCENARIO

Benchmark	Cstubs (s)	ISS (s)	speedup	TurboXim (s)	speedup	# custom instrs	# states
MP3MCH decoder	24.7	7041	285×	86.2	3.5×	400	11
AC3 encoder	33.6	15488	461×	99.6	3.0×	400	11
MS10 decoder	12.5	1403	112×	37.3	3.0×	400	11
OSPF	10.6	50336	4749×	161	15.2×	7	15
IP_REASSEMBLY	7.7	41039	5330×	344	44.7×	6	5
ROUTELOOPUP	6.7	8432	1259×	119	17.8×	4	8
NAT	4.6	64871	14102×	385	83.7×	10	14
QOS	3.6	21303	5918×	369	103×	7	11
TCP	9.4	29153	3103×	100	10.6×	10	20

before any custom register file value is used in a custom instruction, the value needs to be converted from its custom type value in memory via the load instruction prototype. After the custom instruction computes a custom register file value, it is converted back to the custom type value via the store instruction prototype. This is achieved in the functional models of the instruction prototypes by inserting the DFG of the load instruction prototype before each register file input and inserting the DFG of the store instruction prototype after each register file output.

IV. EXPERIMENTAL RESULTS

We have implemented NATIVESIM, which reads in the processor description in Tensilica Instruction Extension (TIE) language [8] and generates the functional models of the processor extensions. The tool is integrated in the Tensilica's Xtensa processor generation flow.

We compared the simulation speed between the functional models of processor extensions (Cstubs), the cycle accurate models without memory modeling (ISS), and the compiled functional models of an entire processor (TurboXim). Unlike Cstubs, which only model the processor extensions, TurboXim models the entire processor. It has the capability of compiling the frequently executed target application code traces on the host system and executing them natively.

The comparison is performed on an Intel Core i7-2600 (4 core with hyper-threading) processor with 8GB memory. The applications modeled using Cstubs are compiled using GNU g++ 4.4.4 with -O3 optimization. The same applications modeled using ISS and TurboXim are compiled using xt-xcc 9.0 with comparable optimization. In all performed experiments, native execution of the target applications linked with Cstubs generate the same result as simulating the same applications in ISS or TurboXim.

Table I compares the simulation run time in the single processor scenario. All simulations utilize one process and one thread. All benchmarks are manually ported to the Xtensa processors and heavily utilize the embedded processor extensions. The number of added custom instructions and custom states are specified in the last two columns of the table.

`MP3MCH decoder`, `AC3 decoder`, and `MS10 decoder` are three audio codecs ported to the Xtensa processors with audio extensions. The Cstubs achieve an average speedup of 286× over the ISS and 3.2× over the TurboXim. `OSPF`, `IP_REASSEMBLY`, `ROUTELOOKUP`, `NAT`, `QOS`, and `TCP` are obtained from EEMBC [9] networking version 2.0 benchmark suite. They are frequently executed kernels in the networking domain. The processors are extended separately to speed up each kernel. The Cstubs achieve an average speedup of 5744× over the ISS and 45.8× over the TurboXim. The overall Cstubs speedup is 3924× over the ISS and 31.6× over the TurboXim. Higher run-time speedup is obtained on EEMBC networking benchmarks because the benchmarks contain self-checking code that does not take advantage of the processor extensions. In the audio codecs, however, the majority of the application run time is spent on the custom extensions. Custom extensions may have complicated semantics that need to be emulated whether TurboXim or Cstubs is used. Standard C code can be better optimized in Cstubs using the host compiler.

V. CONCLUSIONS

Fast simulation is essential for architectural exploration early in the design cycle and the validation of both the embedded processor extensions and the target applications late in the design cycle. In this paper, we present a tool to automatically create the functional models of embedded processor extensions suitable for native simulation. Our experiments demonstrate that natively executing the target applications with the functional models of the processor extensions can achieve orders of magnitude simulation speedup over both the cycle accurate models and the functional models of the entire processor.

REFERENCES

[1] *Xtensa™ microprocessor*. Tensilica Inc. (http://www.tensilica.com).

[2] *DesignWare™ ARC™ core*. Synopsys (http://www.synopsys.com).

[3] F. Pétrot, M. Gligor, M.-M. Hamayun, H. Shen, N. Fournel, and P. Gerin, "On MPSoC software execution at the transaction level," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 32–43, May 2011.

[4] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, "Hybrid simulation for energy estimation of embedded software," *IEEE Trans. Computer-Aided Design*, pp. 1843–1854, Oct. 2007.

[5] L. Gao, K. Karuri, Stefan, and Kraemer, "Multiprocessor performance estimation using hybrid simulation," in *Proc. Design Automation Conf.*, June 2008, pp. 325–330.

[6] P. Gerin, M. M. Hamayun, and F. Pétrot, "Native MPSoC co-simulation environment for software performance estimation," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Oct. 2009, pp. 403–412.

[7] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proc. Design Automation & Test Europe Conf.*, Mar. 2008, pp. 3–8.

[8] H. A. Sanghavi and N. B. Andrews, *Processor Description Languages, P. Mishra and N. Dutt (Editors)*. Morgan Kaufmann Publishers, Burlington, MA, 2008, ch. 8, pp. 183–216.

[9] *EEMBC*. EDN Embedded Microprocessor Benchmark Consortium Inc. (http://www.eembc.org).