Hybrid Simulation for Extensible Processor Cores

Jovana Jovic*, Sergey Yakoushkin*, Luis Murillo*, Juan Eusse*, Rainer Leupers* and Gerd Ascheid*
*Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Aachen, Germany
Email: jovic,yakoushkin,murillo,eusse,leupers@ice.rwth-aachen.de

***Abstract*—Due to their good flexibility-performance trade-off, Application Specific Instruction-set Processors (*ASIPs*) have been identified as a valuable component in modern embedded systems, especially the extensible ones, achieving good cost-efficiency trade-offs. Since the generation of the described hardware is usually automated to a high extent, in order to deliver an ASIP-based design in due time, developers are limited by the performance of the underlying simulation techniques for software development. On the other hand, the Hybrid Processor simulation technology (HySim), which enables dynamic run-time switching between native and instruction-accurate simulation, has reported high speed-up values for some fixed architectures. This paper presents enhanced HySim technology for extensible cores, based on a layered simulation infrastructure. This technology has shown a speed-up on a per-function basis of two orders of magnitude for a realistic MIMO OFDM benchmark on a multi-core platform with customized Xtensa cores by Tensilica.**

I. Introduction

Over the last decade, the high demand for multimedia and telecommunication products has caused a surge in interest for embedded systems. To keep up with the pace, vendors are seeking ever-increasing performance of such devices, that are to be released within shrinking time windows. This puts high expectations on productivity of designers. Application Specific Instruction-set Processors (ASIPs) have offered a good solution to these conflicting goals, providing both efficiency of dedicated solutions and flexibility of programmable devices. The usual practice assumes usage of simulation models of processors - Instruction Set Simulators (ISS) for complex software development, whose speed consequently determines the overall design time. There have been several proposals on how to overcome this speed issue. One of the notable techniques is HySim - Hybrid Processor Simulation [1], a dual mode simulation that allows switching between ISS and native execution at run time, thus providing to the SW developer a trade-off between timing accuracy and speed.

So far, HySim has been successfully applied to fixed architectures (RISC and VLIW) [2]. In order to extend its range of support to modern multimedia and wireless designs, we have looked into ways to enable the HySim technique to cover a conceptually different set of architectures, the *customizable* ASIPs, consisting of a pre-verified base core that can be enhanced in a number of ways to make the final design most efficient, according to its application domain. Hence, the HySim technology needs to account for late changes in the processor architecture. Several things need to be addressed: configurability - register file sizing, memory order or presence of a floating point unit; extensibility - special registers, instructions or functional units; synchronization - any changes on the global system view or processor state need to be visible in both simulators. As the trend in embedded system design is drifting towards multi-core solutions, the need for an upgrade in the available simulation techniques is becoming even more evident. Our proposal is the support for hybrid simulation of extensible cores, which gives the designer the freedom of selecting parts of simulation to be accelerated when exploring possible implementations for the system and consequently helps boosting his productivity.

The rest of the paper is organized as follows. Section II looks into some of the techniques proposed for accelerating the embedded system simulation. Section III gives an overview of the HySim technology and commercial extensible cores. Section IV outlines the challenges and proposed solutions for the application of HySim to extensible cores. Section V introduces the modifications to the instrumentation phase of HySim, which facilitate fast simulation of custom features of extensible cores. Section VI discusses the processor state synchronization on the example of Tensilica processors, and shows speed-up values on a realistic benchmark and platform, advocating for this technique. Finally, Section VII concludes our work and outlines some envisioned future improvements.

II. Related Work

The research in the domain of fast simulation techniques for complex embedded systems has been very lively. However, unlike HySim, most of the techniques target fast simulation of a known set of benchmarks for the purpose of architectural exploration. An example is SMARTS [3], which relies on statistical sampling in an attempt to estimate a cumulative performance metric (most relevantly Instruction Per Cycle (IPC)). Similarly, SimPoint [4] is a simulation framework which allows simulation of only the *representative* samples of the code. They are selected such to cover all the application phases exactly once. Phases are determined by basic block characterization, through profiling for execution frequency. Finally, Muttreja *et.al.* [5] proposed a hybrid single-core simulation technique for SW energy estimation, but the proposed implementation is intrusive, assuming some source code modifications for support of simulation mode switching.

III. Background

This section gives a brief overview of the two domains which our approach of hybrid simulation for extensible cores tries to bring together.

A. The HySim Concept

Being at a level high enough to provide high simulation speeds, yet capturing the features of the target architecture, the solution proposed by Kraemer *et. al.* [1] is designed to assist embedded SW simulation and debugging. The key idea behind is partial execution of code on the host, while keeping the simulation model updated at all times of interest. The switching is enabled by a central control module, and is open to user's interaction. The rationale is that the designer will

978-3-9810801-8-6/DATE12/©2012 EDAA

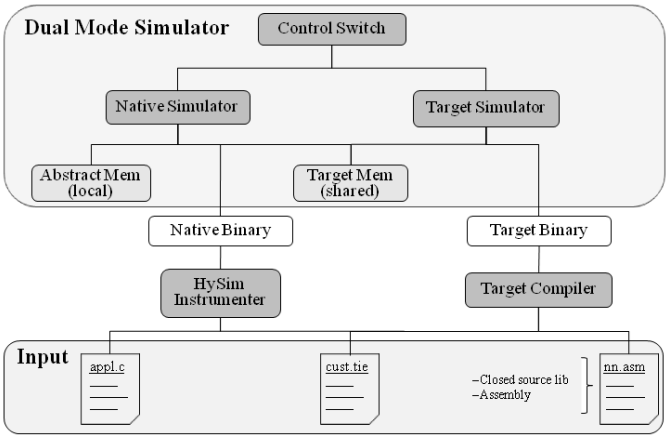


Fig. 1: HySim infrastructure.

typically know which parts of the code do not need to be observed in detail, and can be executed as fast as possible. The switching between the ISS, named *Target Simulator (TS)* and the *Abstract Simulator (AS)* is done at function borders.

One of the key premises of a simulator is consistency. In order to keep the processor state valid in both simulators, certain communication between them is needed, mainly for synchronization between the registers or global variables, i.e. updates of correct memory locations. To make it possible, a preparatory phase has to be gone through, before the actual simulation. The phase is called *application virtualization*, and is conducted by exposing the application source code to a tool called *HySim Instrumenter*. Fig. 1 outlines the components of the HySim simulation framework and their connections.

B. Extensible Processors

In response to the restricted usability of full custom design due to the high costs of silicon production and increasing significance of product differentiation in the expanding embedded market, a whole range of extensible processor solutions have emerged. One of them was proposed by Tensilica. The base technology of all Tensilica processors is the 32-bit RISC Xtensa architecture, customizable by a large set of features. In addition, leveraging the Verilog-based *Tensilica Instruction Extension (TIE)* language, the designers can introduce new instructions, by adding reusable high-level descriptions of datapaths, execution units, and register files. Furthermore, designers can specify new processor states, accessible by TIE instructions [6]. Tensilica provides an automated way for obtaining two flavors of simulation models, cycle- and instruction-accurate. At the other extreme, Tensilica tools automatically translate the TIE description into a native C implementation, thus enabling a fast functional simulation, for checking the functional correctness of the software relying on the custom instructions. The HySim technology targets bridging exactly this gap, between detailed simulation for HW verification and purely functional simulation.

Further examples include DesignWare ARC IP [7], but as opposed to the solution of Tensilica, integration of custom instructions happens on a much lower level. The designer has to write Verilog or SystemC description of the new instruction, which is integrated into the main processor pipeline. Silicon Hive offers a template-based technology, with the ultra-long instruction-word (ULIW) architecture as the base [8], configurable at a finer granularity of logic blocks called *processing and storage elements (PSEs)*. Finally, in contrast to ARC and Tensilica, customization of MIPS cores allows no configurability of the base core, but only extensions by User Defined Instructions (UDIs) which execute in parallel with the MIPS integer pipeline [9].

IV. Processor Extensibility and HySim

This section describes the HySim infrastructure, which has been designed to cope with the challenges of fast simulation of extensible cores. Two aspects of the HySim functionality can be clearly distinguished. On one hand, there must be a central switching control logic. On the other, the presence of a communication link between the target and the abstract simulators is needed. The major conceptual difference observed between the two is the target-architecture independence of the former one, as opposed to the architecture dependence of the latter. Therefore, we propose a *layered simulation infrastructure*, to ease the retargetability of the HySim technology for customizable architectures.

The architecture-independent layer, called *HySim Device*, performs processing of user commands, such as loading executables for abstract and target simulator, mapping/unmapping certain functions for fast execution and sensing when a switching point is triggered in order to pass the control to the abstract simulator. The architecture-dependent component is called the *HySim Wrapper*. It contains the implementation of a set of HySim API functions used for interfacing HySim with the ISS, responsible for preparing the input for the abstract simulator, i.e. reading and properly interpreting function arguments, updating of global variable changes, writing back the return values or fetching and resetting the return address to proceed with ISS execution on completion of the function in the abstract mode. These functions leverage the API functions of the processor simulation model for accessing the registers and memories and provide the functionality needed for the abstract simulator. Pseudo-code for a subset of the HySim API functions is to be found on the right-hand side of Fig. 2.

V. Function Virtualization for Extensible Cores

As outlined in Fig. 1, the HySim framework implies two different compilation flows. The application code usually has a subset of functions which have strictly target-dependent behavior and are therefore not suited for native simulation, like those including inline assembly. Therefore, we can define two function domains, those suitable for virtualization and those that are not. Since the HySim technology provides runtime switching between them, the domains have to be kept synchronized at all times. Functions in the first domain are compiled by the unmodified toolchain of the target processor. For the second domain, in order to ensure data consistency between the ISS and the host, HySim assumes a two-level approach: application transformation at compile time and interfacing to the ISS through the HySim Wrapper. The transformation process is called *virtualization* and is performed by the *HySim Instrumenter*. The Instrumenter virtualizes all suitable functions at compile-time, but this only means there is a possibility of fast execution of a certain function. The actual mapping of functions to abstract or target mode is defined through user-interface.

The HySim Instrumenter relies on the *Low Level Virtual Machine (LLVM)* compiler [10], whose modular design allows easy plugging of external IR(intermediate representation) transformations. The major advantage of the instrumentation on IR level lies in drastic reduction in instrumentation overhead, by avoiding unnecessary memory accesses, which affect

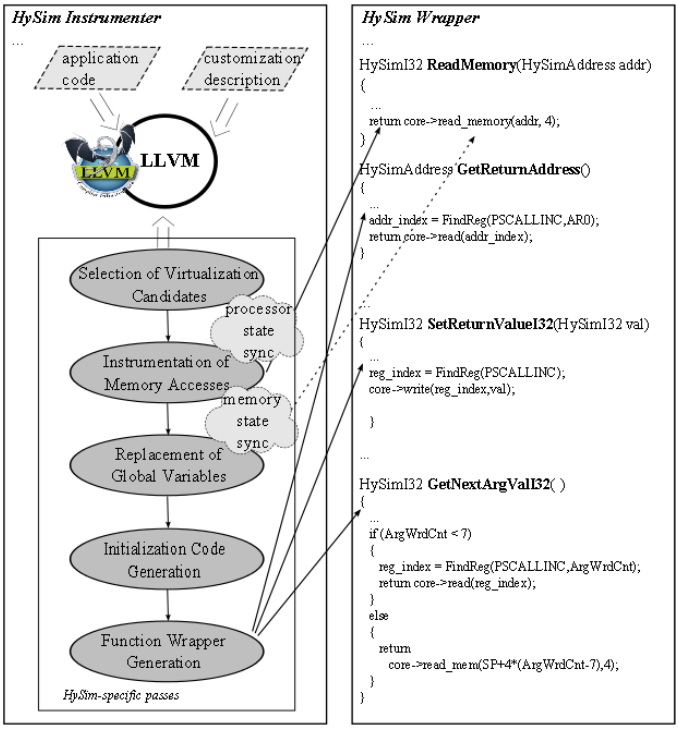


Fig. 2: HySim Instrumenter.

simulation performance greatly. For example, constant propagation and dead code elimination remove redundant HySim API calls for memory reads and writes. Function virtualization for HySim comprises several passes, which are sketched in Fig. 2. During virtualization the spots where synchronization is needed are detected, and links to the matching HySim API functions are inserted. The actual realization of the resource updates is performed by the HySim Wrapper.

A major challenge for the virtualization is the emulation of the functionality imposed by the processor extensions. The custom instructions are usually exposed to the C language as intrinsic functions to the target compiler. In order to simulate them natively, a matching implementation of intrinsics is also needed for the host compiler. Xtensa tools can automatically generate the so called *c-stubs*, a C implementation of the TIE intrinsics. If the toolsuite does not provide them automatically, the emulation functions have to be written manually. In order to illustrate the virtualization process of code which uses custom extensions, TIE code of a complex addition imple-

```
operation ADD_SCPLX32 { in AR op2_real, in AR op2_imag}
     {inout op1_real, inout op1_imag}
{
  //Declare wires to hold results
  wire [32:0] sum_real, sum_imag;
  //Do the actual addition
  assign sum_real = TIEadd(op1_real, op2_real, 1'b0);
  assign sum_imag = TIEadd(op1_imag, op2_imag, 1'b0);
  //Detect overflow, truncate accordingly and assign the
       output
  assign op1_real =
       ((op1_real[31])&(op2_real[31])&(~sum_real[31]))?
       {1'b1,sum_real[31:1]} :
        ((~op1_real[31])&(~op2_real[31])&(sum_real[31])) ?
             {1'b0,sum_real[31:1]} :
         sum_real[31:0];
  assign op1_imag = ...
}
```

Listing 1: TIE language implementation of complex addition operation.

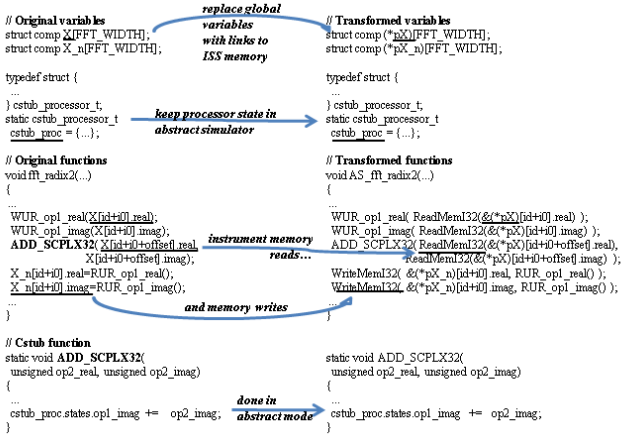


Fig. 3: Virtualization of custom extensions.

mentation is used, which can be found in Listing 1. In this example custom registers (`op1_real` and `op1_imag`) were used to pass one of the complex arguments and store the result. The pseudo code of the original application fragment, C-stub for the complex addition operation and the final virtualized function are outlined in Fig. 3. Processor resources can be separately modeled in abstract and in target mode. In abstract mode, emulation functions model processor state as a global variable. Since an equivalent variable does not exist in target mode, as it is an actual physical resource in the ISS, HySim does not replace this variable by the link to the target memory. This difference can be observed in the example: `X` is a global variable with a link to the target memory, hence in order to access it, the Instrumenter always generates appropriate HySim API calls; variable `op2_imag` exists only as a field in the structure used for the emulation of custom registers. As such, it will be treated as a private variable in the abstract simulator, therefore no synchronization with the ISS is needed.

VI. Case Study

The HySim for extensible processor cores has been evaluated on the example of Xtensa processor cores of Tensilica. This section presents several interesting characteristics of this implementation and the obtained speed-up results.

A. Processor State Synchronization for Xtensa Cores

Since control switching points are always at function borders, the architectural properties of interest for HySim are those that affect a C function call. All Xtensa-based processors support feature called *Windowed Registers*. This option enables general purpose register reuse, and hence reduces code size and increases performance due to a reduced number of register saves and restores. The register window is rotated on a subroutine call, by an index increment defined by the type of call instruction. This means that the location of the function arguments and return value has to be computed accordingly in the HySim Wrapper. If the word count exceeds 6, the value must be popped from the stack. Moreover, a set of HySim API functions for memory interfacing is foreseen for manipulation of the global variables and local pointers. For each data type there is a separate API function, so that the bits are interpreted properly and data of the correct size is read or written.

By using the TIE language, the designer can define additional register files and use them as operands of TIE instructions. The TIE compiler automatically generates a new data type. This data type has can be used in C/C++ programs as the

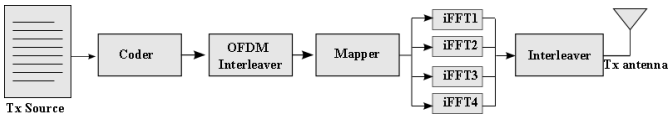
type of variables that reside in the register file. On top of that, further data types, associated with the same custom register file, can be defined. This feature is very common in data processing, for definition of domain specific data types, like 24-bits for audio applications. If a function in the application code has parameters of a custom type, the HySim Wrapper has to retrieve them from a different location (standard types go to the basic register file). The TIE register file name can be retrieved during instrumentation by parsing the TIE file.

B. Experimental Results

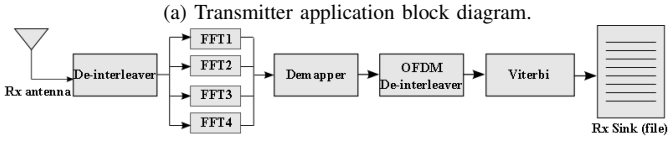
This section presents results of HySim for extensible cores deployed on MIMO-OFDM transceiver application. Block diagrams representing the transmitter and receiver application stages can be seen in Fig. 4. The boxes represent tasks, which are communicating through buffers. In order to achieve realistic speed-up measurements, the experimental framework was based on a full system simulation. To that end we used the Synopsys Platform Architect for building a clustered multi-core platfrom, comprising one control core which schedules tasks between two further DSP cores, used for offloading the intensive computations. The baseline processor was Xtensa XRC_D2MR. Experiments were conducted on a PC with Intel(R) Core2 Quad CPU Q6700 processor, running at 2.66GHz with 4096 KB of cache and 16 GB of RAM memory.

The software stack of the described application comprises several layers. The bottom layer provides low-level drivers through Xtensa Hardware Abstraction Layer, managed by intermediate one comprising the Xtensa operating system (xtos) running on the control core and a simplified in-house OS running on the DSPs, responsible for task scheduling and inter-core communication. Finally, the top application layer is reserved for computational task execution in case of DSPs, and global interrupt-based scheduling and traffic management in case of the control core. Analysis of the described application has shown that significant speed-up gains can be obtained by accelerating the FFT/IFFT calculations, that perform complex arithmetic operations. Therefore, those operations were described by the TIE language.

Irrespective of the simulation technology, the maximal speed is limited by the percentage of accelerated hot-spots. The FFT/IFFT computations consumed approximately one third of the total execution time. The total application speed-up measured was up to 7 times. In our experiments we measured maximal achievable simulation speed-up stemming from the custom blocks. The baseline is pure instruction-accurate ISS simulation. The speed-up number is given in two flavors: for the basic and customized implementation of the OFDM application with TIE instructions. The observed speed-up values are listed in Table I. The simulation times and speed-up numbers are given on a function basis, and include also the switching



(a) Transmitter application block diagram.



(b) Receiver application block diagram.

Fig. 4: OFDM modulation phases.

TABLE I: Simulation speed up results

Function	ISS Time [s]	HySim time [s]	Speed up [x]
FFT	2.584	0.024	107.67
IFFT	2.665	0.023	115.87
FFT-TIE	0.485	0.0012	404.2
IFFT-TIE	0.573	0.0013	440.8

overhead of the HySim control. We observed a speed-up of two orders of magnitude when applying HySim technology for extensible cores. A point worth noticing is the difference in the native execution time of the FFT/IFFT functions in base and customized case. The intrinsics generated by Tensilica tools are more efficient than the original implementation and therefore yield an even higher speed-up with HySim. When evaluating the HySim speed-up values it is worth emphasizing that Tensilica's simulation models use advanced techniques like just-in-time compiled-code simulation [11], hence the baseline simulator is in effect already optimized.

VII. Conclusion

In this paper we have presented a hybrid simulation framework for extensible cores, which offers the SW developers the possibility of run-time accuracy-speed trade-off. The proposed modifications in the HySim Instrumenter enable fast simulation of custom instructions, and, in the case of Xtensa cores, without any additional effort from the designer required. The speed-up values achieved for the base technology are reaching two orders of magnitude, on per function scale. As a next step, it would be interesting to look into support for some more advanced features of extensible processors, such as extensions for VLIW architectures, and a form of automatic generation of HySim Wrappers based on a description of custom features.

Acknowledgment

This work has been supported by the UMIC Research Center and Huawei Technologies. The authors would especially like to thank Yao Zhiliang and Guo Can from Huawei for the valuable discussions that guided this project.

References

[1] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr, "HySim: a fast simulation framework for embedded software development," in *Proc. of CODES+ISSS*, 2007.

[2] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "A fast and generic hybrid simulation approach using C virtual machine," in *Proc. of CASES*, 2007.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Statistical sampling of microarchitecture simulation," *ACM Trans. Model. Comput. Simul.*, vol. 16, pp. 197–224, July 2006.

[4] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. of ASPLOS-X '02*.

[5] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha, "Hybrid simulation for energy estimation of embedded software," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 10, pp. 1843 –1854, oct. 2007.

[6] R. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60 –70, mar/apr 2000.

[7] DesignWare ARC Configurable Cores. [Online]. Available: http://www.synopsys.com/IP/ConfigurableCores

[8] T. Halfhill, "Silicon Hive breaks out," *Microprocessor Report*, Dec. '03.

[9] Extensible MIPS cores. [Online]. Available: http://www.mips.com/products/cores/32-64-bit-cores/pro-series-family

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] G. Martin, N. Nedeljkovic, and D. Heine, "Configurable, extensible processor system simulation," in *Processor and System-on-Chip Simulation*, R. Leupers and O. Temam, Eds. Springer US, 2010, pp. 293–308.