

Design Space Exploration Using Arithmetic-Level Hardware–Software Cosimulation for Configurable Multiprocessor Platforms

JINGZHAO OU

Xilinx, Inc.

and

VIKTOR K. PRASANNA

University of Southern California

Configurable multiprocessor platforms consist of multiple soft processors configured on FPGA devices. They have become an attractive choice for implementing many computing applications. In addition to the various ways of distributing software execution among the multiple soft processors, the application designer can customize soft processors and the connections between them in order to improve the performance of the applications running on the multiprocessor platform. State-of-the-art design tools rely on low-level simulation to explore the various design trade-offs offered by configurable multiprocessor platforms. These low-level simulation based exploration techniques are too time-consuming and can be a major bottleneck to efficient design space exploration on these platforms. We propose a design space exploration technique for configurable multiprocessor platforms using arithmetic-level cycle-accurate hardware–software cosimulation. Arithmetic-level abstractions of the hardware and software execution platforms are created within the proposed cosimulation environment. The configurable multiprocessor platforms are described using these arithmetic-level abstractions. Hardware and software simulators are tightly integrated to concurrently simulate the arithmetic behavior of the multiprocessor platform. The simulation within the integrated simulators are synchronized to provide cycle-accurate simulation results for the complete multiprocessor platform. By doing so, we significantly speed up the cosimulation process for configurable multiprocessor platforms. Exploration of the various hardware–software design trade-offs provided by configurable multiprocessor platforms can be performed within the proposed cycle-accurate cosimulation environment. After the final designs are identified, the corresponding low-level implementations with the desired cycle-accurate arithmetic behavior are generated automatically. For illustrative purposes, we provide an implementation of our approach based on MATLAB/Simulink. We show the cosimulation of two numerical computation applications and one image-processing application on a popular configurable multiprocessor platform within the

Authors' addresses: J. Ou, DSP Design Tools and Methodologies, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124; email: jingzhao.ou@xilinx.com; V. K. Prasanna, Department of Electrical Engineering, University of Southern California, 3740 McClintock Avenue, EEB 200C, Los Angeles, CA 90089; email: prasanna.usc.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2006 ACM 1539-9087/06/0500-0355 \$5.00

MATLAB/Simulink-based cosimulation environment. For these three applications, our arithmetic-level cosimulation approach leads to speed-ups in simulation time of up to more than 800x compared with the low-level simulation approaches. The designs of these applications identified using our arithmetic-level cosimulation approach achieve execution time speed-ups up to 5.6x, compared with other designs considered in our experiments.

Categories and Subject Descriptors: B.7.1 [**Hardware**]: Integrated Circuits—*Types and design styles*; B.8.2 [**Hardware**]: Performance and Reliability—*Performance analysis and design aids*; J.6 [**Computer Applications**]: Computer-aided Engineering—*Computer-aided design (CAD)*

General Terms: Design, Performance

Additional Key Words and Phrases: FPGA, design space exploration, cosimulation, processor

1. INTRODUCTION

Integrated with multimillion gate configurable logic and various heterogeneous hardware components (such as embedded multipliers, memory blocks, etc.), FPGAs (field programmable gate arrays) offer high computational capabilities and have become an attractive choice for implementing many computing systems. FPGA configured soft processors, which are RISC processors realized using configurable resources available on FPGA devices, have also become popular. Examples of such soft processors include Nios from Altera, Inc., OpenRISC from OpenCores [Lampret et al.], LEON3 from Gaisler Research, Inc., and MicroBlaze from Xilinx, Inc.

Recently, there has been a trend in using configurable multiprocessor platforms on FPGA devices for application development. These multiprocessor platforms consist of multiple FPGA configured soft processors working together in a cooperative manner (e.g., the bus topology multiprocessor platform employed by [James-Roxby et al. 2004] and the 2-D mesh multiprocessor platform from CrossBow Technologies [Galicki 2003]). There are several advantages offered by configurable multiprocessor platforms for application development. One advantage is the reuse of legacy code, which can greatly reduce the development cycle. There are many existing C programs for realizing various computations and for coordinating the computations performed by the multiple processors. These C programs can be easily ported to configurable multiprocessor platforms with little or no changes. Another advantage is that multiprocessor platforms implemented using FPGAs are “configurable” and highly extensible by allowing the attachment of customized hardware peripherals. In addition to distributing the software execution among the multiple processors, the application designer can customize the hardware platform and optimize its performance for the applications running on them. Two major types of customization can be applied to configurable multiprocessor platforms for performance optimization. (1) *Attachment of hardware accelerators*: The application designer can optimize the performance of a single soft processor by adding dedicated instructions and/or attaching customized hardware peripherals to it. These dedicated instructions and customized hardware peripherals are used as hardware accelerators to speed up the execution of the time-consuming portions of the target application (e.g., the computation of Fast Fourier Transform and Discrete Cosine Transform). The Nios processor allows users to customize up to five instructions.

The MicroBlaze processor and the LEON3 processor support various dedicated interfaces and bus protocols for attaching customized hardware peripherals to them. The LEON3 processor even allows the application designer to have fine control of the cache organization. Cong et al. [2005] show that adding customized instructions to the Nios processor using a shadow register technique results in an average 2.75x execution time speed-up for several data-intensive digital signal-processing applications. By adding customized hardware peripherals, the FPGA-configured VLIW processor proposed by [Jones et al. 2005] also achieves an average 12x speed-up for several digital signal-processing applications from the MediaBench benchmark [Lee et al. 1997]. (2) *Efficient communication and execution among multiple processors*: The application designer can instantiate multiple soft processors, connect them with some specific topologies and communication protocols, and distribute the computation of the target application among these processors. James-Roxby et al. [2004] have developed a configurable multiprocessor platform for executing a JPEG2000 encoding application [Cook and Delp 1995]. They construct the multiprocessor platform by instantiating multiple MicroBlaze processors and connecting them with a bus topology. They then employ a single-program-multiple-data (SPMD) programming model to distribute the processing of the input image data among the multiple MicroBlaze processors. Their experimental results [James-Roxby et al. 2004] show that a significant speed-up is achieved for the JPEG2000 application on the multiprocessor platform compared with the execution on a single processor. Jin et al. [2005] built a configurable multiprocessor platform using multiple MicroBlaze processors arranged as multiple pipelined arrays for performing IPv4 packet forwarding. Their multiprocessor platform achieves 2x time performance improvement compared with a state-of-the-art network processor.

Paradoxically, while FPGA-based configurable multiprocessor platforms offer a high degree of flexibility for application development, performing design space exploration on configurable multiprocessor platforms is very challenging. State-of-the-art design tools rely on *low-level simulation*, which is based on the register transfer/gate-level implementations of the platform, for design space exploration. These low-level simulation techniques are inefficient for exploring the various hardware and software design trade-offs offered by configurable multiprocessor platforms. This is because of two major reasons. One is that low-level simulation based on register transfer/gate-level implementations is too time-consuming for evaluating the various possible configurations of the multiprocessor platform and the various possible hardware/software partitions of the target applications. Low-level simulation is especially inefficient for simulating the execution of software programs on configurable multiprocessor platforms. Considering the design examples shown in Section 4.1.2, low-level simulation using ModelSim [Mentor Graphics, Inc.] takes more than 25 min to simulate a software program running on the MicroBlaze processor with a 1.5 m execution time. This simulation speed can be overwhelming for development on configurable multiprocessor platforms as many software programs usually takes minutes or hours to complete on soft processors. The other reason is that the various optimization possibilities offered by configurable multiprocessor platforms provide a potentially large design space for application development. There

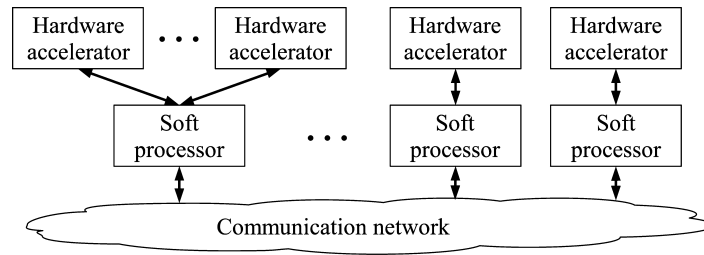


Fig. 1. Hardware architecture of the configurable multiprocessor platform.

are various hardware/software partitions of the target application and various possible mappings of the application to the multiple processors. For hardware designs, there are many possible realizations of the customized hardware peripherals, depending on the algorithms, architectures, hardware bindings, etc., employed by these peripherals. The communication interfaces between various hardware components (e.g., the topology for connecting the processors, the communication protocols for exchanging data between the processors, and the customized hardware peripherals and between the processors) would also significantly complicate the design space exploration problem. Exploring such a large design space using time-consuming low-level simulation techniques becomes intractable.

We address the following design problem in this paper. The target application is composed of a number of tasks with data exchange and execution precedence between each other. Each of the task is mapped to one or more processors for execution. The hardware architecture of the configurable multiprocessor platform is shown in Figure 1. Application development on the multiprocessor platform involves both software designs and hardware designs. For software designs, the application designer can choose the mapping and scheduling of the tasks for distributing the software programs of the tasks among the processors, so as to process the input data in parallel. For hardware designs, there are two levels of customization that can be done to the hardware platform. On one hand, customized hardware peripherals can be attached to the soft processors as hardware accelerators to speed up some computation steps. Different bus interfaces can be used for exchanging data between the processors and their own hardware accelerators. On the other hand, the processors are connected through a communication network in order to cooperatively process the input data. Various topologies and communication protocols can be used when constructing the communication network. Based on the above assumptions, our objective is to build an environment for design space exploration on configurable multiprocessor platform that has the following desired properties. (1) The various hardware and software design possibilities offered by the multiprocessor platform can be described within the design environment. (2) For a specific realization of the application on the multiprocessor platform, the hardware execution (i.e., the execution within the customized hardware peripheral and the communication interfaces) and the software execution within the soft processors are simulated rapidly in a concurrent manner so as to facilitate the exploration of the various hardware-software design

flexibilities. (3) The results gathered during the hardware-software cosimulation process should facilitate the application designer to identify the candidate designs and diagnose the performance bottlenecks. Finally, after the candidate designs are identified through the cosimulation process, the corresponding low-level implementations with the desired functional behavior can be generated automatically.

As the main contribution of this paper, we demonstrate that a design space exploration approach based on arithmetic-level cycle-accurate hardware-software modeling and cosimulation can achieve the objective described above. As discussed in Section 3, on one hand, the arithmetic-level modeling and cosimulation technique does not involve the low-level register transfer/gate-level implementations of the multiprocessor platform and, thus, can greatly increase the simulation speed. It enables efficient exploration of the various design trade-offs offered by the configurable multiprocessor platform. On the other hand, maintaining cycle accuracy during the cosimulation process helps the application designer to identify performance bottlenecks and thus facilitate the identification of candidate designs with “good” performance. We consider performance metrics of time and hardware resource usage in this paper. It also facilitates the automatic generation of low-level implementations once the candidate designs are identified through the arithmetic-level cosimulation process. For illustrative purposes, we show an implementation of the proposed arithmetic-level cosimulation technique based on MATLAB/Simulink [MathWorks]. Through the design of three widely used numerical computation and image-processing applications, we show that the proposed cycle-accurate arithmetic-level cosimulation technique achieves speed-ups in simulation time up to more than 800x as compared to those achieved by low-level simulation techniques. For these three applications, the designs identified using our cosimulation environment achieve execution speed-ups up to more than 5.6x compared with other designs considered in our experiments. Finally, we have verified the low-level implementations generated from the arithmetic-level design on a commercial FPGA prototyping board.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes our approach for building an arithmetic-level cycle-accurate cosimulation environment. An implementation of the co-simulation environment based on MATLAB/Simulink is provided to illustrate the cosimulation process. The development of two signal processing applications and one image processing application is provided in Section 4 to demonstrate the effectiveness of our cosimulation approach. Finally, we conclude in Section 5.

2. RELATED WORK

Many previous techniques have been proposed for performing hardware-software codesign and cosimulation on FPGA-based hardware platforms. Hardware-software codesign and cosimulation on FPGAs using compiler optimization techniques are proposed by [Gupta et al. 2003], [Hall et al. 1999], [Palem et al. 2001], and [Plaks 2001]. A hardware-software codesign technique for a data-driven accelerator is proposed by Becker [Hartenstein and Becker

1997]. To our best knowledge, none of the prior work addresses the rapid design space exploration problem for configurable multiprocessor platforms.

State-of-the-art hardware/software cosimulation techniques can be roughly classified into four major categories, which are discussed as follows.

- *Techniques based on low-level simulation.* Since configurable multiprocessor platforms are configured using FPGA resources, the hardware–software cosimulation of these platforms can use low-level hardware simulators directly. This technique is used by several commercial integrated design environments (IDEs) for application development using configurable multiprocessor platforms. This includes SOPC Builder from Altera, Inc. and Embedded Development Kit (EDK) from Xilinx, Inc. When using these IDEs, the multiprocessor platform is described using a simple configuration script. These IDEs will then automatically generate the low-level implementations of the multiprocessor platform and the low-level simulation models based on the low-level implementations. Software programs are also compiled into binary executable files, which are then used to initialize the memory blocks in the low-level simulation models. Based on the low-level simulation models, low-level hardware simulators (e.g., [Mentor Graphics, Inc.]) can be used to simulate the behavior of the complete multiprocessor platform. From one standpoint, the simple configuration scripts used in these IDEs provide very limited capabilities for describing the two types of optimization possibilities offered by configurable multiprocessor platforms. From another standpoint, as we show in Section 4, such low-level simulation-based cosimulation approach is too time-consuming for simulating the various possibilities of application development on the multiprocessor platforms.
- *Techniques based on high-level languages.* One approach of performing hardware–software cosimulation is by adopting high-level languages such C/C++ and Java. When applying these techniques, the hardware–software cosimulation can be performed by compiling the designs using their high-level language compilers and running the executable files resulting from the compilation process. There are several commercial tools based on C/C++. Examples of such cosimulation techniques include Catapult C from Mentor Graphics and Impulse C, which is used by the CoDeveloper design tool from Impulse Accelerated Technology, Inc. In addition to supporting the standard ANSI/ISO C, both Catapult C and Impulse C provide language extensions for specifying hardware implementation properties. The application designer describes his/her designs using these extended C/C++ languages, compile the designs using standard C/C++ compilers, generate the binary executable files, and verify the functional behavior of the designs by analyzing the output of the executable files. To obtain the cycle-accurate functional behavior of the designs, the application designer still needs to generate the VHDL simulation models of the designs, and perform low-level simulation using cycle-accurate hardware simulators. The DK3 tool from Celoxica, Inc. supports development on configurable platforms using Handel-C [Chappell and Sullivan 2004] and SystemC [Open SystemC Initiative], extensions of C/C++ language. While Handel-C and SystemC allows for the description of hardware and software

designs at different abstraction levels. However, to make a design described using Handel-C or SystemC suitable for direct register transfer-level generation, the application designer needs to write his/her designs at nearly the same level of abstraction as handcrafted register transfer-level hardware implementations [McCloud 2004]. This would prevent efficient design space exploration for configurable multiprocessor platforms.

- *Techniques based on software synthesis.* For software synthesis-based hardware–software cosimulation techniques, the input software programs are synthesized into codesign finite-state machine (CFSM) models, which demonstrate the same functional behavior as that of the software programs. These CFSM models are then integrated into the simulation models for the hardware platform for hardware–software cosimulation. One example of the software synthesis approach is the POLIS hardware–software codesign framework [Balarin et al. 1997]. In POLIS, the input software programs are synthesized and translated into VHDL simulation models, which demonstrate the same functional behavior. The hardware simulation models are then integrated with the simulation models of other hardware components for cosimulation of the complete hardware platform. The software synthesis approach can greatly accelerate the time for cosimulating the low-level register transfer/gate level implementations of multiprocessor platforms. One issue with the software synthesis cosimulation approach is that it is difficult to synthesize complicated software programs (e.g., operating systems, video encoding/decoding software programs) into hardware simulation models with the same functional behavior. Another issue is that the software synthesis approach is based on low-level implementations of the multiprocessor platform. The application designer needs to generate the low-level implementations of the complete multiprocessor platform before he/she can perform cosimulation. The large amount of efforts required by generating the low-level implementations prevents efficient exploration of various configurations of the configurable multiprocessor platform.
- *Techniques based on integration of low-level simulators.* Another approach for hardware–software cosimulation is through the integration of low-level hardware–software simulators. One example is the *Seamless* tool from Mentor Graphics, Inc. It contains precompiled cycle-accurate behavioral processor models for simulating the execution of software programs on various processors. It also provides precompiled cycle-accurate bus models for simulating the communication between hardware and software executions. Using these precompiled processor models and bus models, the application designer can separate the simulation of hardware and software executions and use the corresponding hardware and software simulators for cosimulation. The *Seamless* tool provides detailed simulation information for verifying the correctness of hardware and software codesigns. However, it is still based on the time-consuming low-level simulation techniques and thus is inefficient for application development on configurable multiprocessor platforms.

To summarize, the design approaches discussed above rely on low-level simulation to obtain the cycle-accurate functional behavior of the hardware–software

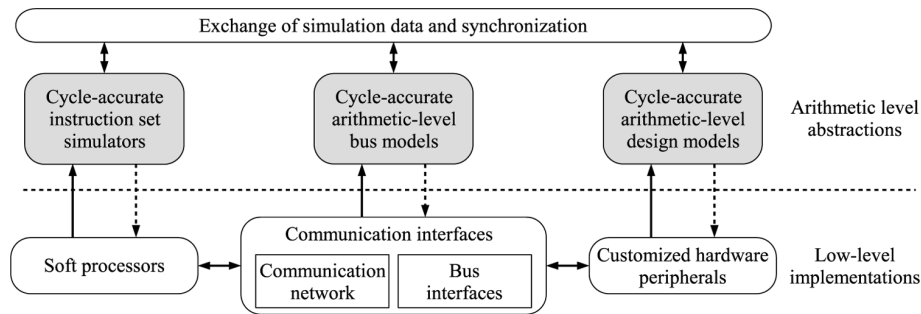


Fig. 2. Our approach for high-level hardware–software cosimulation.

execution on configurable multiprocessor platforms. Such reliance on low-level simulation models prevents them from efficiently exploring the various configurations of multiprocessor platforms. Compared with these approaches, the arithmetic-level cosimulation technique proposed in this paper allows for arithmetic-level cycle-accurate hardware–software cosimulation of the multiprocessor platform without involving the low-level implementations and low-level simulation models of the complete platform. Our approach is able to achieve a significant speed-up in simulation time while providing cycle-accurate simulation details as compared to these low-level simulation techniques.

3. OUR APPROACH

3.1 Basic Idea

Our approach for design space exploration is based on an arithmetic-level cosimulation technique. In the following paragraphs, we first present the arithmetic-level cosimulation technique. Design space exploration for application development on the configurable multiprocessor platform is performed by analyzing the information gathered during the cosimulation process.

- *Arithmetic-level modeling and cosimulation.* The arithmetic-level modeling and cosimulation technique for configurable multiprocessor platforms is shown in Figure 2. The configurable multiprocessor platform consists of three major components: *soft processors* for executing software programs; *customized hardware peripherals* as hardware accelerators for parallel execution of some specific computation steps; and *communication interfaces* for data exchange between various hardware components. The communication interfaces include *bus interfaces* for exchanging data between the processors and their customized hardware peripherals, and *communication networks* for coordinating the computations and communication among the soft processors. When employing our arithmetic-level cosimulation technique, arithmetic-level (“high-level”) abstractions are created to model each of the three major components. The arithmetic-level abstractions can greatly speed up the cosimulation process while allowing the application designer to explore the optimization opportunities provided by configurable multiprocessor platforms. By “arithmetic level”, we mean that only the arithmetic aspects

of the hardware and software execution are modeled by these arithmetic abstractions. Taking multiplication as an example, its low-level implementation on Xilinx Virtex-II/Virtex-II Pro FPGAs can be realized using either slice-based multipliers or embedded multipliers. Its arithmetic-level abstraction only capture the arithmetic property, i.e., multiplication of the values presented at its input ports. Taking the communication between different hardware components as another example, its low-level implementations can use registers (flip-flops), slices, or embedded memory blocks to realize data buffering. Its arithmetic-level abstraction only capture the arithmetic level data movements on the communication channels (e.g., the handshaking protocols, access priorities for communication through shared communication channels). Cosimulation based on the arithmetic-level abstractions of the hardware components does not involve the low-level implementation details. Using the arithmetic-level abstraction, the user can specify that the movement of the data be delayed for one clock cycle before it becomes available to the destination hardware components. He/she does not need to provide low-level implementation details (such as the number of flip-flop registers and the connections of the registers between the related hardware components) for realizing this arithmetic operation. Such low-level implementations can be generated automatically once the high-level design description is finished. Thus, these arithmetic-level abstractions can significantly speed up the time required to simulate the hardware and software arithmetic behavior of the multiprocessor platform. The implementation of the arithmetic level cosimulation technique presented in Section 3.2 demonstrates a simulation speed-up up to more than 800x compared with the behavioral simulation based on low-level implementations.

The configurable multiprocessor platform is described using the arithmetic-level abstractions. The arithmetic-level abstractions allow the application designer to specify the various ways of constructing the data paths through which the input data is processed. Cosimulation based on the arithmetic-level abstractions gives the status of the data paths during the execution of the application. For example, the development of a JPEG2000 application on a state-of-the-art configuration multiprocessor platform using the arithmetic-level abstractions is shown in Section 4.2. The arithmetic-level abstraction of the JPEG2000 application specifies the data paths that the input image data are processed among the multiple processors. The cosimulation based on the arithmetic-level abstraction gives the intermediate processing results at various hardware components that constitute the multiprocessor platform. Using these intermediate processing results, the application designer can analyze performance bottlenecks and identify the candidate designs of the multiprocessor platform. Application development using the arithmetic-level abstractions focus on the description of data path. Thus, the proposed arithmetic-level cosimulation technique is especially suitable for the development of data-intensive applications, such as signal- and image-processing applications.

The arithmetic-level abstractions of the configurable multiprocessor platform are simulated using their corresponding hardware and software

simulators. These hardware and software simulators are tightly integrated into our cosimulation environment and concurrently simulate the arithmetic behavior of the complete multiprocessor platform. Most importantly, the simulations performed within the integrated simulators are synchronized between each other at each clock cycle and provide cycle accurate simulation results for the complete multiprocessor platform. By “cycle-accurate”, we mean that for each clock cycle during the cosimulation process, the arithmetic behavior of the multiprocessor platform simulated by the proposed cosimulation environment matches with the arithmetic behavior of the corresponding low-level implementations. For example, when simulating the execution of software programs, the cycle-accurate cosimulation takes into account the number of clock cycles required by the soft processors for completing a specific instruction (e.g., the multiplication instruction of the MicroBlaze processor takes three clock cycles to complete). When simulating the hardware execution on customized hardware peripherals, the cycle-accurate cosimulation takes into account the number of clock cycles required by the pipelined customized hardware peripherals to process the input data. The cycle-accurate cosimulation process also takes into account the delays caused by the communication channels between the various hardware components. One approach for achieving such cycle accuracy between different simulators in the actual implementations is by maintaining a global simulation clock in the cosimulation environment. This global simulation clock can be used to specify the time at which a piece of data is available for processing by the next hardware or software component, based on the delays specified by the user. Maintaining such cycle-accurate property in the arithmetic-level cosimulation ensures that the results from the arithmetic level cosimulation are consistent with the arithmetic behavior of the corresponding low-level implementations. The cycle-accurate simulation results allow the application designer to observe the instant interactions between hardware and software executions. The instant interaction information is used in the design space exploration process for identifying performance bottlenecks in the designs.

- *Design space exploration.* Design space exploration is performed based on the arithmetic-level abstractions. The functional behavior of the candidate designs can be verified using the proposed arithmetic-level cosimulation technique. As shown in Section 4, the application designer can use the simulation results gathered during the cycle-accurate arithmetic-level cosimulation process to identify the candidate designs and diagnose the performance bottlenecks when performing design space exploration. Considering the development of a JPEG2000 application on a multiprocessor platform, discussed in Section 4.2, using the results from the arithmetic-level simulation can identify that the bus connecting the multiple processors limits the performance of the complete system when more processors are employed. Besides, these detailed simulation results also facilitate the automatic generation of the low-level implementations. By specifying the low-level hardware bindings for the arithmetic operations (e.g., binding the embedded multipliers for realization of the multiplication arithmetic operation), the application designer can also rapidly obtain the hardware resource usage for a specific realization of the

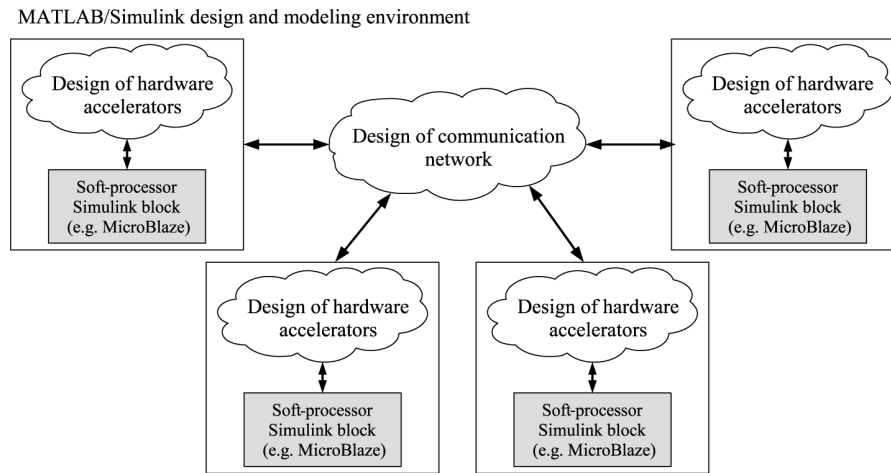


Fig. 3. An implementation of the proposed arithmetic cosimulation environment based on MATLAB/Simulink.

application [Shi et al. 2004]. The cycle-accurate arithmetic-level simulation results and the rapidly estimated hardware resource usage information can help the application designer to efficiently explore the various optimization opportunities and identify “good” candidate designs. For example, in the development of a block matrix multiplication algorithm shown in Section 4.1.1, the application designer can explore the impact of the size of the matrix blocks on the performance of the complete algorithm. Finally, for the designs identified by the arithmetic-level cosimulation process, low-level implementations with corresponding arithmetic behavior are automatically generated based on the arithmetic-level abstractions of the multiprocessor platform.

3.2 Implementation Details

For illustrative purposes, we provide an implementation of our arithmetic-level cosimulation approach based on MATLAB/Simulink for application development using configurable multiprocessor platforms. The software architecture of the implementation is shown in Figure 3 and 4. Arithmetic-level abstractions of the customized hardware peripherals and the communication interfaces (including the bus interfaces and the communication network) are created within the MATLAB/Simulink modeling environment. Thus, the hardware execution platform is described and simulated within MATLAB/Simulink. An addition, we create *soft-processor Simulink blocks* for integrating cycle-accurate instruction set simulators targeting the soft-processors. The execution of the software programs distributed among the soft processors is simulated using these soft-processor Simulink blocks.

Our arithmetic-level cosimulation environment consists of four major components: *simulation of software execution on soft processors*, *simulation of customized hardware peripherals*, *simulation of communication interfaces*, and *exchange of simulation data and synchronization*. They are discussed in detail in the following subsections.

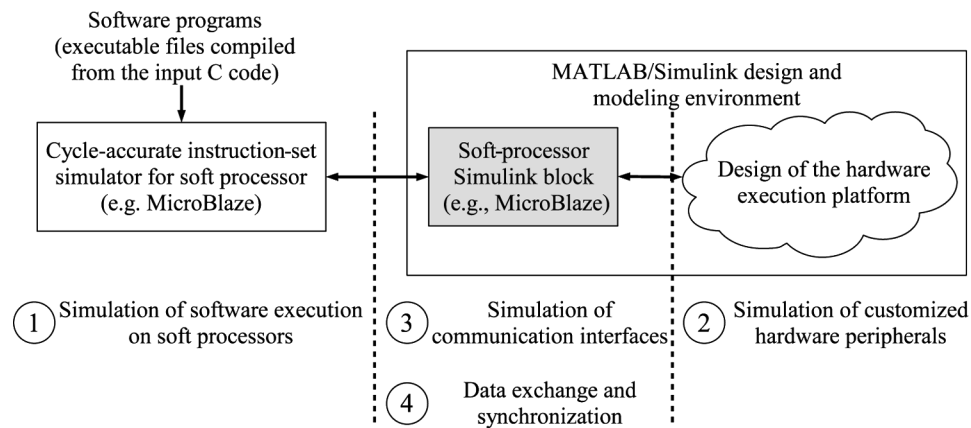


Fig. 4. Architecture of the soft-processor Simulink block.

3.2.1 Simulation of Software Execution on Soft Processors. Soft-processor Simulink blocks (e.g., MicroBlaze Simulink blocks targeting MicroBlaze processors) are created for simulating the software programs running on the processors. Each soft-processor Simulink block simulates the software programs executed on one processor. Multiple soft-processor Simulink blocks are employed when simulating the configurable multiprocessor platform.

The software architecture of a soft-processor Simulink block is shown Figure 4. The input C programs are compiled using the compiler for the specific processor (e.g., the GNU C compiler *mb-gcc* for MicroBlaze) and translated into binary executable files (e.g., *.ELF* files for MicroBlaze). These binary executable files are then simulated using a cycle-accurate instruction-set simulator for the specific processor. Taking the MicroBlaze processor as an example, the executable *.ELF* files are loaded into *mb-gdb*, the GNU C debugger for MicroBlaze. A cycle-accurate instruction-set simulator for the MicroBlaze processor is provided by Xilinx. *mb-gdb* sends instructions of the loaded executable files to the MicroBlaze instruction set simulator and performs cycle-accurate simulation of the execution of the software programs. *mb-gdb* also sends/receives commands and data to/from MATLAB/Simulink through the soft-processor Simulink block and interactively simulate the execution of the software programs in concurrence with the simulation of the hardware designs within MATLAB/Simulink.

3.2.2 Simulation of Customized Hardware Peripherals. The customized hardware peripherals are described using the MATLAB/Simulink-based FPGA design tools. For example, *System Generator* supplies a set of dedicated Simulink blocks for describing parallel hardware designs using FPGAs. These Simulink blocks provide arithmetic-level abstractions of the low-level hardware components. There are blocks that represent the basic hardware resources (e.g., flip-flop based registers, multiplexers), blocks that represent control logic, mathematical functions, and memory, and blocks that represent proprietary IP (intellectual property) cores (e.g., the IP cores for Fast Fourier Transform and finite impulse filters). Considering the *Mult* Simulink block for multiplication

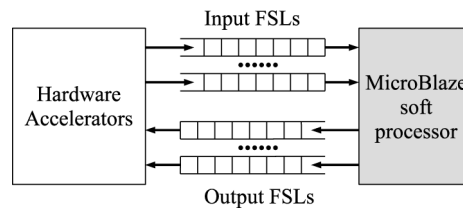


Fig. 5. Communication between MicroBlaze and customized hardware designs through Fast Simplex Links.

provided by *System Generator*, it captures the arithmetic behavior of multiplication by presenting at its output port the product of the values presented at its two input ports. Whether the low-level implementation of the *Mult* Simulink block is realized using embedded or slice-based multipliers is ignored in its arithmetic-level abstraction. The application designer assembles the customized hardware peripherals by dragging and dropping the blocks from the block set to his/her designs and connecting them via the Simulink graphic interface. Simulation of the customized hardware peripherals is performed within the MATLAB/Simulink. MATLAB/Simulink maintains a simulation timer for keeping track of the simulation process. Each unit of simulation time counted by the simulation timer corresponds to one clock cycle experienced by the corresponding low-level implementations.

3.2.3 Simulation of Communication Interfaces. As shown in Figure 3, the simulation of the communication interfaces is composed of two parts, simulation of the bus interfaces between the processors and their corresponding peripherals and simulation of the communication network between the processors, which are described as follows.

- *Simulation of the dedicated bus interfaces.* Simulation of the dedicated bus interfaces between the processors and their hardware peripherals is performed by the soft-processor Simulink block. Basically, the soft-processor Simulink blocks need to simulate the input/output communication protocols and the data buffering operations of the dedicated bus interfaces.

We use MicroBlaze processors and the dedicated Fast Simplex Link (FSL) bus interfaces to illustrate the cosimulation process. FSLs are unidirectional FIFO (First-In-First-Out) channels. Data can move in and out of the FSL channels in two clock cycles. As shown in Figure 5, a MicroBlaze processor provides eight FSL channels for data input and another eight channels for data output. Using these FSL channels, the application designer can attach customized hardware peripherals as hardware accelerators to the MicroBlaze processor. Both synchronous (blocking) and asynchronous (non-blocking) read/write operations are supported by MicroBlaze. For blocking read/write operations, the MicroBlaze processor is stalled until the read/write operations finish. For nonblocking read/write operations, MicroBlaze resumes its normal execution immediately regardless of the outcome of the read/write operations.

The MicroBlaze Simulink blocks simulate the FSL FIFO buffers, and the bus interfaces between the customized hardware peripherals and the FSL channels. The width of the FSL channels is 32 bit while their widths are configurable from 1 to 8192 depending on the application requirements and the available hardware resources on the FPGA device. Let “#” denote the ID of the FSL input/output channel for accessing the MicroBlaze processor. “#” is an integer number between 0 and 7. When the *In#_write* input port of the MicroBlaze Simulink block becomes high, it indicates that there is data from the customized hardware peripherals simulated in MATLAB/Simulink. The data will be written into the FSL FIFO buffer stored at the internal data structure of the MicroBlaze Simulink block. The MicroBlaze Simulink block would then store the MATLAB/Simulink data presented at the *In#_write* input port into the internal data structure and raises the *Out#_exists* output port stored in its internal data structure to indicate the availability of the data. Similarly, when the FSL FIFO buffer is full, the MicroBlaze Simulink block will raise the *In#_full* output port in MATLAB/Simulink to prevent further data coming into the FSL FIFO buffer.

The MicroBlaze Simulink blocks also simulate the bus interfaces between the MicroBlaze processor and the FSL channels. A set of dedicated C functions is provided for the MicroBlaze to control the communication through FSLs. After compilation using *mb-gcc*, these C functions are translated to the corresponding dedicated assembly instructions. For example, C function *microblaze_nbread_datafsl(val,id)* is used for nonblocking reading of data from the *id*-th FSL channel. This C function is translated into a dedicated assembly instruction *nget(*val*, rfsl#id)* during the compilation of the software program. By observing the execution of these dedicated C functions during cosimulation, we can control the hardware and software processes so as to correctly simulate the interactions between the processors and their corresponding hardware peripherals.

During the simulation of the software programs, the MicroBlaze Simulink block keeps track of the status of the MicroBlaze processor by communicating with *mb-gdb*. As soon as the dedicated assembly instruction described above for writing data to the customized hardware peripherals through the FSL channels is encountered by the *mb-gdb*, it informs the MicroBlaze Simulink block. The MicroBlaze Simulink block will then stall the simulation of software programs in *mb-gdb*, extract the data from *mb-gdb*, and try to write the data into the FSL FIFO buffer stored at its internal data structure. For communication in the blocking mode, the MicroBlaze Simulink block stalls simulation of the software programs in *mb-gdb* until the write operation is completed. That is, the simulation of software programs gets stalled until the *In#_full* flag bit stored at the MicroBlaze Simulink block internal data structure becomes low. This indicates that the FSL FIFO buffer is ready to accept more data. Otherwise, for communication in nonblocking mode, the MicroBlaze Simulink block resumes the simulation of the software programs immediately after writing data to the FSL FIFO buffer, regardless of the outcome of the write operation. Data exchange for the read operation is handled in a similar manner.

- *Simulation of the communication network.* Each soft processor and its customized hardware peripherals are developed as a *soft-processor subsystem* in MATLAB/Simulink. In order to provide the flexibility for exploring the different topologies and communication protocols for connecting the processors, the communication network that connects the *soft-processor subsystems* and coordinates the computations and communication between these subsystems is described and simulated within MATLAB/Simulink. As shown in the design example discussed in Section 4.2, an OPB (On-chip peripheral bus) Simulink block is created to realize the OPB shared bus interface. The multiple MicroBlaze *subsystems* are connected to the OPB Simulink block to form a bus topology. A hardware semaphore (i.e. a mutual exclusive access controller) is used to coordinate the hardware–software execution among the multiple MicroBlaze processors. The hardware semaphore is described and simulated within MATLAB/Simulink.

3.2.4 *Exchange of simulation data and synchronization between the simulators.* The soft-processor Simulink blocks are responsible for exchanging simulation data between the software and hardware simulators. The input and output ports of the soft-processor Simulink blocks are used to separate the simulation of the software programs running on the soft processor and that of the other Simulink blocks, e.g., the hardware peripherals of the processor as well as other soft processors employed in the design. The input and output ports of the soft-processor Simulink blocks correspond to the input and output ports of the low-level hardware implementations. For low-level ports that are both input and output ports, they are represented as separate input and output blocks suffixed with port names “_in” and “_out,” respectively, on the Simulink blocks. The MicroBlaze Simulink blocks send the values of the FSL registers at the MicroBlaze instruction set simulator to the input ports of the soft-processor Simulink blocks as input data for the hardware peripherals. In the reverse, the MicroBlaze Simulink blocks collect the simulation output of the hardware peripherals from the output ports of the soft-processor Simulink blocks and use the output data to update the values of the FSL registers stored at its internal data structure.

When exchanging the simulation data between the simulators, the soft-processor Simulink blocks take into account the number of clock cycles required by the processors and the customized hardware peripherals to process the input data. They also take into account the delays caused by transmitting the data through the dedicated bus interfaces and the communication network. By doing so, the hardware and software simulation are synchronized on a cycle-accurate basis.

Moreover, a global simulation timer is used to keep track of the simulation time of the complete multiprocessor platform. All hardware and software simulations are synchronized with this global simulation timer. For the MATLAB/Simulink-based implementation of the cosimulation environment, one unit of simulation time, counted by the global simulation timer, equals one unit of simulation time within MATLAB/Simulink and one clock cycle simulation time of the MicroBlaze instruction-set simulator. It is ensured that one unit

of the simulation time counted by the global simulation timer also corresponds to one clock cycle experienced by the corresponding low-level implementations of the multiprocessor platform.

3.3 Rapid Hardware Resource Estimation

Being able to rapidly obtain the hardware resources occupied by various configurations of the multiprocessor platform is required for design space exploration. For Xilinx FPGAs, we focus on the number of slices, the number of BRAM memory blocks, and embedded 18-bit-by-18-bit multipliers used for constructing the multiprocessor platform.

For the multiprocessor platform based on MicroBlaze processors, the hardware resources are used by the following four types of hardware components: the MicroBlaze processors, the customized hardware peripherals, the communication interfaces (the dedicated bus interfaces and the communication network), and the storage of the software programs. Resource usage of the MicroBlaze processors, the two LMB (local memory bus) interface controllers and the dedicated bus interfaces is estimated from the Xilinx data sheet. Resource usage of the customized hardware designs and the communication network is estimated using the resource estimation technique provided by [Shi et al. 2004]. Since the software programs are stored in BRAMs, we obtain the size of the software programs using the *mb-objdump* tool and then calculate the numbers of BRAMs required to store these software programs. The resource usage of the multiprocessor platform is obtained by summing the hardware resources used by the above four types of hardware components.

4. ILLUSTRATIVE EXAMPLES

To demonstrate the effectiveness of our approach, we show in this section the development of two widely used numerical computation applications (i.e., CORDIC algorithm for division and block matrix multiplication) and one image-processing application (i.e., JPEG2000 encoding) on a popular configurable multiprocessor platform. The two numerical computation applications are widely deployed in systems such as radar systems and software-defined radio systems [Lee et al. 1997]. Implementing these applications using soft processors provides the capability of handling different problem sizes depending on the specific application requirements.

We first demonstrate the cosimulation process of an individual soft processor and its customized hardware peripherals. We then show the cosimulation process of the complete multiprocessor platform, which is constructed by connecting the individual soft processors with customized hardware peripherals through a communication network. Our illustrative examples focus on the MicroBlaze processors and the FPGA design tools from Xilinx because of their wide availability. Our cosimulation approach is also applicable to other soft processors and FPGA design tools. Virtex-II Pro FPGAs [Xilinx, Inc.] are chosen as our target devices. Arithmetic-level abstractions of the hardware execution platform are provided by *System Generator 8.1EA2*. Automatic generation of the low-level implementations of the multiprocessor platform is realized using

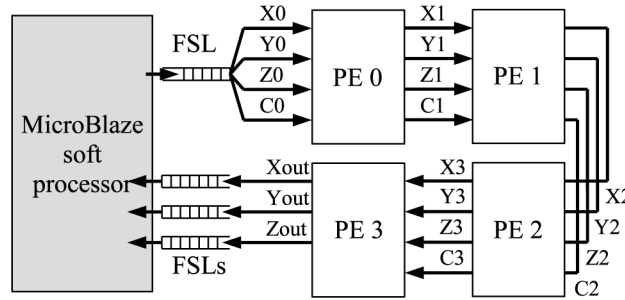


Fig. 6. CORDIC algorithm for division with $P = 4$.

both *System Generator 8.1EA2* and EDK (Embedded Development Kit) 7.1 ISE (Integrated Software Environment) 7.1 [Xilinx, Inc.] is used for synthesis and implementation (including placing-and-routing) of the complete multiprocessor platform. Finally, the functional correctness of the multiprocessor platform is verified using an ML300 Virtex-II Pro prototyping board [Xilinx, Inc.].

4.1 Cosimulation of the Processor and the Hardware Peripherals

4.1.1 Adaptive CORDIC Algorithm for Division. The CORDIC (COordinate Rotation DIgital Computer) iterative algorithm for dividing b by a [Andraka 1998] is described as follows. Initially, we set $X_{-1} = a$, $Y_{-1} = b$, $Z_{-1} = 0$, and $C_{-1} = 1$. Let N denote the number of iterations performed by the CORDIC algorithms. During each iteration i ($i = 0, 1, \dots, N-1$), the following computation is performed.

$$\begin{cases} X_i = X_{i-1} \\ Y_i = Y_{i-1} + d_i \cdot X_{i-1} \cdot C_{i-1} \\ Z_i = Z_{i-1} - d_i \cdot C_{i-1} \\ C_i = C_{i-1} \cdot 2^{-1} \end{cases} \quad (1)$$

where, $d_i = +1$ if $Y_i < 0$ and $d_i = -1$ otherwise. After N iterations of processing, we have $Z_N \approx -b/a$. Implementing this CORDIC algorithm using soft processors not only leads to compact designs, but also offers dynamic adaptivity for practical application development. For example, while many telecommunication systems have a wide dynamic data range, it is desired that the number of iterations can be dynamically adapted to the environment where the telecommunication systems are deployed. For some CORDIC algorithms, the effective precision of the output data cannot also be computed analytically. One example is the hyperbolic CORDIC algorithms. The effective output bit precision of these algorithms depends on the angular value Z_i during iteration i and needs to be determined dynamically.

- *Implementation.* The hardware architecture of our CORDIC algorithm for division based on MicroBlaze is shown in Figure 6. The customized hardware peripheral is configured with P processor elements (PEs). Each PE performs one iteration of computation described in Eq. 1. All the PEs form a

linear pipeline. We consider 32-bit data precision in our designs. Since software programs are executed in a serial manner in the processor, only one FSL channel is used for sending the data from MicroBlaze to the customized hardware peripheral. The software program controls the number of iterations for each set of data based on the specific application requirement. To support more than four iterations for the configuration shown in Figure 6, the software program sends X_{out} , Y_{out} and Z_{out} generated by PE_3 back to PE_0 for further processing, until the desired number of iterations is reached.

For the processing elements shown in Figure 6, C_0 is provided by the software program based on the number of times that the input data has passed through the linear pipeline. C_0 is sent out from the MicroBlaze processor to the FSL as a control word. That is, when there is data available in the corresponding FSL FIFO buffer and $Out\#_control$ is high, PE_0 updates its local copy of C_0 and then continues to propagate it to the following PEs along the linear pipeline. For the other PEs, C_i is updated as $C_i = C_{i-1} \cdot 2^{-1}$, $i = 1, 2, \dots, P - 1$, and is obtained by right shifting C_{i-1} from the previous PE.

When performing division on a large set of data, the input data is divided into several sets. These sets are processed one by one. Within each set of data, the data samples are fed into the customized hardware peripheral consecutively in a back-to-back manner. The output data of the hardware peripheral is stored at the FIFO buffers of the data output FSLs and is further sent back to the processor. The application designer needs to select a proper size for each set of data so that the results generated would not overflow the FIFO buffers of the data output FSL channels.

- *Design space exploration.* We consider different implementations of the CORDIC algorithm with different P , the number of processing elements used for implementing the linear pipeline. When more processing elements are employed in the design, the execution of the CORDIC division algorithm can be accelerated. However, the configuration of the MicroBlaze processor would also consume more hardware resources.

The time performance of various configurations of the CORDIC algorithm for division is shown in Figure 7, while its resource usage is shown in Table I. The resource usage estimated using our design tool is calculated as shown in Section 3.3. The actual resource usage is obtained from the place-and-route reports (*.par* files) generated by ISE. For CORDIC algorithms with 24 iterations, attaching a customized linear pipeline of four PEs to the soft processor results in a 5.6x improvement in time performance compared with “pure” software implementation, while it requires 280 (30%) more slices.

- *Simulation speed-ups.* The simulation time of the CORDIC algorithm for division using our high-level cosimulation environment is shown Table I. For comparison purpose, we also show the simulation time of the low-level behavioral simulation using ModelSim. For the ModelSim simulation, the time for generating the low-level implementations is not accounted for. We only consider the time for compiling the VHDL simulation models and performing the low-level simulation within ModelSim. Compared with the

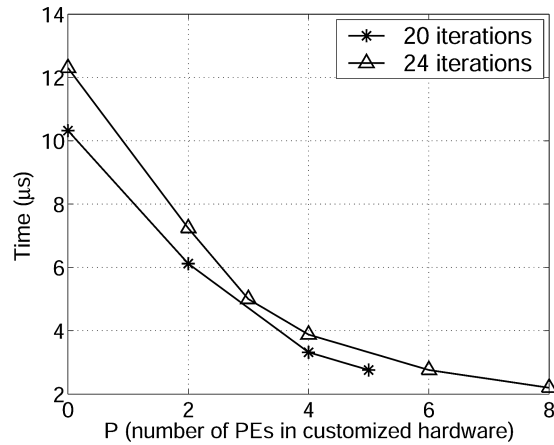


Fig. 7. Time performance of the CORDIC algorithm for division ($P = 0$ denotes “pure” software implementations).

Table I. Resource Usage of the CORDIC-Based Division and the Block Matrix Multiplication Applications and Time for Cycle-Accurate Functional Simulation Using Different Simulators

Designs	Estimated/actual resource usage		
	Slices	BRAMs	Multipliers
24 iteration CORDIC div. with $P = 2$	729/721	1/1	3/3
24 iteration CORDIC div. with $P = 4$	801/793	1/1	3/3
24 iteration CORDIC div. with $P = 6$	873/865	1/1	3/3
24 iteration CORDIC div. with $P = 8$	975/937	1/1	3/3
12×12 matrix mult. with 2×2 blocks	851/713	1/1	5/5
12×12 matrix mult. with 4×4 blocks	1043/867	1/1	7/7

Simulation time	
Our environment (s)	ModelSim (Behavioral) (s)
0.041	35.5
0.040	34.0
0.040	33.5
0.040	33.0
1.724	1501
0.787	678

low-level simulation in ModelSim, our simulation environment achieves speed-ups in simulation time ranging from 825x to 866x and 845x, on average, for the four designs shown in Table I.

4.1.2 Block Matrix Multiplication. In our design of block matrix multiplication, we first decompose the original matrices into a number of smaller matrix blocks. The multiplication of these smaller matrix blocks is then performed within the customized hardware peripheral. The software program is responsible for controlling data to and from the customized hardware peripheral, combining the multiplication results of these matrix blocks, and generating the

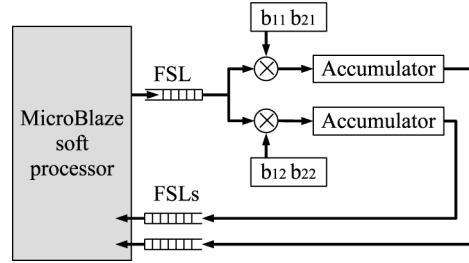


Fig. 8. Matrix multiplication with customized hardware peripheral for 2×2 matrix block multiplication.

resulting matrix. As is shown in Eq. 2, to multiply two 4×4 matrices, A and B , we decompose them into four 2×2 matrix blocks, respectively, (i.e., $A_{i,j}$ and $B_{i,j}$, $1 \leq i, j \leq 2$). To minimize the required data transmission between the processor and the hardware peripheral, the matrix blocks, of matrix A are loaded into the hardware peripheral column by column so that each block of matrix B only needs to be loaded once into the hardware peripheral.

$$\begin{aligned}
 A \cdot B &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
 &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}
 \end{aligned} \tag{2}$$

- *Implementation.* The architecture of our block matrix multiplication based on 2×2 matrix blocks is shown in Figure 8. Similar to the design of the CORDIC algorithm, the data elements of matrix blocks from matrix B (e.g., b_{11} , b_{21} , b_{12} and b_{22} in Figure 8) are fed into the hardware peripheral as control words. That is, when data elements of matrix blocks from Matrix B are available in the FSL FIFO, $Out\#_control$ becomes high and the hardware peripheral puts these data elements into the corresponding registers. Thus, when the data elements of matrix blocks from matrix A come in as normal data words, the multiplication and accumulation are performed accordingly to generate the output results.
- *Design space exploration.* We consider different implementations of the block matrix multiplication algorithm with different number of N , the size of the matrix blocks used by the customized hardware peripherals. For larger N employed by the customized hardware peripherals, a shorter execution time can potentially be achieved by the block matrix multiplication application. At the same time, more hardware resources would be used by the configuration of the MicroBlaze processor.

The time performance of various implementations of block matrix multiplication is shown in Figure 9, while their resource usage is shown in Table I. For multiplication of two 12×12 matrices, the MicroBlaze processor with a customized hardware peripheral for performing 4×4 matrix block multiplication results in a 2.2x speed-up compared with “pure” software implementation.

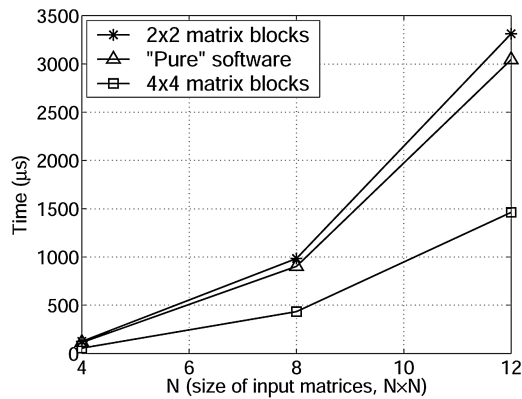


Fig. 9. Time performance of our design of block matrix multiplication.

Attaching the customized hardware peripheral to the MicroBlaze processor also requires an additional 767 (17%) more slices.

Note that attaching a customized hardware peripheral for computing 2×2 matrix blocks to the MicroBlaze processor results in worse performance for all the performance metrics considered. It uses 8.8% more execution time, 56 (8.6%) more slices and 2 (67%) more embedded multipliers, compared with the corresponding “pure” software implementations. This is because the communication overhead for sending data to and back from the customized hardware peripheral is greater than the time saved by the parallel execution of multiplying the matrix blocks.

- *Simulation speed-ups*: Similar to Section 4.1.1, we compare the simulation time in the proposed cycle-accurate arithmetic-level cosimulation environment with that of low-level behavioral simulation in ModelSim. Speed-ups in simulation time of 871x and 862x (866x on average) are achieved for the two different designs of the matrix multiplication application, as shown in Table I.
- *Analysis of simulation performance*. For both the CORDIC division application and the block matrix multiplication application, our cosimulation environment consistently achieves simulation speed-ups of more than 800x, compared with the low-level behavioral (functional) simulation using ModelSim.

By utilizing the C/C++ APIs (application program interfaces) for the System Generator tool provided by Xilinx, we are able to tightly integrate the instruction set simulator for MicroBlaze processor with the simulation models for the other Simulink blocks. The simulators integrated into our cosimulation environment run in lock step with each other. That is, the synchronization of the simulation processes within the hardware and software simulators and the exchange of simulation data between them occur at each Simulink simulation cycle. Thus, the hardware–software partitioning of the target application and the amount of data that needs to be exchanged between the hardware and software portions of the application would have little impact on the simulation speed-ups that can be achieved. Therefore, for the various

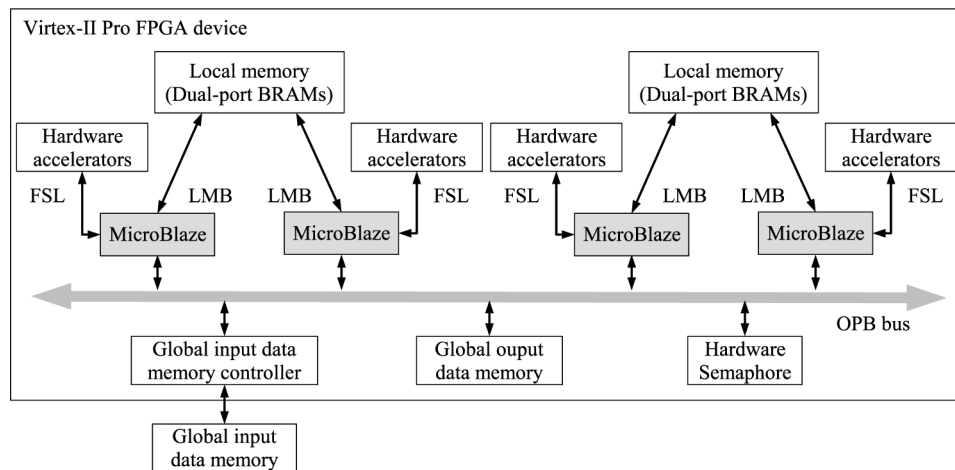


Fig. 10. The configurable multiprocessor platform with four MicroBlaze processors for the JPEG2000 encoding application.

settings of the two applications considered in this section, the variance of the simulation speed-ups is relatively small and we are able to obtain consistent speed-ups for all the design cases considered in our experiments.

4.2 Cosimulation of a Complete Multiprocessor Platform

The cosimulation of the complete multiprocessor platform is illustrated through the design of the 2D DWT (Discrete Wavelet Transform) processing task of a motion JPEG2000 encoding application. Motion JPEG2000 encoding is a widely used image-processing application. Performing JPEG2000 encoding on a 1024-pixel-by-768-pixel 24-bit color image takes around 8 s on a general-purpose processor [Galicki 2003]. Encoding a 1 min video clip with 50 frames per second would take over 6 hours. 2D DWT is one of the most time-consuming processing tasks of the motion JPEG2000-encoding application. In the motion JPEG2000 application, the original input image is decomposed into a set of separate small image blocks. The 2D DWT processing is performed on each of the small image blocks to generate output for each of them. The 2D DWT processing within the motion JPEG2000 application exhibits a large degree of parallelism, which can be used to accelerate its execution. Employing a configurable multiprocessor platform for 2D DWT processing allows for rapid development, while potentially leading to a significant speed-up in execution time.

The design of the configurable multiprocessor platform for performing 2D DWT processing is shown in Figure 10. Different numbers of MicroBlaze processors are used to concurrently process the input image data. Each of the processors has its local memory accessible through the instruction-side and data-side LMB buses. The local memory is used to store a copy of software programs and data for the MicroBlaze processors. By utilizing the dual-port BRAM blocks available on Xilinx FPGAs, two processors share one BRAM block as their local memory. Customized hardware peripherals can be attached to the processors

as hardware accelerators through the dedicated FSL interfaces. The multiple MicroBlaze processors are connected together using an OPB (on-chip peripheral bus)-shared bus interface and gain access to the global hardware resources (e.g., the off-chip global input data memory and the hardware semaphore).

A Single program multiple data (SPMD) programming model is employed for the development of software programs on the multiprocessor platform. The input image data is divided into multiple small image blocks. Coordinated by a OPB-based hardware semaphore, the multiple processors fetch the image blocks from the off-chip global input data memory through the OPB bus. The MicroBlaze processors store the input image blocks in their local memory. The processors then run the 2D DWT software programs stored in their local memory to process the local copies of the image blocks. Once the 2D DWT processing of an image block is finished, the processors send the output image data to the global input data memory coordinated by the hardware semaphore. (See [James-Roxby et al. 2004] for more details on the design of the multiprocessing platform.)

To simulate the multiprocessor platform, multiple MicroBlaze Simulink blocks are used. Each of the MicroBlaze Simulink blocks is responsible for simulating one MicroBlaze processor. The 2D DWT software programs are compiled and provided to the MicroBlaze Simulink blocks for simulation using the MicroBlaze instruction set simulator. The hardware accelerators are described within MATLAB/Simulink. Each processor and its hardware accelerators are placed in a MATLAB/Simulink subsystem. The arithmetic behavior of these MicroBlaze based subsystems can be verified separately by following the cosimulation procedure described in Section 4.1 before they are integrated into the complete multiprocessor platform. The simulation of the OPB-shared bus interface is performed by an OPB Simulink block. The global input/output data memory and the hardware semaphore are described within MATLAB/Simulink.

- *Design space exploration.* We consider different configurations of the multiprocessor platform as described above with the number of MicroBlaze processors used for processing the input image data.

The time performance of the multiprocessor platform under different configurations for 2D DWT processing is shown in Figure 11. For cases that no hardware accelerators are employed, the execution time speed-ups obtained from our arithmetic-level cosimulation environment are consistent with those of the low-level implementations reported in James-Roxby et al. [2004]. For both cases that either hardware accelerators are employed or the hardware accelerators are not employed, the maximum execution time speed-up achieved by the multiprocessor platform is 3.18x. When no hardware accelerators are employed, the time performance of the multiprocessor system fails to improve on increasing the number of MicroBlaze processors beyond 4. When hardware accelerators are employed, the time performance of the multiprocessor system fails to improve on increasing the number of MicroBlaze processors beyond 2. Therefore, when no hardware accelerators are employed, the optimal configuration of the multiprocessor platform is the configuration that uses four MicroBlaze processors. When hardware accelerators are employed,

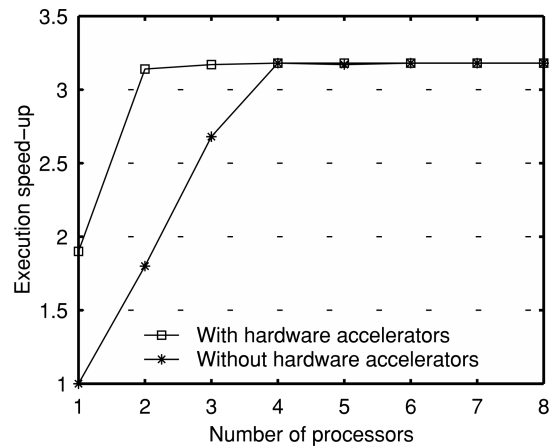


Fig. 11. Execution time speed-ups of the 2D DWT task.

the optimal configuration of the multiprocessor platform is the configuration that uses two MicroBlaze processors.

These optimal configurations of the multiprocessor platform are identified within our arithmetic-level cosimulation environment without the generation of low-level implementations and low-level simulation. This is compared against the approach discussed in, [James-Roxby et al. 2004]. In [James-Roxby et al. 2004], the optimal configurations of the multiprocessor platform are identified after the low-level implementations of the multiprocessor platform are generated and time-consuming low-level simulation is performed based on these low-level implementations.

Besides, the application designer can further identify the performance bottlenecks using the simulation information gathered during the arithmetic-level cosimulation process. Considering the 2D DWT application, the utilization of the OPB bus interface can be obtained from the arithmetic level cosimulation processes, which is shown in Figure 12. The OPB bus provides a shared channel for communication between the multiple processors. For the processing of one image block, each MicroBlaze processor needs to fetch the input data from the global data input memory through the OPB bus. After the 2D DWT processing, each processor needs to send the result data to the global data memory through the OPB bus. The OPB bus thus acts as a processing bottleneck that limits the maximum speed-ups that can be achieved by the multiprocessor platform.

- *Simulation speed-ups.* The simulation speed-ups achieved using our arithmetic-level cosimulation approach as compared to those of low-level simulation using ModelSim are shown in Figure 13. Similar to Section 4.1, the time for generating the low-level simulation models that can be simulated in ModelSim is not accounted for in the experimental results shown in Figure 13. For the various configurations of the multiprocessor platform, we are able to achieve simulation speed-ups ranging from 830x to 967x and 889x on

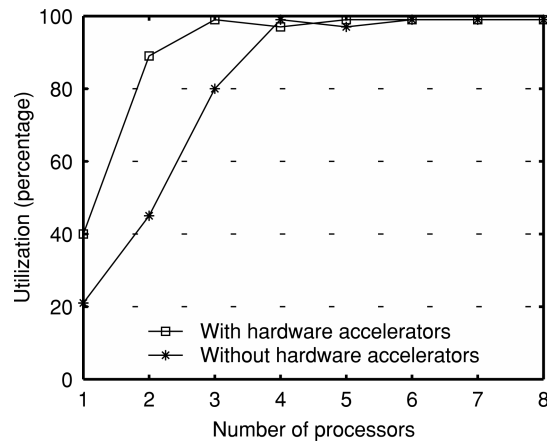


Fig. 12. Utilization of the OPB bus interface when processing the 2D DWT task.

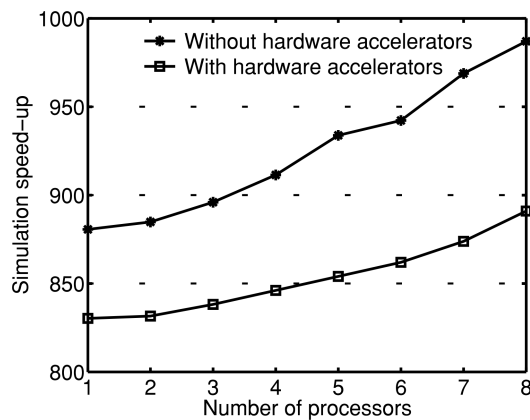


Fig. 13. Simulation speed-ups achieved by the arithmetic-level cosimulation environment.

average, when simulating using our arithmetic-level cosimulation environment, as compared with the low-level behavioral simulation using ModelSim.

- *Analysis of simulation performance.* For the 2D DWT processing task, the proposed arithmetic-level cosimulation environment is able to achieve simulation speed-ups very close to that of the two numerical processing applications discussed in Section 4.1.

Higher simulation speed-ups are achieved as more processors are employed in the designs. The MicroBlaze cycle-accurate instruction-set simulator is a manually optimized C simulation model. Simulating the execution of software programs on MicroBlaze is much more efficient using the instruction-set simulator than using the behavioral simulation model within ModelSim. As more processors are employed in the design, a larger portion of the complete system will be simulated using the instruction-set simulators, which leads to increased simulation speed-ups of the complete system.

In addition, with the same number of MicroBlaze processors employed in the systems, simulation of systems without hardware accelerators consistently have slightly higher simulation speed-ups compared with those with hardware accelerators. This is mainly because two reasons. One reason is the high simulation speed offered by the MicroBlaze instruction-set simulator as discussed above. Less Simulink blocks are also used for describing the arithmetic-level behavior of the systems when no hardware accelerators are used. This would reduce the communication overhead between the Simulink simulation models and thus help to increase the simulation speed-ups.

5. CONCLUSION

We have proposed a design space exploration technique based on arithmetic-level cycle-accurate hardware–software cosimulation for application development using FPGA-based configurable multiprocessor platforms. An implementation of the proposed technique based on MATLAB/Simulink is provided to illustrate the construction of the proposed arithmetic-level cosimulation environment. The design of several numerical computation and image-processing applications were provided to demonstrate the effectiveness of the proposed design space exploration technique based on arithmetic-level cosimulation.

Application development on configurable multiprocessor platforms using multiple clock signals can improve time and energy performance. For the MATLAB/Simulink implementation discussed in Section 3.2, by manipulating the different processing rates of the Simulink blocks and by utilizing the rate-propagation mechanism provided by the Simulink, our work can be readily extended to support the cosimulation of designs driven by multiple clock sources [Ou 2006]. Another extension of our work is rapid energy-estimation and energy-performance optimization for configurable multiprocessor platforms. We have done prior work on rapid energy estimation of computations on soft processors [Ou and Prasanna 2004b] and parallel hardware designs using FPGAs [Ou and Prasanna 2004a]. We are working on integrating these two rapid energy estimation-techniques into the proposed cosimulation framework [Ou and Prasanna 2005].

ACKNOWLEDGMENTS

This work is supported by United States National Science Foundation (NSF) under award No. CCR-0311823. The authors would like to thank Brent Milne, Haibing Ma, and Jim Hwang for providing early access to the next release of System Generator, as well as helping to integrate the work proposed in the paper into the tool. The authors would also like to thank Phil James-Roxby for offering us the original VHDL multiprocessor design of the JPEG2000 application. Editorial assistance from Aditya Kwatra is gratefully acknowledged.

REFERENCES

- ALTERA, INC. <http://www.altera.com/>.
 ANDRAKA, R. 1998. A survey of CORDIC algorithms for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, Monterey, California. ACM Press, New York.

- BALARIN, F., CHIODO, M., ENGELES, D., AND ET AL. 1997. hardware–software Codesign of embedded systems. the POLIS approach. Kluwer Academic Publ. Boston, MA.
- CELOXICA, INC. DK4. <http://www.celoxica.com/products/tools/dk.asp>.
- CHAPPELL, S. AND SULLIVAN, C. 2004. Handel-C for co-processing and Codesign of field programmable System-on-Chip. Celoxica, Inc., online available at <http://www.celoxica.com/>.
- CONG, J., FAN, Y., HAN, G., JAGANNATHAN, A., REINMAN, G., AND ZHANG, Z. 2005. Instruction set extension with shadow registers for configurable processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.
- COOK, G. W. AND DELP, E. J. 1995. An investigation of scalable SIMD I/O techniques with application to parallel JPEG compression. *Journal of Parallel and Distributed Computing (JPDC)*.
- GAISLER RESEARCH, INC. LEON3 user manual. online available at <http://www.gaisler.com/>.
- GALICKI, P. 2003. FPGAs have the multiprocessing I/O infrastructure to meet 3G base station design goals. *Xilinx Xcell Journal*, online available at http://www.xilinx.com/publications/xcellonline/xcell_45/xcell_45.pdf.
- GUPTA, S., LUTHRA, M., DUTT, N., GUPTA, R., AND NICOLAU, A. 2003. Hardware and interface synthesis of fpga blocks using parallelizing code transformations. In *International Conference on Parallel and Distributed Computing and Systems*.
- HALL, M., DINIZ, P., BONDALAPATI, K., ZIEGLER, H., DUNCAN, P., R. JAIN, AND GRANACK, J. 1999. DE-FACTO: A design environment for adaptive computing technology. In *Proceedings of IEEE Reconfigurable Architectures Workshop (RAW)*.
- HARTENSTEIN, R. AND BECKER, J. 1997. Hardware/software Codesign for data-driven Xputer-based accelerators. In *International Conference on VLSI Design: VLSI in Multimedia Applications*.
- IMPULSE ACCELERATED TECHNOLOGY, INC. Codeveloper. <http://www.impulsec.com/>.
- JAMES-ROXBY, P., SCHUMACHER, P., AND ROSS, C. 2004. A single program multiple data parallel processing platform for FPGAs. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- JIN, Y., RAVINDRAN, K., SATISH, N., AND KEUTZER, K. 2005. An FPGA-based soft multiprocessor system for ipv4 packet forwarding.
- JONES, A. K., HOARE, R., AND KUSIC, D. 2005. An FPGA-based VLIW processor with custom hardware execution. In *Proceedings of ACM International Symposium on Field Programmable Gate Arrays (FPGA)*.
- LAMPRET, D., CHEN, C.-M., MLINAR, M., ET AL. OpenRISC 1000 architecture manual. online available at <http://www.opencores.org/>.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of International Symposium on Microarchitecture (MICRO)*.
- MATHWORKS, I. <http://www.mathworks.com/>.
- MCCLOUD, S. 2004. Algorithmic C synthesis optimizes ESL design flows. *Xilinx Xcell Journal*.
- MENTOR GRAPHICS. Catapult C synthesis. http://www.mentor.com/products/c-based_design/.
- MENTOR GRAPHICS, INC. <http://www.mentor.com/>.
- OPEN SYSTEMC INITIATIVE. online available at <http://www.systemc.org/>.
- OU, J. 2006. *Rapid and Energy Efficiency hardware–software Development Using Reconfigurable SoCs*. Ph.D. thesis, University of Southern California (in preparation).
- OU, J. AND PRASANNA, V. K. 2004a. PyGen: A MATLAB/Simulink based tool for synthesizing parameterized and energy efficient designs using fpgas. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- OU, J. AND PRASANNA, V. K. 2004b. Rapid energy estimation of computations on FPGA based soft processors. In *Proceedings of IEEE International System-on-a-Chip Conference (SoCC)*.
- OU, J. AND PRASANNA, V. K. 2005. Rapid energy estimation for hardware/software application development using fpgas. In *Proceedings of International Conference on Engineering of Reconfigurable Systems And Algorithms (ERSA)*.
- PALEM, K. V., TALLA, S., AND WONG, W.-F. 2001. Compiler optimizations for adaptive EPIC processors. In *Workshop on Embedded Software*.
- PLAKS, T. P. 2001. Engineering of reconfigurable hardware/software objects. *Journal of Supercomputing*.

SHI, C., HWANG, J., McMILLAN, S., ROOT, A., AND SINGH, V. 2004. A system level resource estimation tool for FPGAs. In *Proceedings of International Conference on Field Programmable Logic and its applications (FPL)*.

XILINX, INC. <http://www.xilinx.com/>.

Received February 2005; accepted July 2005