A Message-passing Hardware/Software Co-simulation Environment to Aid in Reconfigurable Computing Design using TMD-MPI

Manuel Saldaña
Arches Computing Systems
Toronto, Canada
ms@archescomputing.com

Emanuel Ramalho, Paul Chow
University of Toronto
Toronto, Canada
{eramalho,pc}@eecg.toronto.edu

Abstract

High-performance reconfigurable computers (HPRC) provide a mix of standard processors and FPGAs to collectively accelerate applications. This introduces new design challenges, such as the need for portable programming models across HPRCs, and system-level verification tools. In this paper, we extend previous work on TMD-MPI to include an MPI-based approach to exchange data between X86 processors and hardware engines inside FPGAs that improves design portability by hiding vendor-specific communication details. Also, we have created a tool called the Message-passing Simulation Framework (MSF) that we use to develop TMD-MPI itself as well as an application development tool that enables an FPGA in simulation to exchange messages with other X86 processors.

As an example, we simulate a LINPACK benchmark hardware core using an Intel-FSB-FPGA platform to quickly prototype the hardware, to test the communications and to verify the benchmark results.

1. Introduction

High-performance reconfigurable computers (HPRC) are now in a similar stage as supercomputers were before the appearance of MPI [14]. Every vendor had their own message-passing API to program their own supercomputers causing a lack of portable designs. Currently there is no standard API for the interaction between processors and FPGAs. Companies such as Cray, SGI, Intel, XtremeData, DRC and SRC provide their own software APIs and their own hardware interfaces for application hardware engines. This situation reduces the portability and productivity because vendor specific details distract designers from focusing on the application algorithm. We believe that in the same way a standard C program runs in any X86 processor with most operating systems, a standard VHDL/Verilog design can be implemented on any FPGA; with the exceptions of using non-standard code, for example specific resources on a given chip or non-ANSI C functions.

Additional to the portability issue, the mix of X86 processors (X86 from now on) and FPGAs introduces new design challenges that require new design tools. For example, testing and debugging procedures for software are different from the procedure used in hardware. A typical testing procedure in software is a step-by-step execution or printing debug information to the screen. In contrast, for hardware components, such as FPGAs, a detailed behavioral or even timing simulation is required. Both, software and hardware, can be tested independently up to a certain extent but system-level features or dynamic interaction between them is harder to test that way. For example, a bus functional model (BFM) helps with verifying and developing low-level interactions with a given communication interface, but higher-level protocols or application-level protocols cannot be tested, especially if the behavior changes based on the data received or sent.

In this paper we extend previous work on TMD-MPI [11], which implements a subset of the MPI standard targeting multiple computing elements (hardware engines and embedded processors) inside FPGAs to include X86 processors enabling a uniform and portable MPI-based communication mechanism for HPRCs. To do this, we developed the Message-passing Simulation Framework (MSF) that allows multiple X86 processes, running at full speed, to exchange messages with computing elements inside the FPGAs being simulated. In that way, we exercise the system-level interaction while having full visibility of what happens inside the FPGA.

In this paper, we perform a functional, system-level simulation of the LINPACK benchmark [8] with the purpose of testing the communications, the simulation environment and quickly prototyping and verifying the correctness of the LINPACK hardware engine, which is in the early stages of development.

The rest of the paper is organized as follows: Section 2 provides a quick overview of previous work on TMD-MPI. Section 3 contrasts our work to other related co-simulation environments. Section 4 presents the communication infrastructure and its simulation framework. Section 5 describes an example simulation of the LINPACK benchmark system using the MSF. Finally, Sections 6 and 7 present future work and conclusions.

2. Background

As mentioned before, we use TMD-MPI to provide an abstraction layer for the communications. TMD-MPI has been developed as a result of the need for a programming model for the TMD machine being developed at the University of Toronto [10]. The TMD machine is a scalable Multi-FPGA configurable system designed to accelerate computing intensive applications. Previously, TMD-MPI only supported MPI-based communication between PowerPC embedded processors, MicroBlaze soft-processors and hardware engines (collectively known as computing elements) across multiple FPGAs, but now with HPRC featuring tightly coupled FPGAs to X86 processors, we have extended TMD-MPI to include X86 processors using shared memory as a medium to exchange messages.

TMD-MPI does not include all the MPI functionality described in the standard because it is targeted to embedded systems with limited resources, as is the case of FPGAs, and because functionality will be added as needed. However, it supports blocking and non-blocking communications as well as some collective operations, which is enough to implement many parallel applications. With the appearance of HPRC machines, a new window of opportunities arises to have a more complete implementation of the standard.

The TMD-MPI programming model is based on the assumption that, from the communications perspective, computing elements inside FPGAs can be treated as peers rather then just co-processing units, which is the way a typical MPI program works. Also, modern FPGAs have enough resources to host several computing elements interconnected using an on-chip network, which TMD-MPI also abstracts from the user.

In an FPGA-as-coprocessor model, an X86 usually acts as a message relay between computing elements located in different FPGAs introducing big latencies and limiting what FPGAs can do in terms of communication. In a peer-to-peer model, computing elements can exchange data between them regardless of their physical location and without intermediaries, which reduces the latency and also simplifies the programming model.

We have used TMD-MPI in multi-FPGA machines based on Amirix [1] and BEE2 [3] boards to implement Molecular Dynamics. Currently we are porting the application to use a HPRC with Intel processors and Xilinx FPGAs attached to the FSB; it is for the latter case that we created the MSF, to help us develop and verify our designs.

3. Related Work

There has been abundant research on co-design methodologies and co-simulation environments [5]. The research concludes that the lack of a system-level view of a mixed HW/SW system leads to difficulties in verifying the entire system, and hence to incompatibilities across the HW/SW boundary leading to inefficient designs. However, most of the research focuses on embedded systems with microcontrollers, DSPs, ASICs and FPGAs, but the research does not address explicitly the High-performance Supercomputing sector. The appearance of FPGAs in Supercomputers opens a new window of opportunities to adapt and apply co-design techniques and co-simulation environments to HPRCs. Our TMD-MPI and MSF is one step towards that direction by framing co-simulation into an MPI-based paradigm.

Most of the co-simulation environments typically use hardware in the form of accelerators to speedup the simulation itself. For example, in [13], the authors provide a co-simulation environment where an X86 and an FPGA are placed on a dual socket motherboard to accelerate a processor simulation tool called Simplescalar. In [15], the authors use an FPGA plugged into the PCI bus to accelerate Modelsim's functional simulations. In contrast, we do not use the FPGA to accelerate a simulation, we use Modelsim [9] running in an X86 to simulate and emulate the FPGA, and let it interact with the X86 processors as if the FPGA were present. Once the design inside the FPGA has been verified in simulation it can run at full speed in the real FPGA.

Other vendor-specific simulation frameworks such as Cray's simulation framework [4] and SGI's SSP Stub [12] only allow a Bus Functional Model (BFM) testing procedure or low-level data transfer primitives. The user can provide a set of inputs to the FPGA with certain delays and expected outputs to compare the results against. This static kind of verification is adequate to test the interaction with a given interface or for independent FPGA testing, but not as a system-level approach. Our simulation approach is more generic and portable, allowing the simulation of multiple hardware engines interacting with multiple X86 MPI software processes concurrently.

4. Simulation Environment

In this section we describe the computing architecture and the MPI-based communication system. Then we explain how the MSF enables the simulation of such architectures.

4.1. Message-Passing System Architecture

As a reference architecture, we use an Intel motherboard that has four sockets, one of them with an Intel quad-core processor and the second socket has a Xilinx XC5VLX110 FPGA, which shares 8GB of main memory with the processors via the FSB. The system runs a standard Linux SMP configuration.

Figure 1 shows an example of a parallel MPI application mapped to our reference platform. In this case, the application has a total of six tasks known as ranks in the MPI jargon. Each rank in the system (logically represented as circles in Figure 1) has its own private memory space. Three of the ranks (R0, R1, and R2) are software processes and run in three X86 cores inside the Quad-core Intel Xeon chip. The remaining three ranks (R3, R4 and R5) are inside the FPGA running as hardware engines, although they could be Microblaze soft-processors or embedded PowerPC processors as well. The X86 processors exchange messages using shared memory, and the hardware engines exchange messages using the Network-on-Chip (NoC). To exchange messages between X86s and hardware engines the data must travel through a shared memory MPI bridge (*MPI_Bridge*), which implements in hardware the same shared memory protocol that the X86 processors use. This bridge takes data to/from the NoC and issues read or write memory requests to the vendor-specific low-level communications core (LLCC), which executes the request. The *MPI_Bridge* effectively abstracts the vendor-specific communication details from the rest of the on-chip network.

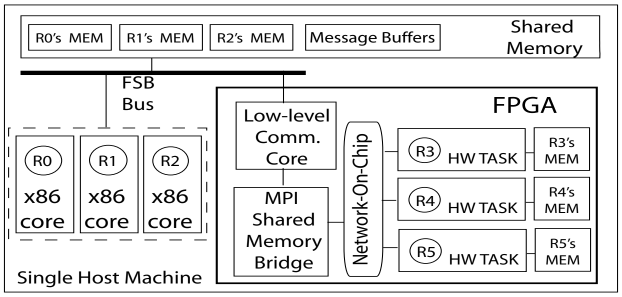

Figure 1. Shared Memory Host with three X86 processors and one FPGA, all physically attached to the FSB

For this paper, we used Intel's FSB Bus as the communication media but the same concepts can be applied to other communication media, such as AMD's HyperTransport [6], Intel's QuickPath [7], Cray's Rapid Array Transport [4], SGI's Scalable System Port-NUMA link connection [12], or even with a standard PCI Express core because they all provide a physical connection to the main system memory. The communication media determines what LLCC to use. In this paper, we use a Xilinx-Intel FSB communication core that handles the low-level protocol to read and write to memory as well as the memory coherence control.

However, TMD-MPI's shared memory message-passing protocol should be the same across HPRCs or with minor variations, the only change is the physical interconnection between the *MPI_Bridge* and the vendor-specific LLCC. By implementing a *MPI_Bridge* for each type of LLCC we make the system portable. For example, in this paper we use a MPI_Xilinx_FSB_Bridge, but we could also implement a MPI_Cray_Bridge to use a Cray HPRC machine.

An extension of this approach to a distributed memory machine (a Cluster) or many HPRC hosts is natural since message-passing assumes no shared memory. A distributed memory approach could use an *MPI_Ethernet_Bridge*, or any other point-to-point communication interface to allow the connection of multiple hosts through the FPGA itself; however, this remains future work for now and in this paper we focus only on a single host machine.

4.2. Abstraction Layers

Figure 2 shows the abstraction layers for software and hardware in the TMD-MPI programming model. A software application relies on the MPI library layer to send and receive data (calls to MPI_Send(), MPI_Recv(), etc.). In turn, TMD-MPI uses a kernel driver to allocate memory for the shared memory buffers and to perform virtual-to-physical memory translations. Data is then placed in memory via the FSB and the MPI shared memory bridge will read it and send it over the NoC, which will route the packets to the proper destination Message Passing Engine (MPE). Finally, the MPE will deliver the message to the application hardware engine. Data traveling in the opposite direction is also possible; the FPGA can be a master and send data without the X86 first having to request it.

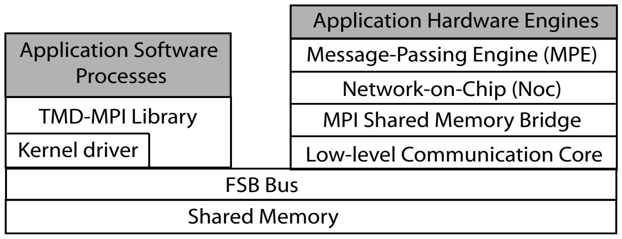
Application Hardware Engines
Application Software Processes
Message-Passing Engine (MPE)
Network-on-Chip (Noc)
TMD-MPI Library
MPI Shared Memory Bridge
Kernel driver
Low-level Communication Core
FSB Bus
Shared Memory

Figure 2. HW/SW abstraction layers

The MPE encapsulates the MPI functionality in hardware. It is responsible for handling requests, acknowledgements and full-duplex data transmission and reception. Also, it is in charge of packetizing/depacketizing large messages as well as handling unexpected messages. A hardware engine interacts with its MPE via FSLs, which are Xilinx unidirectional FIFOS. An example of the interface between a hardware engine and the MPE is further discussed in Section5.

4.3. The MSF FLI Module

By using TMD-MPI, hardware engines and software processors are isolated from machine-specific communications hardware. However, it introduces a new challenge for the design and verification of applications. In a typical MPI parallel program, an MPI rank is not tested in isolation from the other MPI ranks, it has to be tested with all ranks running at once to verify the correct collective operation and synchronization between them. With FPGAs as containers of MPI ranks, they must be part of the system-level testing process. As mentioned before, FPGA testing requires a simulation, so the question now becomes how to simulate such a system.

The MSF provides a portable simulation environment based on Modelsim that emulates the FPGA and lets the MPI ranks inside of the FPGA exchange messages with the ranks running in X86 processors. Figure 3 shows the simulation scheme of the architecture depicted in Figure 1. Note that the FPGA in Figure 1 is now an X86 core running Modelsim simulating the FPGA design. For ranks R0, R1 and R2 running in the X86 processors, the FPGA in simulation will be seen as a slow FPGA (simulation speed). In this sense, the FPGA simulation is actually an emulation of the FPGA. Naturally, the time it takes to send a message will be drastically reduced when the FPGA is no longer in simulation and runs in the real FPGA. However, keep in mind that a message-passing paradigm assumes a coarse grain parallelism in which tasks should have reasonable communication demands to be efficient and also should be latency tolerant. In other words, a correct MPI program does not rely on the time it takes to send or receive a message to produce the correct results, and therefore the latency introduced by the simulation should not change the results when the FPGA design runs in the actual physical FPGA.

The central part of the MSF is the use of Modelsim's Foreign Language Interface (FLI), which is a typical way to perform co-simulations by allowing a C program (actually a shared library) to have access to Modelsim's simulation information, such as signal or register values, components instantiated and simulation control parameters. The MSF FLI module replaces the vendor-specific LLCC by providing the required functionality directly to the MPI_Bridge. The MSF FLI accepts the MPI_bridge memory requests (address and data) and performs the reads and writes directly to shared memory. In the case of the distributed memory environment, the MSF FLI module would translate the send/receive requests to socket writes/reads allowing the interaction of remote machines with the FPGA under simulation.

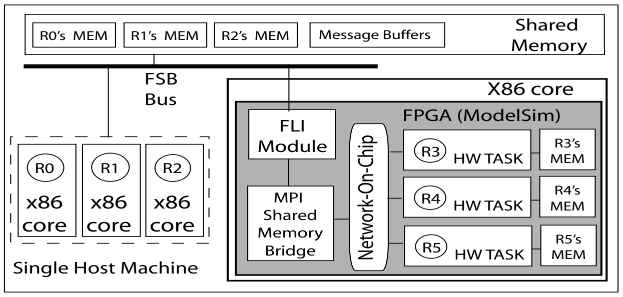

Figure 3. Simulation scheme for the reference architecture. One X86 core runs Modelsim with the FLI module for shared memory.

The MSF FLI module uses TMD-MPI's memory allocation and memory mapping subroutines to be able to access the shared memory message buffers. In a typical transaction, the MSF FLI module receives the physical addresses of a buffer from the MPI_Bridge and translates it to virtual addresses before reading or writing to main memory. This is required because the MSF FLI module is under the control of the operating system as a normal user process, which cannot access memory using a physical address.

Since there can be a variety of MPI_Bridges based on the vendor-specific LLCC, there will be a corresponding FLI module that Modelsim can load at runtime. That is, there will be MSF FLI module variations. For example, we use the *FLI_Xilinx_FSB* module, but we could implement the *FLI_Cray* to simulate the interaction with the FPGA in a Cray machine. This is convenient because the simulation itself becomes portable. TMD-MPI and the MSF FLI module absorb the platform changes and make the simulation in different HPRCs transparent to the user.

An additional advantage of the MSF is that there is no need to simulate the vendor-specific LLCC, which can be proprietary and not public, such as the Intel FSB signals. The MSF does not need to know the details of those vendor-specific internals because the FLI module provides the MPI_Bridge with the same memory access (or network device access for the distributed version) that the LLCC provides.

During the simulation the user has full visibility inside the FPGA at the resolution available in Modelsim, which is useful when tracking bugs in the design, such as glitches, signal delays or any other sub-cycle events with the caveat of reduced simulation speed. Other co-simulation environments can be faster but limit the design's visibility to its outputs treating it as a black box, and use just cycle-accurate simulations. Also, in the MSF the user has full control of the simulation by using Modelsim's console or GUI to stop it, pause it and continue it. Even breakpoints can be asserted to stop executing a particular hardware MPI rank and the software MPI ranks can continue executing because they are completely decoupled due to the implicit asynchronous nature of the message-passing programming model. Only those MPI ranks that are exchanging messages with a stopped rank will be automatically blocked if using MPI blocking calls or if there is a collective operation, such as a MPI_Barrier, MPI_Bcast, etc. However, if non-blocking communications are used, then the ranks can overlap computation and communication.

5. Case Example: LINPACK

This section presents a brief description of our LINPACK benchmark implementation and some insights of the hardware engine and its communication interface. This paper focuses on the co-simulation environment and not the performance of the LINPACK benchmark.

5.1. LINPACK Implementation

The LINPACK Benchmark [8] is a widely used algorithm that measures floating-point computing performance by solving a system of linear equations, **Ax=b**. It has two main subroutines: *DGEFA* (performs an LU decomposition on the matrix **A**) and *DGESL* (solves the system of linear equations by using vector **b**). More than 97% of the time taken to compute the benchmark is spent inside the *DGEFA* subroutine. *DGEFA* comprises three BLAS [2] level 1 functions, *IDAMAX*, *DSCAL* and *DAXPY*, where the latter, alone, is responsible for about 95% of the time spent in the *DGEFA* subroutine. The original benchmark uses double precision, but for simplicity we use single precision, which is acceptable for the purposes of this paper.

To implement a parallel version of this algorithm, we first parallelized the sequential LINPACK code using MPI with all the ranks running in X86 processors, and verify the correctness of the parallel algorithm itself purely in software. At this point, high-level application decisions can be made, such as the communication pattern or data partitioning scheme. For the LINPACK benchmark, *DAXPY* accounts for most of the time spent inside *DGEFA*, however, the *DGEFA* subroutine was chosen to be the parallelization focus to reduce the number of messages being sent across the ranks. As in Figure 3, we use six MPI ranks, all of them have the same functionality and perform the same computation, except for rank 0, which also performs the inital data distribution, stores the results back to the file system and computes the *DGESL* subroutine.

After successfully parallelizing the algorithm in software, three of the six ranks are targeted to run in the FPGA. This decision is just to show how software processes (ranks 3, 4 and 5) can be turned into engines without changing a single line of code for ranks (0,1 and 2), following the peer-to-peer model between X86 processors and hardware engines. The *DGEFA* subroutine is converted to hardware manually, without using any C-to-gates compiler, however, nothing in the MSF or TMD-MPI paradigm prevents that.

Since each rank contains a full *DGEFA* subroutine, columns of matrix A are cyclically distributed across the ranks, i.e. 1st column goes to rank 0, 2nd column to rank 1 and so on. This reduces the data communication in each iteration. After the first data distribution, which is only done once, only two broadcasts occur in each iteration, one column of matrix A and the corresponding pivot. These broadcasts are performed after the *DSCAL* function is executed and done by the rank storing the respective column, which means that each iteration will have a different broadcast source; therefore, there is communication between all the ranks. Finally, when all the data is computed, it must be sent back to rank 0, which will run the *DGESL* subroutine and end the algorithm with the residual calculation.

5.2. The DGEFA Benchmark Hardware

The *DGEFA* computing engine, shown in Figure 4, consists of a state-machine that has encoded the DGEFA benchmark flow, and a *BLAS1* block, which is a special-purpose fully pipelined engine that calculates the *BLAS* level 1 functions. The DGEFA state-machine issues send and receive commands to the MPE, similar to the MPI calls for X86 processors. The MPE implements the TMD-MPI protocol in hardware and gives the DGEFA engine the ability to communicate with all the other ranks in the system. The MPE has independent command and data FIFOs, that allow the streaming of data directly into the datapath of the *DGEFA* computing engine. Figure 4 shows how the *DGEFA* engine connects to the MPE.

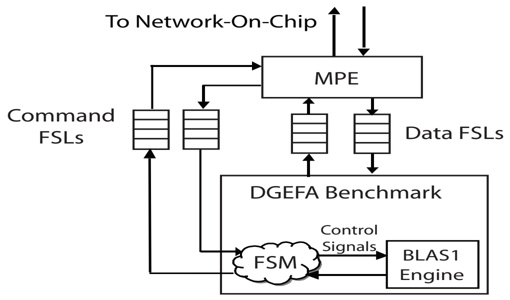

Figure 4. Interface between MPE and hardware engines

5.3. Verification of the results

At the end of the application, rank 0 has the final matrix, which is compared agains the sequential version of the code. There is a maximum error of 0.64% with an average error of 0.0011% in the results due to fact that the X86 uses 80- and 64-bit precision floating-point units compared to only 32 bits in the engine, but that can be improved. The point being made is that by using the MSF we have a way to measure further improvements to the engine's precision or mixed (X86 + engine) precision calculations.

A caveat of our approach is that the communication latency between X86s and FPGAs is not known because the MSF FLI module does not simulate the LLCC; therefore, it is hard to predict the entire application's performance exactly. However, we can estimate it based on measuring the most time consuming part of the application. By using the LINPACK function *second()* we know that the DAXPY loop takes 482μs in the X86 processor (3.4 GHz), compared to 818μs in the *DGEFA* engine (100 MHz) measured in simulation. This is a fair comparison since there is no communication involved in that loop, just raw computation.

Due to the MPI paradigm and the DGEFA core implementation it is very easy to increase the number of ranks in the system, as long as there are enough resources in the FPGA. The code (C and VHDL) does not need to change at all to include more ranks. Based on preliminary synthesis results, we can place around 16 *DGEFA* engines (excluding the on-chip network, MPEs and the MPI_Bridge) on the XC5VLX110 FPGA.

6. Future Work

Future work includes the support for external memory simulation for those systems that have memory chips next to the FPGA; this will allow us to simulate designs with larger datasets. Also, we will include the simulation of multiple FPGAs and multiple hosts, that will enable the development and simulation of larger systems.

7. Conclusions

In this paper we describe a portable MPI-based approach to co-simulate multiple hardware engines implemented in an FPGA communicating with multiple X86 processes for reconfigurable computers. Although, in this paper, we use an Intel-FSB-Xilinx-FPGA platform, the same concepts and ideas can be applied to other platforms, making this paper a first attempt, to the best of our knowledge, to standardize such communication between X86 processors and FPGAs, and hide vendor-specific details from the user during co-simulation.

To do this, we created the MSF to let X86 processors interact with an FPGA in simulation. The MSF is demonstrating its usefulness during the developing of a LINPACK system. It allows a fast compile-debug-modify-recompile cycle speeding up the design task because there is no need to run place and route to test the algorithm. We co-simulated the system using six MPI ranks, half of them running as X86 processes and the other half as hardware engines in simulation; all exchanging messages in a peer-to-peer fashion.

Acknowledgments

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the tools and funding provided for this project.

References

[1] Amirix Systems, Inc. `http://www.amirix.com/`.

[2] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[3] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Des. Test '05*, 22(2):114–125, 2005.

[4] I. Cray. *"CRAY XD1 FPGA Development"*, 2005. pp. 9-11, pp. 63-66.

[5] H. Hubert. "A Survey of HW/SW Cosimulation Techniques and Tools". Master's thesis, Royal Inst. of Tech., Stockholm, Sweeden, June 1998.

[6] HyperTransport Consortium. `http://www.hypertransport.org`.

[7] Intel. "Intel Quick Path Architecture (White Paper)". `http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf`.

[8] A. P. J. Dongarra, P. Luszczek. "The LINPACK Benchmark: Past, Present and Future". `http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf`.

[9] Mentor Graphics, Corp. `http://www.mentor.com/`.

[10] A. Patel, M. Saldaña, C. Comis, P. Chow, C. Madill, and R. Pomès. A Scalable FPGA-based Multiprocessor. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, 2006.

[11] M. Saldaña and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications*, Madrid, Spain, 2006.

[12] SGI. *"Reconfigurable Application-Specific Computing Users Guide"*, Jan 2008. pp. 9-12, pp. 223-244.

[13] H.-h. S. L. Taeweon Suh. Initial Observations of Hardware/Software Co-Simulation using FPGA in Architecture Research. In *2nd Workshop on Architecture Research using FPGA Platforms (WARFP-2006)*, February 2006.

[14] The MPI Forum. MPI: a message passing interface. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, New York, NY, USA, 1993. ACM Press.

[15] M. Wageeh, A. Wahba, A. Salem, and M. Sheirah. FPGA Based Accelerator for Functional Simulation. *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, 5:V–317–V–320 Vol.5, May 2004.