Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 376232, 9 pages
doi:10.1155/2009/376232

Research Article

A Message-Passing Hardware/Software Cosimulation Environment for Reconfigurable Computing Systems

Manuel Saldaña,[1] Emanuel Ramalho,[2] and Paul Chow[2]

[1] *Arches Computing Systems, 708-222 Spadina Avenue, Toronto, ON, Canada M5T 3A2*
[2] *The Edward S. Rogers, Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, ON, Canada M5S 3G4*

Correspondence should be addressed to Manuel Saldaña, ms@archescomputing.com and Paul Chow, pc@eecg.toronto.edu

Received 15 March 2009; Accepted 19 June 2009

Recommended by Lionel Torres

High-performance reconfigurable computers (HPRCs) 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. To address the need for cosimulating a complete heterogeneous application using both software and hardware in an HPRC, we have created a tool called the Message-passing Simulation Framework (MSF). We have used it to simulate and develop an interface enabling an MPI-based approach to exchange data between X86 processors and hardware engines inside FPGAs. The MSF can also be used as an application development tool that enables multiple FPGAs in simulation to exchange messages amongst themselves and with X86 processors. As an example, we simulate a LINPACK benchmark hardware core using an Intel-FSB-Xilinx-FPGA platform to quickly prototype the hardware, to test the communications. and to verify the benchmark results.

1. Introduction

High-performance reconfigurable computers (HPRCs) are now at a similar stage as supercomputers were before the appearance of MPI [1]. Prior to MPI, every vendor had their own Message passing application program interface (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 [2], SGI [3], Intel [4], XtremeData [5], DRC [6], and SRC [7] 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 nonstandard code, for example, specific resources on a given chip or non-ANSI C functions.

In addition 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.

A multi-FPGA multicore system with heterogeneous computing elements running in parallel requires system-level tests in addition to tests to the individual components in isolation.

In this paper we extend previous work on TMD-MPI [8, 9], 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. With this approach we exercise the system-level interaction while having full visibility of what happens inside the FPGAs.

In this paper, we perform a functional system-level simulation of the LINPACK benchmark [10] 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 cosimulation environments. Section 4 presents the communication infrastructure and its simulation framework. Section 5 explains how TMD-MPI and the MSF can help with system-level architecture exploration. Section 6 describes an example simulation of the LINPACK benchmark system using the MSF. Section 7 presents future work. Finally, conclusions are discussed in Section 8. At the end of the paper there is a glossary of all the acronyms used in this paper.

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 Toronto Molecular Dynamics (TMDs) machine being developed at the University of Toronto [11]. The TMD machine is a scalable Multi-FPGA configurable system designed to accelerate Molecular Dynamic simulations, although the machine is not limited to this particular application. In fact, the generic MPI-based programming model and the flexibility of FPGAs allow us to target a broader spectrum of computing-intensive applications; however, the TMD-MPI name still remains for historical reasons.

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 run on FPGAs with limited resources, such as memory (e.g., 4 MB of on-chip RAM in Virtex5 chips compared to GigaBytes for a typical X86 system). Nevertheless, functionality can be added as needed depending on the application. TMD-MPI supports blocking and nonblocking 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 opportunity arises to have a more complete implementation of the standard. For example, including MPI-2 functionality such as remote memory access to implement DMA capabilities to exchange data between X86 processors and FPGAs.

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 than just coprocessing 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. A system-level approach at the beginning of the design flow can help to conceive hardware accelerators as peers to processors rather than mere coprocessors.

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. For example, in a typical Master-Slave parallel program, FPGA coprocessors attached to the software slave processes would actually be slaves of the slave processes. In contrast, with a peer-to-peer model, Computing Elements (including X86 processors) can exchange data between themselves 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 [12] and BEE2 [13] boards to implement Molecular Dynamics. Currently we are porting the application to use an 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 codesign methodologies and cosimulation environments [14]. 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 opportunities to adapt and apply codesign techniques and cosimulation environments to HPRCs. Our TMD-MPI and MSF are one step towards that direction by framing cosimulation into an MPI-based paradigm.

Most of the cosimulation environments typically use hardware in the form of accelerators to speedup the simulation itself. For example, in [15], the authors provide a cosimulation environment where an X86 and an FPGA

are placed on a dual socket motherboard to accelerate a processor simulation tool called Simplescalar. In [16], the authors use an FPGA plugged into the PCI bus to accelerate ModelSim's [17] functional simulations. In contrast, we do not use the FPGA to accelerate a simulation. We use ModelSim running in an X86 to simulate and emulate the FPGA, and let it interact with other X86 processors as if the FPGAs 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 [18] and SGI's SSP Stub [19] 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 multi-FPGA approach. Our simulation approach is more generic and portable, allowing the simulation of multiple FPGAs, each with multiple hardware engines (possibly heterogeneous) interacting with multiple X86 MPI software processes concurrently.

4. Simulation Environment

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

4.1. Reference Architecture. Figure 1 shows our reference architecture, which is based on an Intel 4-Processor Server System S7000FC4UR motherboard and the Xilinx ACP M2 FPGA modules distributed by Nallatech [20]. These modules can be stacked one on top of another (M2 Stack) to group a number of FPGAs and plug them into the processor socket on the motherboard, providing higher compute density. Any combination between three Intel Xeon quad-core processors and one M2 Stack, or three M2 Stacks and one Intel Xeon quad-core processor is permitted. In all cases they share the main system memory through the FSB North Bridge chip.

The M2-Stack consists of two kinds of M2 modules: the M2 Base (M2B) and the M2 Compute (M2C). The M2B module has one XC5VLX110 FPGA, which contains the Xilinx FSB interface to provide access to the main system memory. The M2C module has two XC5VLX330 FPGAs and it plugs on top of the M2B. An additional M2C can be stacked on top of the first M2C. The FPGAs in this three-layer stack are connected through parallel LVDS lines.

Currently, the system runs a 64-bit CentOS Linux SMP operating system with 8 GB of memory.

4.2. Message Passing on the ACP Platform. Figure 2 shows an example of a parallel MPI application mapped to our reference platform. For simplicity, in this case, one Quad-core Xeon processor and one M2B module are used. The application has a total of six tasks known as ranks in the MPI jargon. Each rank in the system (logically represented as ovals in Figure 2) has its own private memory space (on-chip memory). Three of the ranks (R0, R1, and R2) are software processes and run in three X86 cores inside the Quad-core Intel Xeon processor. The remaining three ranks (R3, R4, and R5) are inside the FPGA running as hardware engines (hardware ranks), 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.

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 [21], Intel's QuickPath [22], Cray's Rapid Array Transport [18], SGI's Scalable System Port-NUMA link connection [19],

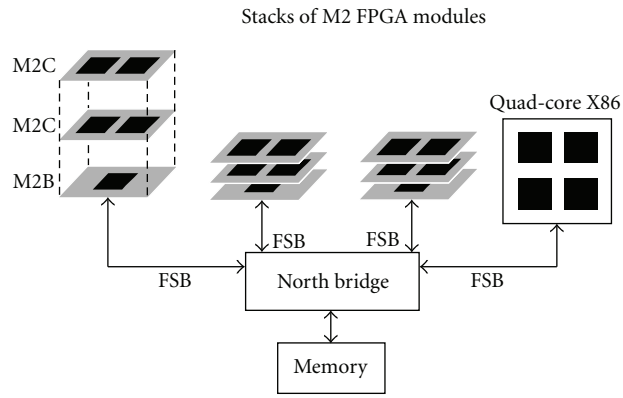


Figure 1: Stacks of Xilinx M2 FPGA modules that can be placed in standard CPU sockets on the motherboard.

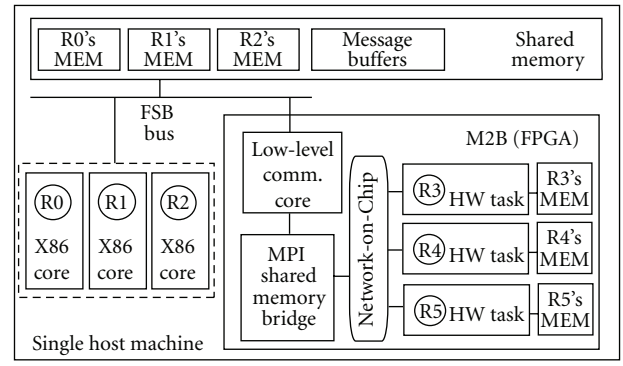


Figure 2: Example of a 6-rank MPI application, with three ranks mapped to X86 processor cores and three ranks mapped to hardware engines.

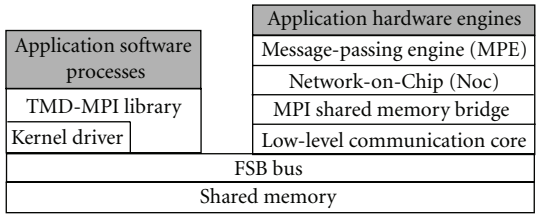


Figure 3: HW/SW abstraction layers.

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.

TMD-MPI's shared-memory, message-passing protocol should be mostly the same across HPRCs. The only change is the physical interconnection between the *MPI_Bridge* and the vendor-specific LLCC. By implementing an *MPI_Bridge* for each type of LLCC we make the system portable. For example, in this paper we use an MPI_Xilinx_FSB_Bridge, but we could also implement an 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.

In Figure 2 only one FPGA is shown, but multi-FPGA systems can also be part of the MPI communication by plugging two more M2B modules or stacking M2C modules. This is further explained in Section 4.5.

4.3. Abstraction Layers. Figure 3 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, to perform virtual-to-physical memory translations and some low-level setup for the LLCC in the FPGA. 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.

The MPE encapsulates part of the MPI functionality in hardware. It is responsible for handling requests, acknowledgments, 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 Section 6.

With this communications architecture, applications become more portable because the hardware accelerators should remain unchanged from one HPRC to another as well as the software code for the X86 processors since they are all behind the message-passing abstraction.

4.4. 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 cycle-accurate simulation, so the question now becomes how to simulate such a system. Furthermore, the complexity of the testing process increases if there are multiple FPGAs to simulate, each with potentially different hardware engines or embedded processors.

The MSF provides a portable simulation environment based on ModelSim that emulates the FPGA and lets the MPI ranks inside of the FPGAs exchange messages with the ranks running in X86 processors or in other FPGAs. Figure 4 shows the simulation scheme of the architecture depicted in Figure 2. Note that the FPGA in Figure 2 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) [23], which is a typical way to perform cosimulations 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

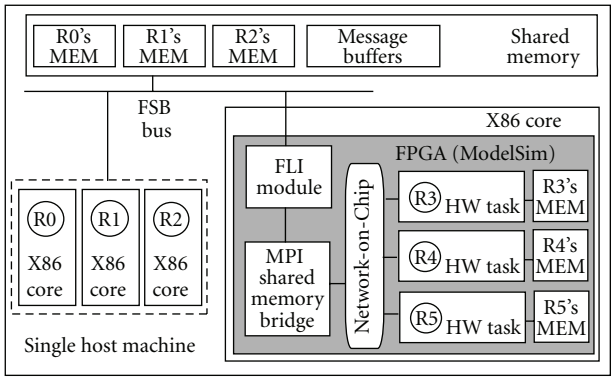


Figure 4: Simulation scheme for the sample application. One X86 core runs ModelSim with the FLI module for shared memory.

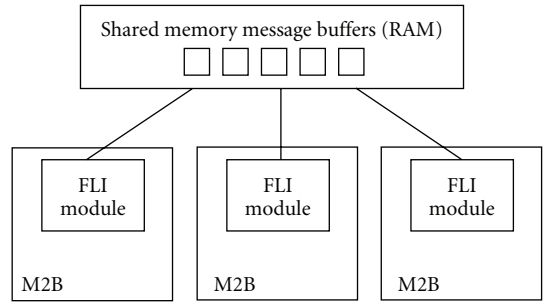


Figure 5: Multiple FSB FLI Modules can independently access the message buffers in shared memory during simulation.

the send/receive requests to socket writes/reads allowing the interaction of remote machines with the FPGA under simulation.

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 them 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. In other words, the MSF FLI module models the functional behaviour of the LLCC in terms of memory access.

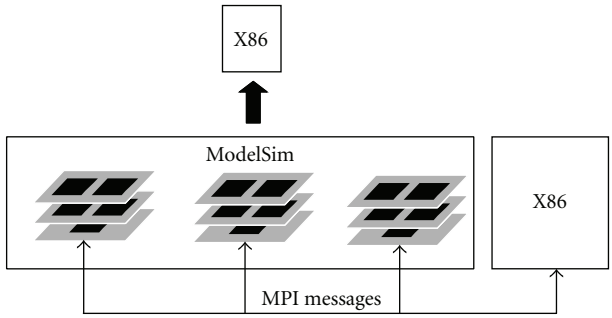
During the simulation, the user has full visibility inside the FPGA at the resolution available in ModelSim, which is useful when tracking bugs in the design, such as glitches, signal delays, or any other subcycle events with the caveat of reduced simulation speed. Black-box cosimulation environments can be faster than ModelSim but limit the design's visibility to its outputs, and only use cycle-accurate simulations, whereas Modlsim can simulate subcycle delays. 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 an MPI_Barrier and MPI_Bcast. However, if nonblocking communications are used, then the ranks can overlap computation and communication.

4.5. Multistack Simulation. At this point, only one M2B plus one Xeon processor system has been presented to explain the basic concept of TMD-MPI and the MSF. However, a fully populated system with three M2-stacks (1 M2B and 2 M2C per stack) would contain 15 FPGAs and a quad-core Xeon processor per motherboard. To perform system-level tests, it should be possible to simulate all the FPGAs at once while communicating with MPI software processes running in the X86.

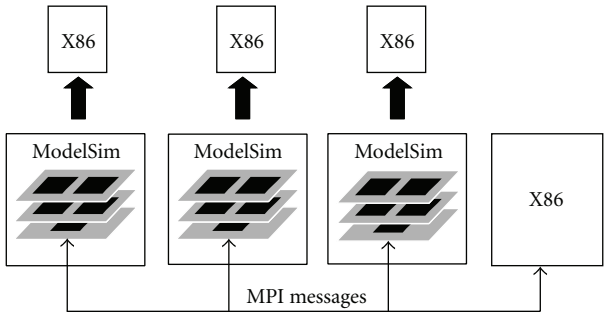
Such a multi-FPGA, multistack simulation imposes challenges on the CAD tools to generate simulation models and set up testbenches. In addition, an interstack communication mechanism for simulation is required, and it poses an increased demand on computing power to simulate such large systems.

To address the multistack communication mechanism in simulation, the shared memory protocol in the MPI_FSB_Bridge and the TMD-MPI software library were modified to support this feature. However, the MSF FLI module only required minor modifications because it is only an intermediary for performing memory reads and writes as commanded by TMD-MPI's shared-memory message-passing protocol. From the MSF perspective, interstack communication was a straightforward extension of the one-stack MSF version. Now, as shown in Figure 5, multiple FLI modules (multiple M2Bs in simulation) exchange information through shared memory without Xeon processor intervention, or having to set up or start the communication.

To generate the simulation models, the MSF uses the Xilinx *simgen* command included as part of the ISE/EDK design tools. *Simgen* generates a self-contained simulation model of an EDK project, which represents an entire FPGA



(a) One single ModelSim process is executed by a single CPU.



(b) Multiple ModelSim processes take advantage of multicore processors.

Figure 6: Two options to simulate multiple M2 Module Stacks.

design. This includes simulation models for the components instantiated inside the FPGA. The MSF then runs *simgen* for as many FPGAs as there are in the system, and finally it creates a ModelSim script to generate the clock and reset signals.

Once the simulation models for all the FPGAs are generated there are two options on how to set up the simulation using ModelSim, and this is shown in Figure 6. One option is to create a top-level testbench that instantiates the FPGAs as components in the testbench and simulates all the FPGAs in one ModelSim process. The second option is to start three ModelSim processes, one per stack, and let them simulate independently. In either case, the FLI module will communicate the same way (through shared memory) and every M2B will have its own FLI module. Also, the amount of computation required to perform the simulation is the same in both options because it is the same logic. However, the second option should be able to take advantage of the quad-core processor. By having three ModelSim processes, each process can execute concurrently in their own X86 core, leaving the fourth core for the X86 MPI software application.

To the best of our knowledge, ModelSim is not a parallel application nor takes advantage of multicore processors. But by virtue of using an implicit parallel programming model (MPI) to program FPGAs we can also split and run the simulation in parallel without ModelSim being a parallel program itself and without modifying its code.

To prove this hypothesis we simulate a simple test that consists of two hardware engines exchanging ping-pong (round trip) messages between two M2B modules, one hardware engine per module. One experiment will have both M2B modules in one ModelSim process and the second experiment will use two ModelSim processes, one per M2B. The MPI application contains three ranks, rank 0 (X86 processor) will send the test configuration parameters as messages as well as the "start test" message to ranks 1 and 2, which are the hardware engines. Then ranks 2 and 3 will exchange round trip messages of increasing size and when the test is complete they will inform rank 0 by sending a "done message." The actual messages will pass through the FLI modules and shared memory because ranks 1 and 2 are located in different stacks, and it is the same amount of logic to simulate.

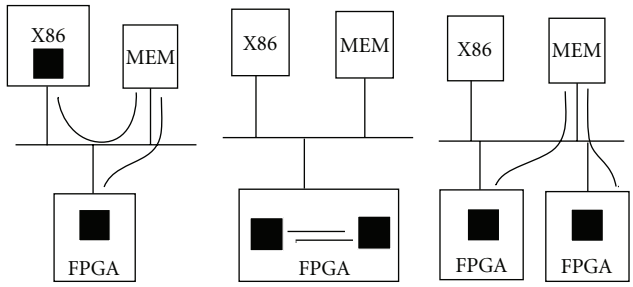
We measure the simulation time by using the MPI_Wtime() function, which returns the number of seconds elapsed since the application started to run, and obtain the time difference between the "start" and "stop" messages. The time measurement reports 20 seconds for the first option (both FPGAs in one ModelSim process) and 10 seconds for the concurrent simulation. As expected, the concurrent simulation is twofold faster than the single process option. We anticipate that for a three-stack system, the parallel simulation option would be three times faster than the all-in-one simulation option.

5. Flexibility and Architecture Exploration

One of the biggest advantages of abstracting and standardizing the communications using MPI is the flexibility to place the computing ranks in different places and using different types of computing elements. Ranks can initially be a processor (X86, PowerPC, MicroBlaze, etc.) or hardware engines, and the designer can change forth and back the type of the rank without having to change the code for the rest of the ranks. In other words, a rank can be a software process and then be replaced by a hardware engine with the same functionality.

Similarly, if a rank is implemented in an FPGA, the rank can be physically placed in one FPGA and later it can be moved to another FPGA without changing the code of any of the other ranks or the code of the rank that is being moved. From the programmer and designer perspective, the communication is being performed between two or more ranks, regardless of their location. The on-chip network will route the packets and make sure the messages arrive at their destinations.

Figure 7 shows the ping-pong example from Section 4.5 but the ranks are implemented in different ways. In Figure 7(a) the X86 processor is a software rank and it exchanges messages with a hardware rank. The ping-pong messages are exchanged through shared memory. In Figure 7(b) the X86 software rank is replaced by a hardware rank resulting in two hardware ranks implemented in the same FPGA. In this case the ping-pong messages are exchanged using only the on-chip network; there is no shared memory access. Finally, in Figure 7(c) one of the hardware ranks is moved to another FPGA and now the ping-pong messages are exchanged using shared memory. Note that in



(a) X86-M2B (b) Intra-M2B (c) Inter-M2B

Figure 7: Different ways two MPI ranks can be mapped and communicate.

this case the X86 processor is independent and does not intervene in the ping-pong messages between the hardware ranks. In all three scenarios the C and HDL code is the same, changing only configuration files that indicate the mapping of computing ranks to physical resources available.

The Message-Passing Simulation Framework allows the designer to simulate the three options in Figure 7 and perform this architectural exploration before running a potentially lengthy place and route processes for all the different combinations.

6. Case Example: LINPACK

This section presents a brief description of our LINPACK benchmark parallel implementation and some insights of the core and its communication interface. Keep in mind that we use this application to test the MPI_Bridge, the TMD-MPI software library, and the MSF FLI module, rather than to obtain peak performance in LINPACK.

6.1. LINPACK Implementation. The LINPACK Benchmark [10] is a widely used algorithm that measures floating-point computing performance by solving a system of linear equations, $\mathbf{Ax} = \mathbf{b}$. 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 [24] level 1 functions, *IDAMAX*, *DSCALI*, 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 4, 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, that is, 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.

6.2. The DGEFA Benchmark Hardware. The *DGEFA* computing engine, shown in Figure 8, 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 8 shows how the *DGEFA* engine connects to the MPE.

6.3. Verification of the Results. At the end of the application, rank 0 has the final matrix, which is compared against the sequential version of the code run only in software. 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 with precision because of cache effects, system load and memory traffic bring uncertainty about the exact memory transfer

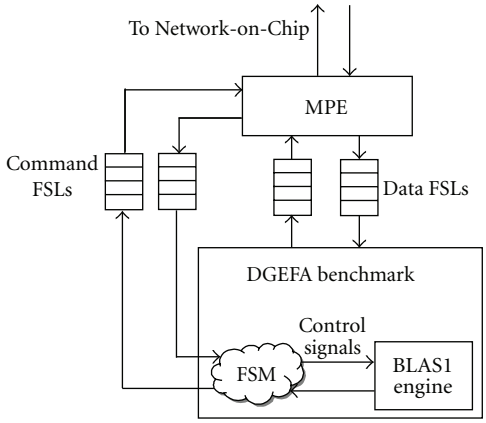


Figure 8: Interface between MPE and hardware engines.

latencies. The MSF module does not simulate the LLCC nor the FSB bus transactions; therefore, it is hard to predict the entire application's performance exactly. However, keep in mind that the focus of the MSF is correct functionality rather than accurate performance measurement. Nevertheless, we can estimate the LINPACK core performance based on measuring the most time consuming part of the application. By using the LINPACK function *second()* we know that the DAXPY loop takes 482 microseconds in the X86 processor (3.4 GHz), compared to 818 microseconds in the *DGEFA* engine (clocked at 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, MPI_Bridge, and the Xilinx FSB core) on the XC5VLX110 FPGA.

7. Future Work

Future work includes the support for external memory simulation models for those modules that have memory chips next to the FPGA; this will allow us to simulate designs with larger datasets because currently the hardware engines and embedded processors are limited to work with on-chip RAM. The MSF should be able to automatically generate top-level testbenches that include these external memory models.

Currently, the FLI module provides a one-clock read/write main memory latency, which is not realistic. By using Xilinx Chipscope in a placed and routed design we have been able to measure the latency of a memory read (MPI_FSB_Bridge requesting to read a memory location) to be between 40 and 50 clock cycles (approximately 150 nanoseconds) in an unloaded system. However, this changes in heavily loaded systems. Future releases of the MSF FLI module will include parameters or functions to introduce these latencies to further improve simulation accuracy, although changes in latency should not change the results of the application.

In this paper, only multiple M2B modules were simulated, but to simulate 15 FPGAs (including M2C modules) could be quite demanding on computing power, especially for large FPGA designs. Future work will include further performance measurements and optimizations to the MSF.

8. Conclusions

In this paper we describe a portable MPI-based approach for cosimulating multiple hardware engines implemented in FPGAs communicating with multiple X86 processes. Although, in this paper, we use an Intel-FSB-Xilinx-FPGA platform, the same concepts and ideas can be applied to other platforms.

Vendor-specific details should be hidden in a portable design and considered perhaps for further optimization. For initial stages of the design, a quick prototyping simulation environment such as the MSF can be very useful to accelerate the design flow and test the functionality as well as explore design alternatives. The MSF has demonstrated its usefulness during the development of a LINPACK system by allowing 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. The LINPACK system was cosimulated using six MPI ranks, half of them running as X86 software processes and the other half as hardware engines in simulation; all exchanging messages in a peer-to-peer fashion.

By virtue of using an implicit parallel programming model, the simulation of multiple FPGAs can also be distributed to multiple ModelSim processes and take advantage of multicore processors. We show that a simulation distributed across two ModelSim processes achieved a twofold speedup over a single ModelSim process simulation.

Similarly, latency tolerant designs are natural when the designer has an asynchronous, distributed, message-passing mind set. Therefore, communication latency between processors and FPGAs, although key for performance, does not need to be simulated to obtain a functionally correct system.

Glossary

ACP: Accelerated computing platform
FLI: Foreign language interface
FSB: Front side bus
FSL: Fast simplex link
HDL: Hardware description language
HPRC: High-performance reconfigurable computer
LLCC: Low-level communication core
LVDS: Low-voltage differential signal
M2B: M2 base
M2C: M2 compute
MPE: Message passing engine

MPI: Message passing interface
MSF: Message passing simulation framework
NoC: Network-on-chip
TMD: Toronto molecular dynamics.

Acknowledgments

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

References

[1] The MPI Forum, "MPI: a message passing interface," in *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '93)*, pp. 878–883, ACM Press, 1993.
[2] CRAY, Inc., http://www.cray.com.
[3] SGI, http://www.sgi.com.
[4] Intel, Corp., http://www.intel.com.
[5] Xtreme Data Inc., http://www.xtremedatainc.com.
[6] DRC computer, http://www.drccomputer.com.
[7] "General purpose reconfigurable computing systems," Tech. Rep., SRC Computers, Inc., 2005, http://www.srccomp.com.
[8] 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.
[9] M. Saldaña, E. Ramalho, and P. Chow, "A message-passing hardware/software cosimulation environment to aid in reconfigurable computing design using TMD-MPI," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 265–270, December 2008.
[10] A. Petitet, J. Dongarra, and P. Luszczek, "The LINPACK Benchmark: Past, Present and Future," http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf.
[11] 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*, Napa, Calif, USA, 2006.
[12] Amirix Systems, Inc., http://www.amirix.com.
[13] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.
[14] H. Hubert, *A survey of HW/SW cosimulation techniques and tools*, M.S. thesis, Royal Institute of Technology, Stockholm, Sweeden, June 1998.
[15] T. Suh, H.-H. S. Lee, S.-L. Lu, and J. Shen, "Initial observations of hardware/software co-simulation using FPGA in architecture research," in *Proceedings of the 2nd Workshop on Architecture Research Using FPGA Platforms (WARFP '06)*, February 2006.
[16] M. N. Wageeh, A. M. Wahba, A. M. Salem, and M. A. Sheirah, "FPGA based accelerator for functional simulation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '04)*, vol. 5, pp. 317–320, May 2004.
[17] Mentor Graphics, Corp., http://www.mentor.com.
[18] Cray Inc., "CRAY XD1 FPGA Development," pp. 9–11, pp. 63–66, 2005.
[19] SGI, "Reconfigurable Application-Specific Computing Users Guide," pp. 9–12, pp. 223–244, January 2008.
[20] Nallatech, http://www.nallatech.com.
[21] HyperTransport Consortium, http://www.hypertransport.org.
[22] Intel, "Intel Quick Path Architecture (White Paper)," http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf.
[23] Mentor Graphics, "ModelSim SE Foreign Language Interface Manual," February 2008.
[24] 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*, vol. 5, no. 3, pp. 308–323, 1979.