Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI

Manuel Saldaña, Daniel Nunes, Emanuel Ramalho and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
email: {msaldana,dnunes,eramalho,pc}@eecg.toronto.edu

Abstract

Recent research has shown that FPGAs have true potential to speedup demanding applications even further than what state-of-the art superscalar processors can do. The penalty is the loss of generality in the architecture, but reconfigurability of FPGAs allows them to be reprogrammed for other applications. Therefore, an efficient programming model and a flexible design flow are paramount for this technology to be more widely accepted. Furthermore, in the history of computers, standards have been a positive experience because they provide a common ground for research and development. A programming model for multiprocessor Systems-On-FPGAs should be standard and application independent, but optimized for a particular architecture.

In this paper, we use TMD-MPI, a subset implementation of the message passing standard MPI, and a flexible system-level design flow to implement heterogeneous Multiprocessor Systems-on-Chip on FPGAs. Hardware engines are also included into the multiprocessor system by using a message passing engine, which encapsulates the TMD-MPI functionality in hardware, to enable the communication between hardware engines and embedded processors. We test the functionality and scalability of the system by implementing a 45-processor system across five FPGAs. As a test example, we solve the heat equation by using the Jacobi iterations method. Some performance metrics are measured to demonstrate the impact of different computing cores on the overall computation.

1 Introduction

As the level of integration in modern silicon chips increases, the computing throughput requirements of new applications also increases. Currently, the limiting factor of modern processors is not the number of transistors-per-chip available but the wire delay and power consumption issues caused by high frequency operation. An alternative to alleviate this problem is the inclusion of multiple lower frequency, specialized computing cores per chip. For example, microprocessor manufacturers, such as Intel, Sun Microsystems, IBM and AMD have announced their multi-core microprocessors [9], although they incorporate only a few homogeneous cores.

Heterogeneous cores can provide further power savings by specialization [13]. Also, hardware optimization for a particular algorithm can significantly improve the execution speed with lower frequencies. Specialized computing cores can be designed and implemented in FPGAs and their flexibility allows programmers to configure them as needed. Some high-performance computer vendors are including FPGAs in their machines [5, 19, 21], and AMD's Opteron processors can be connected to FPGAs using the hypertransport protocol [6, 25]. FPGAs have evolved from just glue logic to embedded systems to entire Systems-on-Chip and recently to high-performance computing elements. Initiatives such as the FPGA High-Performance Computing Alliance [7] and OpenFPGA [15] are examples of research efforts to incorporate FPGAs into demanding computing applications.

The capacity of modern FPGAs and the interconnection of them provide enough resources to implement multiprocessor machines to exploit coarse-grain and fine-grain parallelism on a massive scale. However, fast communications, enough memory and fast memory access times, an efficient programming model, and a flexible hardware-software codesign flow are mandatory to be able to use all the resources effectively in multi-FPGA systems.

This paper extends previous work on TMD-MPI [18], which is a lightweight message passing programming model for embedded processors, to be used in heterogeneous processor systems. We show a number of multiprocessor configurations programmed to solve the heat equation, which is a typical parallel scientific application that

describes temperature change over time, given an initial temperature distribution. The processors and the communication links are both heterogeneous, but by using TMD-MPI the hardware complexities are abstracted from the programmer. The use of a standardized message-passing application program interface promotes code portability and the changes required to a C code that compiles for a Linux cluster are minimal in order to be compiled in our testbed system. To detect and understand possible bottlenecks in the scalability of the design flow, the programming model, and the Network-on-Chip infrastructure, we implement a 45-processor system across five FPGAs. We found that the limiting factor for creating the systems is the on-chip memory available and the intense computing power required to synthesize, place and route the designs, and not the design entry time. The focus of this work is on the design flow, which will help us to rapidly create prototypes of large systems that can be used to analyze performance bottlenecks.

The rest of this paper is organized as follows. Section 2 provides some background about concepts from previous work. Section 3 describes a multi-FPGA prototype system infrastructure. In Section 4, a multiprocessor system implemented in the prototype machine is explained. Section 5 describes the heat equation problem and how the solution is implemented in our multiprocessor testbed system. Section 6 presents some experiments with the system and the results obtained. Section 7 explores possible improvements and future work, and finally Section 8 provides some conclusions.

2 Background

In this section we review previous work on the programming model, design flow, on-chip/off-chip communications, and on the classification of computers relative to the use of FPGAs in the computation.

2.1 Classes of Machines

First, let us specify the scope of the paper by categorizing the types of high-performance machines; this categorization was first introduced in Patel [16]. As previously stated, FPGAs are now an active field of research in the HPC world, and based on the trend to incorporate FPGAs in computation, high-performance machines are classified into three categories. The first class of machines consists of current supercomputers and clusters of workstations without any involvement of FPGAs in the computation. Typical programming of such machines are coarse-grain parallel programming using message passing or shared memory libraries depending on how the memory is accessed by the CPUs.

The second class of computers are those that connect FPGAs to CPUs. Such machines can execute application kernels in a FPGA and take advantage of the bit level parallelism and specialized hardware computing engines implemented in the FPGA fabric. Coarse grain parallelism can be achieved by using a cluster of machines, each one with its own FPGA [5, 6, 11, 19, 21, 25]. The penalty in performance for Class 2 machines comes from the high latency communication between the CPUs and the FPGA. The programming model for these machines is based on libraries to send data to the FPGA to be processed and to receive the results; the FPGA acts as a co-processor.

The third class of machines is purely FPGA-based computers. These machines provide a tight integration between processors and hardware engines because processors can be embedded into the FPGA fabric. Although these embedded processors are not as sophisticated as the state-of-the-art superscalar processors, they can adequately perform control, I/O tasks and some computations. Time critical computations can be performed in special hardware engines that can exceed in performance the fastest processor available with less power consumption. Class 3 machines open the possibility of exploring heterogeneous computer architectures to more efficiently use the resources at hand by letting specialized hardware blocks do what they do best. Although Class 3 machines are application-specific in nature, reconfigurability is implicit in the FPGAs and they can be reconfigured as needed. Examples of Class 3 machines are BEE [3], TMD [16], and the one presented in Charles [2]. In this paper, we focus on Class 3 machines.

2.2 The Abstraction of Communications

In a Class 3 machine, a network infrastructure is required to enable the communications between computing elements. Three network tiers are identified and must be made to provide the view of a single, extensible FPGA fabric to the programmer [16]. The Tier 1 network is the network-on-chip inside each FPGA and it is used for intra-FPGA communications between processors and hardware blocks. This network is based on point-to-point unidirectional links implemented as FIFOs. This simplifies the design of hardware blocks and provides isolation between them, allowing the use of multiple clock domains by using asynchronous FIFOs. The FIFO is a powerful abstraction for on-chip communication, and more can be found in Williams [23]. Several FPGAs can be placed together in the same printed circuit board to form a cluster. The Tier 2 network would be for inter-FPGA communication on the same board using high-speed serial I/O links. The Tier 3 network is reserved for inter-cluster communication allowing the use of several clusters of FPGAs providing a massive amount of resources. On top of this network infrastructure, a program-

ming model can be developed to provide a homogeneous view of the entire system.

2.3 Programming Model

A major challenge in Class 3 machines is the design flow and the programming model. From the user perspective, a programming model should provide an efficient method for implementing applications while abstracting the hardware complexities. A programming model is defined by the nature of the application and by the architecture of the machine. Despite the implicit application-specific architecture of Class 3 machines, the programming model has to remain generic, but optimized, to be able to use it in other applications.

The two most common paradigms for parallel programming are message passing and shared memory. This work is focused on the viability of the message passing paradigm applied to System-on-Chip design; a study on parallel programming paradigms is beyond the scope of this paper. We use the message passing paradigm because it is well-known and our testbed system has distributed memory. Also, for a Multiprocessor System-on-Chip, shared memory over a shared bus does not scale well with the number of processors because of congestion in the bus. For distributed memory machines, message passing has proven to be more scalable because memory and communications are not shared. MPI [22] is the *de facto* message passing API for programming scientific applications and enables code portability by standardizing the syntax and semantics of the programming interface. MPI was originally developed in a joint effort between academia and industry, and it is used extensively in the high-performance community.

The idea of using message passing for Multiprocessor System-on-Chip can be found in recent research [8, 12, 17] and working groups such as the T-GENAPI from OpenFPGA [15], and the Message passing and resource management working group from the Multicore Association [14]. MPI is appropriate for Class 3 machines because it provides a generic application-independent interface, and does not specify an implementation style or underlying technology. The MPI implementation is entirely machine dependent, but not the application program interface, in which the functionality and syntax are standardized. This enables machine vendors to provide an MPI implementation optimized for a particular architecture. The entire MPI standard is meant originally to be used for high-end computers with abundant resources. However, not all the functionality established in the standard is required to code a parallel application. TMD-MPI [18] is a lightweight subset of MPI designed for embedded systems with scarce memory restrictions. TMD-MPI does not require an operating system, has a small memory footprint, and represents a low overhead for the processor. TMD-MPI is modular and designed in layers, such that it is easy to port to other platforms. Programmers familiar with MPI can develop applications using TMD-MPI without having to learn new APIs. The role of TMD-MPI is two fold; it provides interprocess communication and it enables the design flow to gradually map high-level descriptions of an application into hardware.

2.4 Design Flow

The design flow is the set of steps to follow to map an application to a particular type of architecture. In Class 1 machines the design flow is relatively simple because only software is involved; just compile and link libraries to create the final executable code. In contrast, Class 2 and Class 3 machines involve software and hardware design flows. In particular, for Class 3 machines, the boundary between hardware and software becomes vague and functionality can be implemented in both levels; it becomes a tradeoff between efficiency and flexibility.

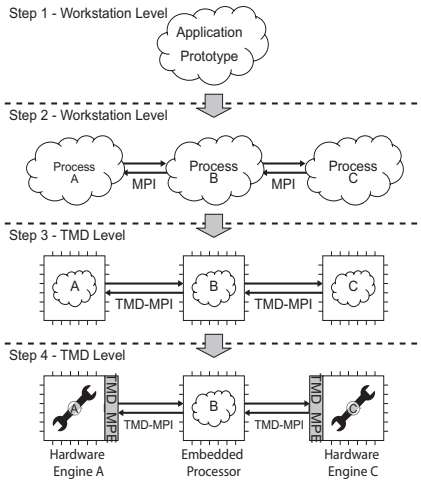

Figure 1. Multi-FPGA Design Flow

Figure 1 shows the design flow for our Class 3 machine [16]. In Step 1, the flow starts by creating the entire application in a standard sequential C/C++ program in a Linux workstation. In Step 2, the application is parallelized using a well known MPI distribution, such as MPICH [10]; this is still at the workstation level. In Step 3, the parallel version of the application is ported to the Class 3 machine using purely embedded processors. At this point, the MPI implementation used is no longer MPICH, but TMD-MPI. Because MPICH and TMD-MPI are using the same standard API, this step should be straightforward. Finally, in Step 4, those embedded processors that are computing the most demanding part of the code can be substituted by faster hardware computing blocks. The hardware blocks

will have to use a message passing engine (TMD-MPE), which contains the TMD-MPI functionality in hardware, to allow hardware blocks to communicate with embedded processors. Moreover, the TMD_MPE can also be used by the processors to enable more efficient the communications by relieving the processor from handling the message passing algorithms.

3 A Class 3 Machine Testbed

This section describes the Class 3 machine testbed system that we use to experiment with the programming model and the design flow to create the multiprocessor designs. First, we describe the hardware available on the testbed, followed by a description of the software used.

3.1 Hardware Infrastructure

Our Class 3 machine prototype is shown in Figure 2. We use five Amirix AP1100 PCI development boards [1]. Each board has a Xilinx 2VP100 FPGA [24] with 128 MB of DDR RAM, 4 MB of SRAM, support for high-speed serial communication channels, two RS232 serial ports, and an Ethernet connector. Based on a specific jumper setup, the FPGA can be configured from the on-board configuration flash memory, a compact flash card or from a JTAG interface. We use the on-board configuration flash memory as it can be accessed through the PCI bus.

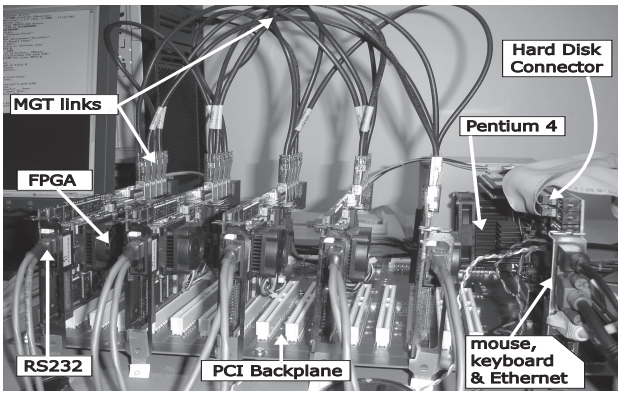

Figure 2. A Class 3 machine testbed

Each FPGA board is connected to a 16-slot PCI backplane, which provides power to the FPGA boards and it is also used to access the configuration flash on each board. Additional to the FPGA boards, there is a Pentium 4 card plugged into the backplane. However, the Pentium processor is used only for the FPGA configuration and to provide access to I/O peripherals, such as keyboard, mouse, monitor, and a hard disk drive. The Pentium card is running Linux and is also used as a terminal to print out all the standard output from the FPGAs.

The boards are interconnected by a fully-connected topology of Multi-Gigabit Transceiver links (MGTs), which are high-speed serial links [24]. Each MGT channel is configured to achieve a full-duplex bandwidth up to 2.5Gbps. The MGT links form the Tier 2 network for inter-FPGA communications. Although the infrastructure is present for a Tier 3 network, its use is scheduled for future work. Xilinx 2VP100 FPGAs have 20 MGT channels available, but only five boards are used because only four MGT channels have direct connectors to the board. More MGTs are available through an expansion card, which is not used in this work.

3.2 Software Infrastructure

In this work, we use the Xilinx EDK/ISE 7.1i suite of tools to design the multiprocessor system. For multiprocessor systems with a small number of nodes, the actual design entry can be done manually directly coding in the EDK integrated environment. It is especially easy for homogeneous multiprocessors where the specification of one processor can be copied and pasted a few times, and then the particular parameters for each processor can be modified. However, for larger systems, the multiprocessor specification becomes error prone due to the number of connections in the system. The use of an automated tool is recommended, such as the System Generator [20], which is a CAD tool that, based on a graph description of a multiprocessor system, generates EDK hardware and software specification files, as well as some auxiliary files for EDK. After the system is generated, particular parameters of each core can be modified.

The design of multiprocessor systems across multiple FPGAs requires considerable computing power to synthesize, place and route. A 40 MHz, 9-processor project takes approximately 1.5 hours to be generated in a 3 GHz Pentium Xeon computer. The design could be generated with higher frequencies but the time it takes multiplied by the number of boards make it extremely time consuming for an architectural study, as in this paper, where changes to the hardware requires frequent regenerations of the entire system. For this reason, a small cluster of workstations is used to generate the designs. Each FPGA board in the system has its own EDK project, and the EDK projects are distributed among the computers in the cluster. The cluster is used to generate the bitstreams only; the actual configuration of the FPGAs is done by the Pentium 4 card in the PCI backplane.

To configure the FPGA using the on-board configuration flash, a PROM binary file is required. For this, we use the Xilinx *promgen* utility, which generates PROM memory files from a bitstream file. The binary files generated by the

cluster are transferred by FTP to the Pentium 4 card. Scripts on the Pentium 4 card automate the configuration process of each FPGA using the *Apcontrol* program [1], which is a command line interface application developed by Amirix to access the on-board configuration Flash through the PCI bus.

In such a distributed environment, it is convenient to have a centralized repository for the source code of the processors; otherwise, there would be as many source files as projects. Without a centralized scheme, a change in the code of one library would have to be replicated for all the projects increasing the risk of source-code version problems. This scheme is adequate for this particular design example in which we use a Single-Program-Multiple-Data (SPMD) programming paradigm to exploit data parallelism. However, other paradigms such as Multiple-Program-Multiple-Data (MPMD) to implement functional and data parallelism can also be used.

4 A Multiprocessor System in a Class 3 Machine

At this point, we have described our Class 3 machine testbed and how it is programmed. In this section, we show what is inside of each FPGA. We first present the different types of processing units and network components, followed by an introduction to the message passing engine, and later we describe the different multiprocessor configurations.

4.1 The Processing Elements

Three different processing units were used in this paper. First, we use the Xilinx MicroBlaze soft-processor, which is a processor implemented in the FPGA fabric. We can instantiate up to 13 MicroBlaze processors per FPGA, if they use 64KB of internal memory and no other logic requires internal memory blocks. The second processing unit is an IBM PowerPC405, which is a processor embedded in the FPGA chip; each 2VP100 FPGA has two PowerPCs. The third type of processing unit is a Jacobi hardware engine designed to compute the Jacobi iterations algorithm [26]. The description of the algorithm is presented in Section 5.

Each processing unit has its own local memory. The MicroBlaze has a single dual-port RAM of 64KB for code and data. The PowerPC has two independent memory blocks, 32KB for code and 32KB for data. For both processors, the application code and the TMD-MPI code are both located in the code section of the memory. The Jacobi hardware engine has two 64KB dual-port memories to store only data, because the algorithm is implemented as a state machine. Although the Amirix FPGA boards have external memory, they are not used in this work.

The MicroBlaze and the PowerPC405 have inherently different architectures. The PowerPC405 has a 5-stage pipeline, branch prediction logic, and no floating point unit (FPU); and the MicroBlaze V.4 has a 3-stage pipeline and an optional FPU. This makes the PowerPC405 faster for control sections of the code, for example in array initializations, but it has to emulate in software the FPU operations, which degrades its performance. In contrast, the MicroBlaze is not as efficient as the PowerPC405 for control operations, but if the FPU is enabled, the MicroBlaze is 20x faster in the math section of the code than the PowerPC405 at the same clock frequency. Although the PPC405 supports faster clock frequencies than the MicroBlaze, for simplicity of the design, both are running at the same frequency. Recall that the focus of this study is on the functionality of communications, the programming model and the design flow rather than to achieve peak performance, which will be addressed in future work by optimizing all the algorithms and hardware blocks, and using the maximum frequency possible.

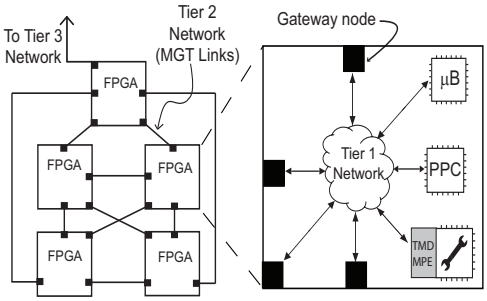

Figure 3. Multiprocessor System on Multiple FPGAs

4.2 The Network Components

Each board in Figure 3 has four gateway points to communicate with other FPGAs, one gateway per FPGA. The gateway point is the interface between the Tier 1 network and the Tier 2 network. Both networks have different protocols, and as a consequence, a network *bridge* block is included in the gateway to translate network packets on-the-fly as they go off-chip and on-chip again. The off-chip point-to-point communication using the MGT links is performed by a hardware block called the Off-Chip Communication Controller (OCCC) [4]. The OCCC provides a reliable link between FPGAs, it detects transmission errors and requests a packet retransmission, if needed.

In this heterogeneous multiprocessor system, we define a *node* as an element connected to a Tier 1 network. It can be a MicroBlaze, a PowerPC405, a hardware engine, or a gateway point. All the nodes are connected to a Tier 1 network

interface block (NetIf), as can be seen in Figure 4. The NetIf has the responsibility to route all the packets in the network based on a unique identification number for each processor and a routing information table. The NetIf in a gateway point can detect when a packet is not local to the FPGA and forward it to a remote FPGA, if the packet's destination field corresponds to the range of processor IDs in the remote FPGA.

The FIFOs used to connect all the blocks in a Tier 1 network are Xilinx FSLs [24]. Each FSL is a 32-bit wide, 16-word deep FIFO. The depth of the FSL has an impact on performance for the Tier 1 network because deeper FSLs would increase the buffering capacity of the network and decrease the number of FSLs in the full condition. However, in this paper we use the same depth for all the FSLs and a more detailed study of this parameter remains as future work.

Each MicroBlaze soft-processor has eight FSL channels that provide a direct connection to its respective NetIf block using an FSL. In contrast, the PowerPC405 in the 2VP100 FPGA does not have an FSL interface and an additional interface block was developed to allow the PowerPC405 to be connected to the Tier 1 network. This additional block, called *dcr2fsl*, is a bridge between the PowerPC405 DCR bus, which is a high-speed local bus, and the FSL interface. The dcr2fsl block is shown in Figure 4.

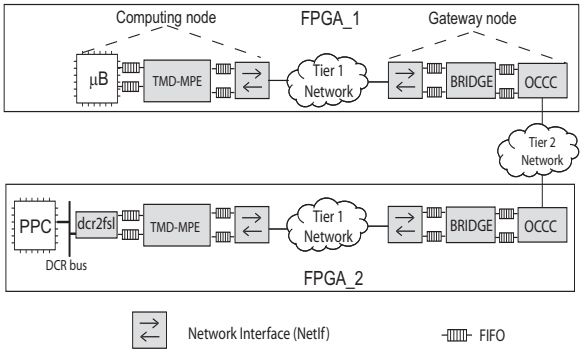

Figure 4. Path of a packet from one FPGA to another

Figure 4 shows the path that a packet follows in a typical *send* operation from one processor in one FPGA to another processor in another FPGA. First, the MicroBlaze sends the entire message to the TMD-MPE, which will split the message into smaller packets of data adding the proper headers. Each packet is sent to the NetIf, which will route the packets to the NetIf of the destination node. In this case, the destination node is the gateway node because the message is not local to the FPGA in which the MicroBlaze is located. The packets are translated into a Tier 2 network packet format by the outgoing bridge and sent to the OCCC. The OCCC will manage the off-chip communication with the receiving OCCC. Once the packets are on the other FPGA, the incoming bridge will translate the Tier 2 network packets to a Tier 1 network packet format again. The NetIf on the destination gateway will route the packets to the PowerPC's NetIf, which will pass the received packets to the dcr2fsl block, and finally the PowerPC will read the packets through the dcr bus.

4.3 The Message Passing Engine

Transferring the TMD-MPI functionality into a hardware block provides considerable benefits to improve the overall performance of the system. The TMD-MPE is a message passing engine that reduces latency, increases the bandwidth, and relieves the processor from handling the message passing protocol and algorithms. TMD-MPE provides support for handling unexpected messages, handles the communication protocol, and divides large messages into smaller size packets to be sent through the network. As shown in Figure 4, the TMD-MPE is connected between the computing element (processor or hardware engine) and the network interface (NetIf). The TMD-MPE, will receive the message parameters and data from the computing element. These parameters are the operation (whether it is sending or receiving a message), the destination node id (rank of the process in the MPI environment), the length of the message, and an identification number for the message (the tag parameter in a normal MPI send operation). From this point, the TMD-MPE will handle the communications through the network with the destination TMD-MPE or with the destination embedded processor, assuming it is running TMD-MPI.

However, three factors limit the advantages of using the TMD-MPE with embedded processors. The first factor is the memory access time of the processor when reading or writing data to be sent or received. The second factor is the access time to the FSL link. In the case of the MicroBlaze, the FSL access is approximately 3 cycles, for constant values (no memory access) because the FSL channel is built into the architecture. In contrast, the PowerPC405 has an extra delay when sending or receiving data because the information has to pass through the DCR bus and the dcr2fsl bridge. The third factor is the implicit sequential execution of instructions in a normal processor. Conversely, a hardware engine can execute operations in parallel considerably faster. The most interesting feature of TMD-MPE is that it allows specialized hardware computing engines to be connected to the Tier 1 network and send/receive messages from other nodes in the system. Using a hardware engine instead of a processor as a computing element exploits more efficiently the TMD-MPE's potential.

4.4 Computing Node Configurations

Based on the the type of computing element and whether the TMD-MPE is used, different configurations can be formed. Figure 5 shows the possible combinations for a computing node. As previously explained, the PowerPC405 requires the dcr2fsl adapter to use FSLs. The MicroBlaze and PowerPCs that have the TMD-MPE, require a lightweight version of TMD-MPI because much of the functionality is now in hardware; the processors that do not have the TMD-MPE, require a full TMD-MPI version. The hardware engines will always require the TMD-MPE to be able to connect to the network. The version of TMD-MPI used by the processors is defined at compile time by declaring a constant in the command line of the compiler.

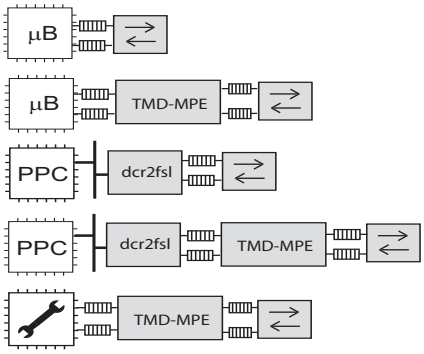

Figure 5. Different node configurations based on the use, or not, of TMD-MPE

5 The Heat Equation Application

To test the functionality of the system, an example application is presented in this section. This application is the heat equation, which is a partial differential equation that describes the temperature change over time, given a specific region, initial temperature distribution and boundary conditions. The thermal distribution is determined by the Laplace equation $\nabla(x, y) = 0$, and can be solved by the Jacobi iterations method [26], which is a numerical method to solve a system of linear equations. It is not the goal of this paper to develop the best possible algorithm to solve the heat equation. This method was chosen for its simplicity and because it requires communication among the processors to perform the computation. This example application is meant to prove that the multiprocessor system in our Class 3 machine works by doing a meaningful computation.

The basic Jacobi algorithm is to solve iteratively Equation 1, where u is the matrix with the temperatures in a given iteration step and v is the matrix with the temperatures for the next iteration. In every iteration, the values of u have to be updated with the values of v. The convergence condition is computed as the square root of the sum of the square differences between the old values and the new values as shown in Equation 2.

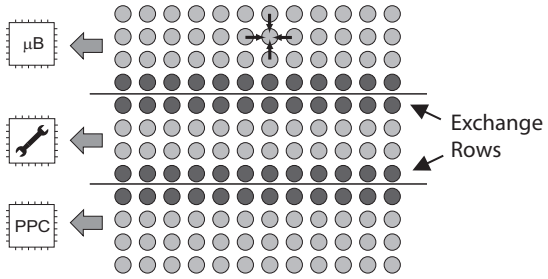

Figure 6. Data decomposition and point-to-point communication in Jacobi algorithm

$$v_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}}{4} \quad (1)$$

$$\sqrt{\sum_{i,j}(u_{i,j} - v_{i,j})^2} < \epsilon \quad (2)$$

The basic parallel algorithm for Jacobi is to split the matrix into smaller, equal-size matrices as shown in Figure 6. Each processor would have to exchange the border rows with its immediate neighbors to compute the limiting rows in each section. The steps in the algorithm are:

Step 1. Send to every node the section of data to process

Step 2. Exchange rows with neighbors

Step 3. Perform computation

Step 4. Compute the convergence data

Step 5. Reduce the convergence data and send it to the master

Step 6. Receive from master convergence signal, go to step 2 if data has not converged

To program the Jacobi iterations method we follow the programming flow described in Section 2.4. The changes in the parallel C code from the workstation to our testbed are minimal. Based on the type of processor used, conditional compilation macros are coded to include the proper header files for the MicroBlaze and the PowerPC405 processors generated by the EDK; the actual MPI code remains untouched. We validate the solution of the heat equation by comparing the final results of our Class 3 machine with the results of the sequential version of the code running on the Pentium processor.

6 Experiments and Results

In this section we conduct two experiments to test the design flow, the programming model and the network infrastructure running the Jacobi iterations method in different heterogeneous multiprocessor systems. Also, these tests provide an initial reference to compare the performance with further improvements to TMD-MPI, TMD-MPE and the network components.

6.1 Performance Test

In this first test, we have a fixed problem size, which is a square region of 60x60 elements, and we measure the impact on performance of heterogeneous cores. We show that by using our design flow and TMD-MPI, we can create and program a number of multiprocessor configurations. For this objective, we use only one FPGA with nine processing units. There are a considerable number of possible combinations for this experiment based on the configuration of the computing nodes, as explained in Section 4.4. We use only a subset of the possible combinations; those that we believe are the most representative. Table 1 summarizes the different configurations for this experiment. The *#μB, #PPC* and *#Jacobi Engines* columns are the number of MicroBlaze processors, the number of PowerPC405 processors, and the number of Jacobi hardware engines, respectively. The *FPU for μB* column indicates whether the floating point unit is enabled in the MicroBlaze. Finally, the *Experiment ID* column is a label to identify the experiment.

Table 1. Experiment configurations

#μB	#PPC	#Jacobi Engines	FPU for μB	Experiment ID
9	0	0	no	*9uB*
9	0	0	yes	*9uB_FPU*
7	2	0	yes	*7uB_FPU_2PPC*
1	0	8	yes	*1uB_FPU_8HW*
4	2	3	yes	*4uB_FPU_2PPC_3HW*

Figure 7 shows the main-loop execution time of each configuration in Table 1. The initialization of the square region is not considered because the time spent in that part of the code is not significant. The y-axis has a log scale because the differences in execution times vary considerably between configurations. In this experiment, a MicroBlaze with FPU unit enabled, achieves 60x faster execution times than a MicroBlaze without FPU, and 20x faster execution times than a PowerPC405 because of the overhead of software emulation of the floating point operations. The impact of this can be seen in configuration *9uB*, which is slow compared to the other FPU-enabled configurations, as expected. For configuration *4uB_FPU_2PPC_3HW*, three MicroBlazes are substituted by Jacobi hardware engines. The Jacobi hardware engines use the TMD-MPE to connect them with the MicroBlazes and PowerPCs. The execution time decreases gradually for the first seven nodes. However, only the first four nodes are MicroBlaze processors, the next three are the Jacobi hardware engines, but there is no improvement in performance because the MicroBlazes are considerably slower than the Jacobi hardware engines forcing them into an idle state while waiting for the stop message coming from the first MicroBlaze. The performance degrades even further when nodes eight and nine are used. These nodes are PowerPC processors without FPU support, which causes a drastic increase in the execution time. A similar situation happens with configuration *7uB_FPU_2PPC* because the PowerPC is slower than the MicroBlaze with FPU. For clarity, the plot for configuration *7uB_FPU_2PPC* is not included in Figure 7 since it is identical to the plot of configuration *4uB_FPU_2PPC_3HW*. Finally, in configuration *1uB_FPU_8HW*, eight Microblazes are substituted by hardware engines, but the improvement in performance is marginal compared to the *9uB_FPU* configuration, because the first Microblaze is still part of the computation, which slows down the Jacobi hardware engines. In this case, a master-slave approach, in which the MicroBlaze is not part of the computation and only collects the convergence data and sends the stop message would produce faster execution times for configuration *1uB_FPU_8HW*.

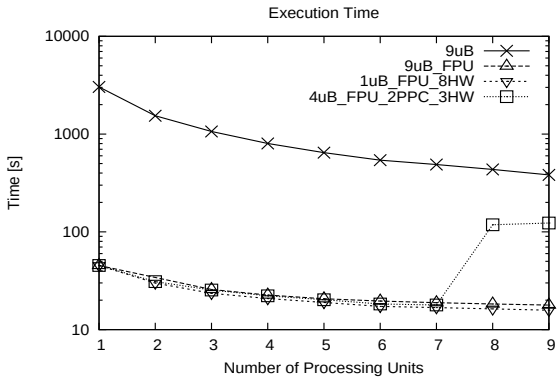

Figure 7. Main loop execution time of different Multiprocessor configurations

6.2 Scalability Experiment

In this experiment, we test the scalability of the system by using the five boards available. Although each FPGA board can have a different MPSoC configuration, we use the same in all the FPGAs for simplicity. The changes from one FPGA to another are the network routing table and a constant defined at compile time for each processor to define

the MPI rank. We use seven MicroBlazes and two PowerPC405 in each board. None of the processors are using TMD-MPE and the MicroBlaze FPU unit is disabled. There is a total of 45 processors, limited by the amount of internal RAM in the FPGA. In a fixed-size problem, the region to be computed has to be the same size regardless of the number of processors. Therefore, we need a region with enough rows for each one of the 45 processors and still big enough to fit in the memory of a single processor. A narrow region of 240 rows by 16 columns is used; a larger region will not fit into the memory of a single processor. For one processor, there will be a computation of 240 rows and there will not be communication. For 45 processors, each node will compute only five rows on average and exchange two rows of 16 data values per iteration.

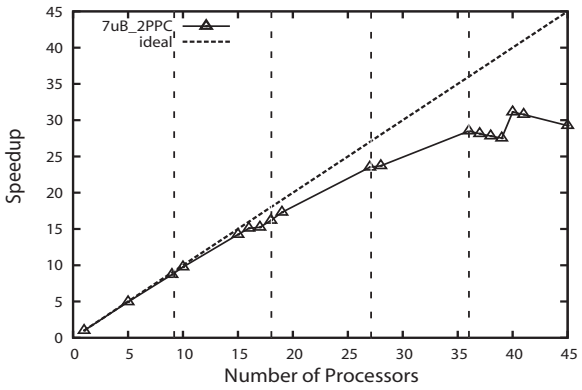

Figure 8. Speedup of Jacobi iterations in the 45-node multiprocessor system

Figure 8 shows the speedup of the execution of the Jacobi algorithm for 45 processors. The irregularities of the plot are due to the nonlinearities of the network interface's scheduling algorithm and the heterogeneity of the computing cores and communication links. The effect of using multiple boards connected by the MGT links produces a slight change in the slope of the plot. The vertical dashed lines show when there is an addition of a new board. For this particular architecture-algorithm pair, we found a sustained speedup up to 35 processors. There is an inflection point around 40 processors representing a peak performance-point after which adding more processors will only slow down the execution of the application.

7 Future Work

With this working Class 3 machine testbed we will focus on optimization and improvement of the architecture and the design flow. In this section we present some of the ideas for future versions of the system. We will experiment with different algorithms under a more controlled environment. We can modify the software and hardware, and study the impact on performance.

We intend to use this platform to optimize the TMD-MPI algorithms; for example, how to implement more efficient collective operation algorithms, such as tree-based reduction instead of the basic linear algorithm that is currently implemented. Also, TMD-MPI uses only the rendezvous protocol, which saves memory and avoids buffer overflow problems for large messages, but is inefficient for short messages. To alleviate this limitation, we plan to implement a hybrid scheme including the eager protocol or buffered scheme for short messages, and the rendezvous protocol for large messages.

Currently, the TMD-MPE only performs the basic send and receive operations, and collective operations are still coordinated by the computing element that the MPE is connected to. Future versions of the MPE should handle the collective operations as well. Also, a DMA version of the TMD-MPE would eliminate the overhead for the processor of copying the data to the FSL.

Similarly, improvements of the hardware blocks that form the Tier 1 network are also scheduled for future work, such as, better scheduling and routing algorithms in the NetIf block. One caveat of our system is that to include a new node in the multiprocessor system, all the boards in the entire system have to be synthesized, placed and routed again, which is time-consuming, because the routing tables of the NetIf and some links are added to the system. To solve this problem, we will investigate incremental synthesis, partial reconfiguration or software configuration of the routing table using an embedded processor.

In terms of the design flow, we are working on CAD tools to develop multiprocessor systems for multi-FPGA platforms to help with the layout of the architecture, the mapping of code into processors, and to automatically generate the design files for the FPGA design tools. Finally, we will concentrate our efforts on the implementation of molecular dynamics simulation on this prototype as it is a demanding application with a low communication-to-computation ratio, in which a Class 3 machine, such as the one presented in this work can provide a significant speedup.

8 Conclusions

In this work, we presented a heterogeneous multiprocessor system implemented in a multi-FPGA architecture. We implemented the Jacobi iterations method to solve the two-dimensional Laplace equation in a 45 embedded processor system. A sustained increase in the speedup of the application up to 40 processors shows that the system is scalable and limited by the amount of on-chip memory and on-chip

resources, but not by the communications.

The standard API used by TMD-MPI made it easy to port the application from a Linux cluster to the multi-FPGA system by just including the proper header files for the MicroBlaze and the PowerPC405. The rest of the code remained the same. By using the design flow explained in this paper, the substitution of some MicroBlazes by hardware engines was transparent for the rest of the processors in the system. This level of abstraction and isolation from the actual implementation provides great flexibility to prototype hardware engines and test them in a real multiprocessor architecture on-chip and not in simulation. It would be difficult and time consuming to simulate 45 processors.

Finally, with the design flow methodology used, the most time consuming task to adding more processors to the system is not the design entry but the synthesis, place and route processes; again, it is a limitation on computing power.

Acknowledgments

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the funding provided for this project. CONACYT in Mexico provided funding to Manuel Saldaña. Thanks to Amirix for the help with their hardware and tools.

References

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

[2] C. L. Cathey, J. D. Bakos, and D. A. Buell. A Reconfigurable Distributed Computing Fabric Exploiting Multilevel Paralellism. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. IEEE Computer Society Press, 2006.

[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] C. Comis. A high-speed inter-process communication architecture for FPGA-based hardware acceleration of molecular dynamics. Master's thesis, University of Toronto, 2005.

[5] Cray XD1 supercomputer for reconfigurable computing. Technical report, Cray, Inc., 2005. `http://www.cray.com/downloads/FPGADatasheet.pdf`.

[6] DRC computer. `http://www.drccomputer.com/`.

[7] FPGA High Performance Computing Alliance. `http://www.fhpca.org/`. Curr. Jan. 2006.

[8] P. Francesco, P. Antonio, and P. Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 736–741, Washington, DC, USA, 2005. IEEE Computer Society Press.

[9] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, sep 1996.

[11] T. Hamada, T. Fukushige, A. Kawai, and J. Makino. PROGRAPE-1: A Programmable Special-Purpose Computer for Many-Body Simulations. In *FCCM*, pages 256–257, 1998.

[12] A. Jerraya and W. Wolf, editors. *Multiprocessor Systems-on-Chip*. Morgan Kaufmannn, 2004.

[13] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, 2005.

[14] Message passing and resouerce management working group. Technical report, Multicore Association, (Accessed: April 2006). `http://www.multicore-association.org/workgroup/ComAPI.html`.

[15] OpenFPGA - Defining Reconfigurable Supercomputing. `http://www.openfpga.org/`. Curr. Jan. 2006.

[16] 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.

[17] P. G. Paulin et al. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53, New York, NY, USA, 2004. ACM Press.

[18] 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.

[19] Extraordinary Acceleration of Workflows with Reconfigurable Application-specific Computing from SGI. Technical report, Silicon Graphics, Inc., Nov. 2004. `http://www.sgi.com/pdfs/3721.pdf`.

[20] L. Shannon and P. Chow. Maximizing System Performance: Using Reconfigurability to Monitor System Communications. In *International Conference on Field-Programmable Technology (FPT)*, pages 231–238, Brisbane, Australia, Dec. 2004.

[21] General Purpose Reconfigurable Computing Systems. Technical report, SRC Computers, Inc., 2005. `http://www.srccomp.com/`.

[22] 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.

[23] J. A. Williams and X. X. N. W. Bergmann. FIFO Communication Models in Operating Systems for Reconfigurable Computing. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 277–278. IEEE Computer Society Press, 2005.

[24] Xilinx, Inc. `http://www.xilinx.com`.

[25] Xtreme Data Inc. `http://www.xtremedatainc.com/`.

[26] J. Zhu, editor. *Solving Partial Differential Equations on Parallel Computers*. World Scientific, 1994.