A Scalable FPGA-based Multiprocessor

Arun Patel[1], Christopher A. Madill[2], Manuel Saldaña[1], Christopher Comis[1,*]
Régis Pomès[2], and Paul Chow[1]
[1]Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada M5S 3G4
[2]Structural Biology and Biochemistry, The Hospital for Sick Children, Toronto, ON, Canada M5G 1X8,
and the Department of Biochemistry, University of Toronto
{apatel, msaldana, pc}@eecg.toronto.edu, {cmadill, pomes}@sickkids.ca

ABSTRACT

It has been shown that a small number of FPGAs can significantly accelerate certain computing tasks by up to two or three orders of magnitude. However, particularly intensive large-scale computing applications, such as molecular dynamics simulations of biological systems, underscore the need for even greater speedups to address relevant length and time scales.

In this work, we propose an architecture for a scalable computing machine built entirely using FPGA computing nodes. The machine enables designers to implement large-scale computing applications using a heterogeneous combination of hardware accelerators and embedded microprocessors spread across many FPGAs, all interconnected by a flexible communication network. Parallelism at multiple levels of granularity within an application can be exploited to obtain the maximum computational throughput. By focusing on applications that exhibit a high computation-to-communication ratio, we narrow the extent of this investigation to the development of a suitable communication infrastructure for our machine, as well as an appropriate programming model and design flow for implementing applications.

By providing a simple, abstracted communication interface with the objective of being able to scale to thousands of FPGA nodes, the proposed architecture appears to the programmer as a unified, extensible FPGA fabric. A programming model based on the MPI message-passing standard is also presented as a means for partitioning an application into independent computing tasks that can be implemented on our architecture. Finally, we demonstrate the first use of our design flow by developing a simple molecular dynamics simulation application for the proposed machine, which runs on a small platform of development boards.

1. INTRODUCTION

Most of the FPGA-based accelerators built or proposed to date either significantly improve the computational throughput of a serial algorithm, or perform multiple iterations of a homogenous, data-parallel computation simultaneously. In either case, the problems considered require at most a handful of FPGAs to implement. These approaches still cannot address intensive, large-scale computing applications that require much greater speedups.

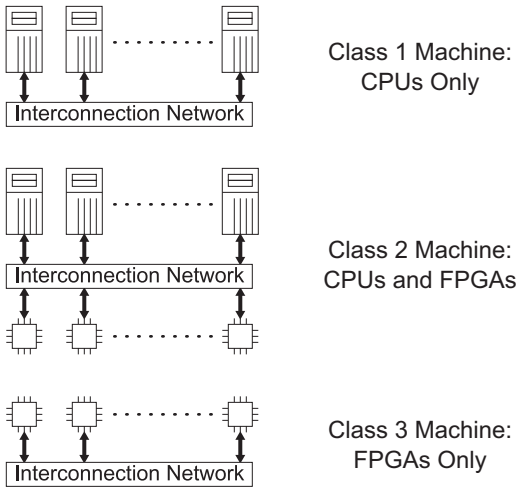

Figure 1: Three Classes of Computing Machines

In this paper, we propose an architecture and design flow capable of scaling from just a few FPGAs up to thousands. The approach is also able to support a heterogenous collection of hardware accelerators and embedded microprocessors, making it more suitable for implementing very large-scale applications.

We begin by categorizing today's high-performance computing machines into three classes as depicted in Figure 1. The first class consists of present-day supercomputers based on CPU clusters. Computing tasks are partitioned coarsely such that they are amenable to execution on multiple CPU nodes. The second class is comprised of CPU-based clusters that incorporate FPGA hardware for acceleration purposes. In this model, the FPGA is largely a slave device that acts under the direction of a processor node. We will focus on the third class of computing machines, a machine constructed solely of FPGA hardware nodes.

The purpose of this project is to establish the viability of Class 3 architectures by demonstrating a significant application speedup. Our eventual goal is to create a novel high-performance computing platform. This paper presents the TMD[1], an implementation of a Class 3 architecture.

*Now with PMC Sierra, Inc.

[1]The acronym *TMD* originally meant the *Toronto Molecular Dynamics* machine, but this definition was rescinded as the platform is not limited to Molecular Dynamics. We kept the name in homage to earlier TM-series projects at the UofT.

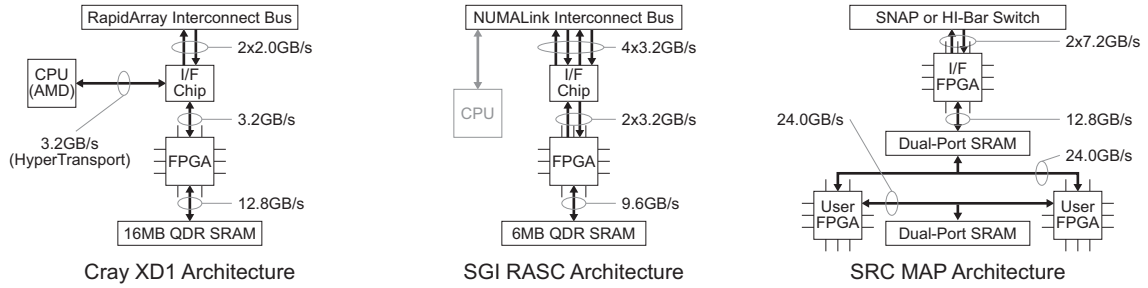

Figure 2: Implementations of Class 2 High-Performance Computing Machines [11], [28], [31]

The modern supercomputing paradigm integrates copious amounts of memory, I/O bandwidth, communication bandwidth, and processing power. Addressing all four of these requirements simultaneously with a new architecture is a subject for much further research. We are concentrating our investigation on the subset of applications that do not require vast amounts of shared memory or tremendous I/O bandwidth, namely applications that exhibit high compute-to-communication ratios. Since the breadth of the study has been pared down, we can focus on the issues of distributed control and communication and developing an appropriate programming model for such a system. A preliminary flow for translating multithreaded software descriptions into a collection of hardware netlists is presented, along with a sample application based on this flow.

The design of this system is inspired by molecular dynamics (MD) simulations, a highly-parallelizable n-body problem with computational complexity of $O(n^2)$. There are two dominant types of calculations that constitute over 99% of the computational effort in MD, each requiring a different hardware accelerator structure. Developing a working MD simulator that scales with our proposed architecture and provides orders of magnitude in speedups is the first application for this machine.

The TMD architecture can also be used to solve many other computing challenges. The reconfigurability of its constituent elements allows the platform to target a variety of applications, such as finite element analysis, optical simulation, and weather prediction, as well as MD. Once we have successfully developed the MD simulator, our focus will shift to automating the development process for TMD applications.

The first half of this paper concentrates on the proposed architecture, beginning with a survey of related work in the next section followed by a technical overview of the system in Section 3. The second half of the paper focuses on the programming model, starting with a discussion of the requirements in Section 4, followed by a description of a MPI-based message passing mechanism implementation in Section 5. A preliminary implementation of a molecular dynamics simulation system is outlined in Section 6 and we conclude and present avenues for future work in Section 7.

2. RELATED WORK

Although many of the design concepts used in the TMD have architectural underpinnings in supercomputer technology, the degree of parallelism we hope to exploit with the TMD is much finer than that of threads and processes executing on CPUs. This section gives a review of existing architectures that have been designed to address the requirements of large-scale computing applications, and how various groups have used FPGA hardware to accelerate application performance.

2.1 Class 1 Machines

The majority of architectures that are designed to meet the demands of high-performance computing applications fall in the Class 1 category, which consists of a network of CPUs. The IBM Blue Gene/L [13] is the most recent example of an advanced Class 1 architecture, but this classification also includes simple clusters of workstations. Despite the wide variety of Class 1 implementations available, there are numerous levels of parallelism that a CPU simply cannot exploit, even with multiple execution units, SIMD instruction extensions, and out-of-order processing. For this reason, vendors of Class 1 machines have recently incorporated reconfigurable FPGA hardware into their respective architectures to create Class 2 machines.

2.2 Class 2 Machines

In Class 2 machines, FPGAs are used as coprocessors to exploit fine-grained parallelism in algorithms via pipelining, or data parallelism by replicating multiple instances of functional units. This category encompasses all machines that combine some form of reconfigurable hardware with at least one CPU. Figure 2 differentiates the methods used by three high-performance computing vendors for integrating reconfigurable hardware into their architectures. Each company has devised different methods for handling the data transfer bottleneck between the CPU and FPGA environments, as well as mechanisms for implementing computing kernels in FPGA hardware.

The Cray XD1 [11] platform connects FPGAs directly to CPUs using its proprietary RapidArray interconnect mechanism. The FPGA can be configured to map external SRAM components into the processor user memory space, allowing a software application to transfer data directly to/from the FPGA domain. Under ideal operating conditions, the FPGA can process data in one region of the SRAM, while the CPU concurrently fills or drains buffers in another region. A library of software routines with support for predesigned FPGA hardware is provided to end-users of the system for application acceleration.

SGI uses its proprietary NUMALink^{TM}interconnect fabric to connect FPGA nodes to the rest of the system network [28]. Unlike the Cray configuration, which uses external SRAM as a cache, the SGI architecture allows the FPGA direct access to the memory coherency domain used by the

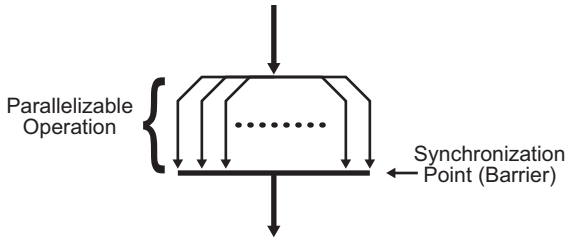

Figure 3: Fork-Join Control Flow Graph

processors. The external SRAM is used by the FPGA for intermediate storage only. Software access to the FPGA is also achieved through a set of FPGA-aware library routines.

Contrary to the approaches used by Cray and SGI, the SRC Computers system does not couple the FPGAs for user logic directly to the system bus [31]. An intermediate FPGA is first used as a bridge between the system bus and a dual-port RAM array. The second port of the RAM connects to multiple FPGA chips containing user-definable logic. The intermediate RAM array is used to implement buffers for exchanging data between the system bus and FPGA domains. SRC uses a custom compiler for automatically extracting algorithms suitable for hardware implementation from C code.

The final machine we consider is relevant to our intended application of Molecular Dynamics, even though it is not considered a supercomputing platform. The PROGRAPE (PROgrammable GRavity Pipe) [16] hardware is a FPGA-based implementation of the GRAPE engine [32], which was developed for n-body simulations of stellar bodies and later extended to molecular dynamics. PROGRAPE and its predecessors accept as input a list of particles and their associated position coordinates, and returns the vector summation of the net force on each particle due to the other $n-1$ particles in the system. Since PROGRAPE is based on reconfigurable hardware, the equations governing interparticle forces can be altered to suit a variety of n-body applications. The PROGRAPE FPGA hardware is connected to a host computer using a PCI bus. Software executing on the host CPU performs the $O(n)$-complexity computing tasks and uses the FPGA hardware to accelerate the $O(n^2)$ calculations.

Despite the design effort expended by Cray, SGI, and SRC to integrate FPGAs into their supercomputer architectures, there are a number of issues that limit the performance gain of the overall system. FPGAs and the hardware structures that are constructed using them are inherently parallel devices. Yet in Class 2 machines they are controlled by CPUs that execute a serial sequence of instructions. The resulting control flow graph typically resembles the fork-join model illustrated in Figure 3. Processors execute serial streams of instructions, and when an opportunity arises for exploiting parallelism or accelerating a single task, the computation is transferred to hardware. The results are then transferred back to the CPU once computation has finished. Some of the overhead for transferring data and waiting for the hardware acceleration to complete can be amortized by performing unrelated tasks in the CPU, similar to techniques used for masking functional unit stalls in modern processors. However, identifying these optimizations and implementing them effectively requires significant effort on the part of both the hardware and software developers. The GRAPE systems and all of their programmable derivatives are classic examples of this drawback: at each timestep, coordinate data is

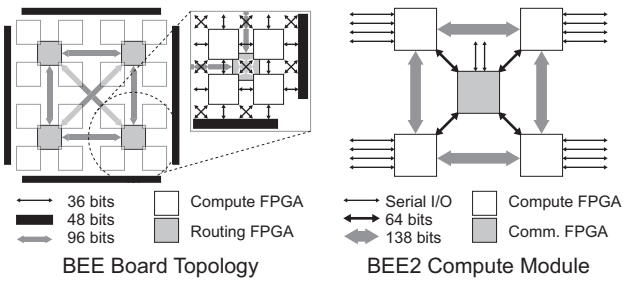

Figure 4: BEE and BEE2 Architectures [8], [9]

transferred across the PCI bus, the hardware engines are polled until the calculations have completed, and the force results are transferred back to the CPU. The CPU is often idle when the FPGA is busy processing and vice-versa.

2.3 Class 3 Machines

A number of groups have begun to investigate the merits of building high-performance computing platforms using FPGA-based technology exclusively. For example, the newly-commissioned FPGA High Performance Computing Alliance (FHPCA) is working on developing a new type of supercomputer using reconfigurable FPGA technology, although details on the implementation are not yet available [12]. The OpenFPGA effort, launched in early 2005, is also a joint effort by industry and academia to explore various aspects of reconfigurable supercomputing [24]. Researchers at the Berkeley Wireless Research Center have been using Class 3 machines since 2003 to assist with the rapid prototyping and development of wireless communication algorithms.

The Berkeley Emulation Engine (BEE) is a machine designed for prototyping wireless communication algorithms. It is comprised of a network of 20 Xilinx Virtex 2000E FPGAs, interconnected with multiple parallel buses, and 16 SRAM chips. It resides on a single 53cm x 58cm, 26-layer printed circuit board (PCB) [8].

BEE2, shown beside its successor BEE in Figure 4, is more modular in design. It is is constructed using smaller PCBs consisting of five Xilinx Virtex 2 Pro 70 FPGAs. BEE2 also uses parallel buses to implement connections between FPGAs. One FPGA is designated as the control processor, and it connects via four buses to the remaining four computation FPGAs. The computation FPGAs are also interconnected using a mesh topology. All of the buses within a module transmit data on both edges of a 300MHz clock signal, providing over 2×40.0Gbps of bandwidth between two adjacent computing FPGAs, and 2×20.0Gbps of bandwidth between each of the computing FPGAs and the communication FPGA. Every FPGA also has four independent DDR2-400 memory interfaces capable of addressing a total of 4GBytes. Global interconnection of the compute modules is achieved through commercial high-speed serial interconnection protocols, such as 10-Gbit Ethernet or Infiniband [9]. The computing modules conform to the 8U Blade form factor (32.225cm x 28.0cm) and are designed to be used in a rack-mounted chassis.

The BEE/BEE2 systems are designed to have large memory bandwidth and high-capacity data links, making the architecture ideal for applications that can be readily described using a synchronous dataflow model. An entire tool

flow has been developed for BEE/BEE2 based on this set of applications, allowing designers to enter high-level algorithm descriptions using the MathWorks Simulink [29] language and generate the appropriate FPGA configuration files. However, this flow does not lend itself to applications that cannot be described using synchronous dataflow models, such as the high compute-to-communication applications we are interested in. This is a limitation of the tool flow and not the BEE/BEE2 architecture.

3. ARCHITECTURAL OVERVIEW

A processing node in a Class 1 machine is typically comprised of memory, I/O interfaces, and a CPU consisting of several arithmetic units. Class 2 machines augment this archetype with external FPGA hardware. However, implementing a node in either type of machine requires multiple physical components. Accordingly, there is a limit to the integration density achievable in Class 1 and 2 architectures. Recent advances in FPGA technology have enabled the integration of circuits such as arithmetic cores, memory blocks, high-speed I/O interfaces and microprocessors into a single FPGA package. This fusion of reconfigurable hardware with processing and communication elements results in an ideal building block for a computing platform.

3.1 Computing Model

We define a unit of computation, regardless of its complexity or implementation method, as a *task*. Tasks communicate data with each other but are otherwise self-contained.

In Class 1 architectures, tasks are implemented as software *processes* executed by a CPU. The granularity of a process is fairly coarse, since there is a trade-off between parallelizing a computation by adding extra processes to it, and the overhead incurred by creating each additional process. Regardless of the memory requirements or computational complexity of a process, a CPU is capable of executing only one process at a time. As a result, processes that do not require significant amounts of computational effort will underutilize a CPU, and computationally-intensive processes will monopolize CPU time. Extra CPUs must be added to a system to increase the number of processes that can be executed simultaneously.

Tasks can be implemented using one of two methods in a Class 3 machine. The first method is to use a software process executing on a FPGA-based *embedded microprocessor*. CPU cores implemented using FPGA fabric (soft processors) and fixed CPU cores incorporated into the FPGA fabric are both considered embedded microprocessors. Although using embedded processors to implement tasks is relatively straightforward, this approach suffers from the same drawbacks as Class 1 architectures: the maximum level of parallelism that can be achieved is limited.

The second method of implementing a task on a Class 3 machine is to create a hardware *computing engine*. This approach allows designers to take advantage of much finer granularities of parallelism, and to create a solution that is optimized to meet the performance requirements of a task. The major advantage of this approach is that only the hardware relevant to performing a task is used to create a computing engine, allowing many small engines to fit within one FPGA or one large engine to span across multiple FPGAs. Increasing the number of parallel operations performed by a computing engine can be achieved by modifying it to capitalize on data parallelism within the task or by adding more engines to the system to exploit task-level parallelism. Figure 5 illustrates different methods for implementing a task on CPU or FPGA platforms.

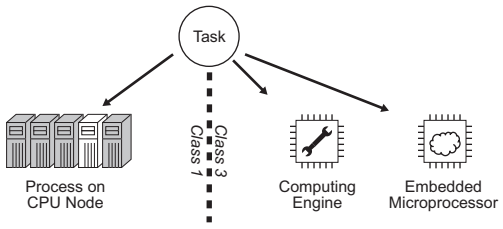

Figure 5: Implementations of Computing Tasks

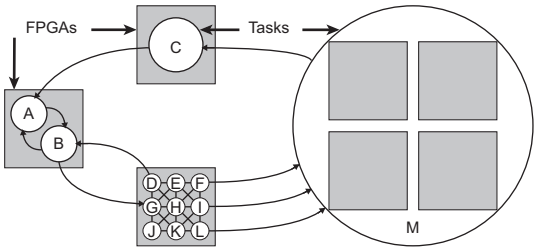

Figure 6: Multiple Computing Engine Example

The computing model for the TMD is described as a collection of independent tasks executing concurrently. Tasks exchange data by passing messages to each other via the TMD communication network. System designers can specify whether tasks are implemented as processes executing on CPUs, or as hardware computing engines according to the requirements of the application. This is a manual process in the current flow. Figure 6 illustrates a possible architecture for a hypothetical application consisting of multiple tasks implemented across several FPGAs. Tasks A and B reside on one FPGA, as does task C. Tasks D through L are all contained in a single FPGA, indicating that they are less complex than the other tasks in the system. Finally, four FPGAs are required to implement task M.

3.2 TMD System Architecture

The TMD is a scalable computing platform constructed solely of FPGA processing nodes. Applications for the TMD platform are designed as a collection of software processes and hardware computing engines interconnected by a configurable communication network. Application designers have the flexibility to specify the structure of each component, as well as the network topology for interconnecting them. The goal of the TMD architecture is to avoid imposing constraints in the design of an application by ensuring that the logical links between computing engines and CPUs are as flexible and abundant as possible. Furthermore, the network interface is abstracted in a manner that allows a computing engine or CPU to use a standard protocol and physical interface to communicate with another engine or CPU, regardless of which FPGA node either component resides in. Essentially, the TMD architecture allows application designers to treat the system as a single piece of scalable FPGA fabric.

A distributed memory model was chosen for the TMD as it allows the platform to scale well with additional FPGA nodes. Each computing task contains a separate instance

of local memory, and data is exchanged between tasks by passing messages. Although the TMD architecture is tailored to MD simulations, it can accommodate the subset of high-performance computing applications that require a distributed memory model and exhibit a high computation-to-communication ratio.

The TMD architecture is divided into three hierarchical tiers, allowing it to scale up to configurations potentially containing thousands of FPGAs. Figure 7 illustrates each of the three tiers. The lowest tier of the TMD hierarchy exists within a FPGA package. Designers can specify any network topology for interconnecting computing engines and embedded processors contained in a single FPGA, subject to resource constraints.

The second tier of the hierarchy is at the PCB level, and is referred to as a *cluster*. One cluster consists of eight FPGAs for implementing computing tasks, and an additional FPGA for communicating with other clusters. A fully-connected topology is used to network the FPGAs on a cluster.

The third tier of the hierarchy is used to interconnect clusters to form a large network. Commercially-available switches designed for high-performance computing applications could be used in this tier, since they are capable of scaling up to systems containing hundreds of clusters and therefore thousands of FPGA nodes.

3.3 System Implementation

The Xilinx Virtex-II Pro XC2VP100 FPGA is used to implement computing nodes in the TMD. The XC2VP100 features twenty high-speed serial I/O links, 444 multiplier cores, 7.8Mbits in distributed BlockRAM structures, and two embedded PowerPC processors [35].

3.3.1 Intra-FPGA Communication

Intra-FPGA communication is achieved through the use of point-to-point unidirectional FIFOs. The FIFOs are implemented using the Xilinx Fast Simplex Link (FSL) core, as it is fully-parameterizable and optimized for the Xilinx FPGA architecture [35]. Recent work by Saldaña *et. al.* [27] has shown that point-to-point connections can be used to attain a fully-connected network topology on modern FPGA fabrics for systems containing up to 16 nodes.

Both computing engines and embedded microprocessors use the FSL physical interface for sending and receiving data across communication channels. FSL modules provide 'full' and 'empty' status flags that can be used by transmitting and receiving computing engines as flow control and synchronization mechanisms. Using asynchronous FSLs allows a computing engine to operate different clock frequency than other components in the system.

3.3.2 Inter-FPGA Communication

Many FPGA-based systems use parallel buses to communicate data between FPGAs and other components in the system. To achieve high data rates, wide buses operating at high clock frequencies must be carefully routed between components on a PCB. Since this method is rapidly approaching physical limits [5], FPGA manufacturers have incorporated hardware to support gigabit-rate serial I/O interfaces into their products [3, 35]. The TMD inter-chip communication network uses multi-gigabit transceiver (MGT) hardware to implement the physical communication links between FPGA components.

Twenty MGTs are available on the XC2VP100 FPGA, each capable of providing 2×3.125Gbps of full-duplex communication bandwidth over only two pairs of traces. Future revisions of both Xilinx and Altera FPGAs will increase this data rate to over 10Gbps per channel [3, 35]. Our current hardware configuration limits the maximum raw data rate to 2.5Gbps per direction, yielding an effective bandwidth of 2×2.0Gbps after 8B/10B encoding.

A fully-connected network topology is used to interconnect all nine FPGAs on a cluster PCB. Using MGT links to implement this topology requires only 144 traces, and yields a maximum theoretical bisection bandwidth of 2×32.0Gbps (assuming 2×2.0Gbps per link) between the eight computing FPGAs. By contrast, each BEE2 module requires 808 traces to interconnect five FPGAs and can obtain a maximum bisection bandwidth of 2×80.0Gbps between four computing FPGAs. Therefore, PCB complexity can be reduced considerably by using MGTs as a communication medium, and with 10.0Gbps serial transceivers on the horizon, bandwidth will increase accordingly.

The Aurora core available from Xilinx [35] is designed to interface directly to MGT hardware and provides link-layer communication features. An additional off-chip communication controller (OCCC), described in [10], was also developed to supplement the Aurora and MGT hardware cores. The OCCC provides reliable transport-layer communication between tasks residing on different FPGAs, and currently uses a lightweight protocol designed to minimize communication latency. A minimum trip time of 1.23μs is observed for a 32-byte packet, while larger packets achieve a peak full-duplex throughput of 2×1.928Gbps corresponding to 96.4% link efficiency.

Computing engines and embedded microprocessors connect to the OCCC using the FSL interface described in Section 3.3.1. This enables hardware and software for the TMD to be developed without consideration for whether data will travel through on-chip or off-chip channels.

3.3.3 Inter-Cluster Communication

The MGT links described in Section 3.3.2 can be used to emulate standardized high-speed interconnection protocols such as Infiniband [20] or 10-Gbit Ethernet [19]. The 2×10.0Gbps 4X SDR subset of the Infiniband specification can be implemented by aggregating four MGT links, enabling the use of commercially-available Infiniband switches for accomplishing the global interconnection network between clusters. This approach reduces the design time of the overall system, and provides a multitude of features necessary for large-scale systems, such as fault-tolerance, network provisioning, and scalability.

4. PROGRAMMING MODEL

A programming model provides a user with an efficient method for implementing applications while abstracting underlying hardware complexities. It is arguably as important a component to the performance of a high-performance computing machine as the design of the architecture itself. Programming models are defined by the architecture of a machine, as well as the nature of applications intended to be executed on the machine. This section presents a programming model for the TMD that is well-matched to its scalability, parallel processing ability and distributed memory architecture. However, this programming model is not

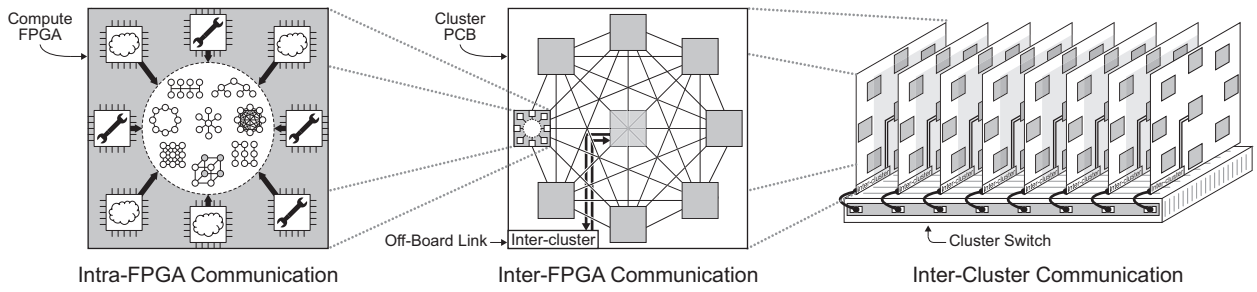

Figure 7: TMD Architecture Hierarchy

limited to the TMD. It can also be applied to other Class 3 architectures with support for distributed memory, such as the BEE/BEE2 [8, 9] platforms described in Section 2.3.

4.1 Message Passing Model

The architecture of the TMD does not utilize shared memory or shared bus structures; rather, each computing engine contains its own local memory. A programming model suitable for distributed memory is necessary for such a system. The message-passing programming model fits this requirement, and has also proven to be a successful paradigm for scientific computing applications running on supercomputers and clusters of workstations. Using a standardized message-passing interface such as MPI [23] as the basis for the TMD programming model provides a familiar environment for application developers. Moreover, using MPI enables portability of parallel applications to different platforms by standardizing the syntax and semantics of the programming interface.

The programming approach we present begins with a network of embedded microprocessors, implemented on FPGA fabric, exchanging messages using MPI functions. Applications designed for the TMD are initially described as a set of software processes executing on the microprocessor network. Each process emulates the behaviour of a computing engine and uses MPI function calls to exchange data with other processes. These function calls use the TMD communication infrastructure described in Section 3.3 to transmit and receive data packets. An entire system can be prototyped on the TMD in this manner. We use the abstraction of embedded microprocessors executing MPI function calls as an intermediate step in the process of mapping an application to a set of computing engines implemented on the TMD.

4.2 Design Flow

Prior to describing the TMD design flow, it is instructive to differentiate the challenges between mapping a single computational kernel to an FPGA-based hardware module, and mapping an entire parallel application to an architecture containing multiple FPGAs.

A number of design tools exist for directly translating C code into synthesizable hardware, such as HardwareC [21], Handel-C [17], or C2Verilog [30]. These tools are best suited for identifying computationally-intensive application kernels and implementing them in hardware modules, usually confined to a single FPGA. However, software applications contain other programming tasks that can pose challenges to automated software-to-HDL design flows, such as I/O functions, complex control structures, and pointer-based memory accesses. Consequently, not every application can be translated directly into hardware.

Additional complexities arise when developing a method for automatically implementing *parallel* applications on the TMD. Communication networks for interconnecting FPGAs in the TMD are defined, but networks for interconnecting multiple computing engines within individual FPGAs must be implicitly inferred by analyzing the communication pattern of the software implementation. Intrinsic data and functional parallelism existing within the application should also be automatically detected and taken advantage of. Higher-level parallel application features such as dynamic load balancing and task allocation are also important constructs that may not be realizable in hardware. In Class 2 machines, many of these challenges are mitigated due to the presence of the host processors in the system.

The current design flow developed for the TMD does not attempt to address these issues automatically. Rather, a manual flow is used to transform parallel software applications into networks of hardware computing engines. This approach allows us to study and evaluate design decisions for the TMD architecture, identify potential problems in the flow, and debug the initial architecture. An automated design process would follow as a result of these activities. Our current flow resembles that presented by Youssef *et. al* [36], but we extend the model to many embedded microprocessors distributed across multiple FPGAs.

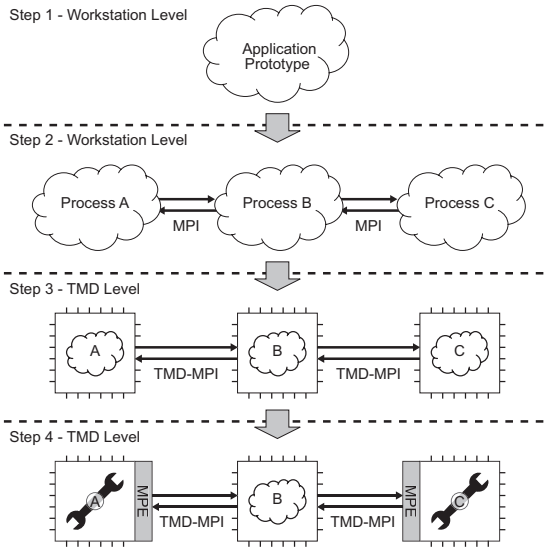

Figure 8: TMD Application Design Flow

Figure 8 illustrates the TMD design flow. Step 1 begins by developing a prototype of the application in a high-level programming language such as C/C++. The resulting code is sequential in nature and is only intended to provide a generic solution to the computing problem. At this stage, the application can be profiled to identify computationally-intensive routines.

Step 2 refines the prototype application by partitioning it into simple, well-defined processes that can be replicated to exploit the implicit parallelism of the application. The granularity of each process is much finer than that of software processes normally used in parallel applications, since each process is meant to emulate the behaviour of a computing engine. Inter-process communication is achieved using a full implementation of the MPI message passing library, allowing the application to be developed and validated on a workstation. This approach has the advantage of allowing the programmer access to standard tools for developing, profiling, and debugging parallel applications. To proceed with the next step, the programmer refines the parallel program so that it is compatible with the functionality provided by the TMD-MPI library, as described in Table 1 of Section 5.2.

Step 3 takes the collection of software processes developed in Step 2 and implements them on the TMD using embedded microprocessors. Each microprocessor contains a library of MPI-compliant message-passing routines designed to transmit messages using the TMD communication infrastructure. The portability of MPI allows the software code to be recompiled and executed on the microprocessors. At this stage, execution of the entire application is possible, allowing the interaction between emulated computing engines and CPUs to be tested and the implementation correctness to be validated on the architecture.

The final step of the programming flow replaces algorithms executing on embedded microprocessors with hardware computing engines. This step is only required for performance-critical computing engines and can be omitted for less intensive computing tasks. Additionally, control-intensive tasks that are difficult to implement in hardware can remain as software executing on microprocessors. The tight integration between embedded microprocessors and hardware engines implemented on the same FPGA fabric makes this a viable option. Translating the computationally-intensive processes into hardware engines is done manually in the current flow. Since the system has been already partitioned into individual computing tasks and all communication primitives have been explicitly stated at this stage, C-to-HDL tools may also be used to perform this translation. Once a computing engine has been designed, a hardware message-passing engine (MPE) is used to perform message-passing operations in hardware.

5. MPI IMPLEMENTATION

The MPI standard does not specify a particular implementation architecture or style. Consequently, there are multiple implementations of the standard, each with differing performance characteristics, such as OpenMPI [14], MPICH [15], and LAM [6]. This section presents some background on MPI and then describes our own implementation of MPI called TMD-MPI.

5.1 Background

Current MPI implementations are targeted to computers with copious memory, storage, and processing resources, but these resources are scarce in the embedded microprocessor domain. We have implemented a subset of the MPI standard that provides sufficient functionality for many applications, such as the MD simulator presented in Section 6.

In projects such as eMPI/eMPICH [22], the authors port MPICH to the embedded systems domain. They begin by compiling a basic MPICH implementation consisting of six fundamental primitives, and later add functionality to it. A set of embedded libraries are obtained with varying degrees of functionality. However, their approach assumes the existence of a compact lower layer of MPICH as well as an operating system. In our approach, we also present a basic MPI implementation, but it encompasses everything between the programming interface to the hardware access layer and does not require an operating system.

Aggarwal *et. al* [1] present a Class 2 computing machine that uses Handel-C and MPI to implement parallel architectures within and across multiple FPGAs, but the use of MPI is limited to inter-host communication. In our work, MPI is used to communicate between computing engines and embedded microprocessors without the presence of an external host.

Youssef *et. al.* [36] describe the development of a hardware OpenDivX video encoder. The application is initially described using four parallel software processes communicating using MPI send and receive primitives. The system is then implemented using SystemC, and appropriate versions of the MPI primitives are also implemented for simulation purposes. Such simulation aids could prove beneficial in the translation process performed in Step 3 of the TMD design flow. However, only the send and receive primitives are implemented, which do not provide sufficient functionality for our scope of applications.

The BEE2 project described in Section 2.3 also suggests using MPI as a programming model to simplify the task of porting supercomputing applications to the BEE2 architecture [9]. However, to the best of our knowledge, no results have been presented yet based on this endeavour.

5.2 Implementation of TMD-MPI

This section describes the implementation of a simplified, light-weight MPI library called 'TMD-MPI' implemented specifically for embedded microprocessors on the TMD architecture. Although TMD-MPI is currently written for the Xilinx MicroBlaze microprocessor [35], it can easily be ported to different platforms by modifying the lower hardware interface layers.

There are several hardware and software issues to resolve before an implementation of the TMD-MPI library can be developed. We begin by describing the hardware issues pertaining to the communication infrastructure of the TMD. Two classes of networks exist within the TMD; the first is the external network described in Section 3.3.2 that interconnects FPGA components, and the second is the internal communication network described in Section 3.3.1 that interconnects the computing engines and embedded microprocessors within a single FPGA. A lightweight point-to-point protocol is used to transmit data over the internal communication network. The external communication uses a slightly different packet format required by the OCCC, and bridge modules are used to translate data packets between the internal and external network formats.

In the current version of the TMD-MPI library, message-passing functions such as protocol processing, management of incoming and pending message queues, and packetizing and depacketizing of long messages are performed by the embedded microprocessor executing the application process. This method allows the message-passing functionalities to be developed and tested easily, but incurs a performance penalty in the application process. Once the details of the TMD-MPI implementation have been finalized, most of the message-passing functionality will be provided by more efficient hardware cores. This translates into a reduction in processing overhead for embedded microprocessors as well as simplification of hardware computing engines. An example of how certain MPI functionality can be implemented in hardware is found in Underwood [34], where the authors manage MPI message queues using hardware buffers. This implementation reduced the latency for queues of moderate length while adding only minimal overhead to the management of shorter queues.

The TMD-MPI implementation follows a layered approach similar to the method used by MPICH. The primary advantage of this technique is that TMD-MPI can be ported to different platforms by modifying only the lowest layers of the implementation. Figure 9 illustrates the four layers of the TMD-MPI implementation.

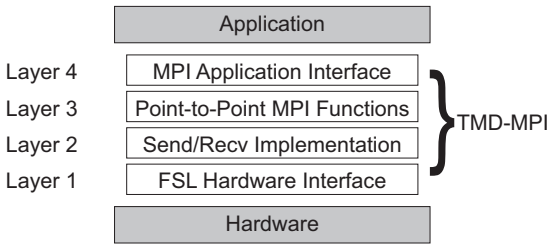

Figure 9: TMD-MPI Implementation Layers

Layer 4 represents the MPI functional interfaces available to the application. Layer 3 implements collective operations such as synchronization barriers, data gathering, and message broadcasting (`MPI_Barrier`, `MPI_Gather`, and `MPI_Bcast`, respectively) using simpler point-to-point MPI primitives. Layer 2 consists of the point-to-point MPI primitives, namely `MPI_Send` and `MPI_Recv`. Implementation details such as protocol processing, data packetizing and depacketizing, and message queue management are handled here. Finally, Layer 1 is comprised of four macros that provide access to physical communication channels. Porting TMD-MPI to another platform requires a replacement of Layer 1 and possibly some minor changes to Layer 2.

TMD-MPI currently implements only a subset of functionality specified by the MPI standard. Although this set of operations is sufficient for our initial MD application, other features can be added as the need arises. Table 1 lists a description of the functions implemented to date.

There are several outstanding issues in TMD-MPI that are currently being investigated. In a MPI-based application executing on a Class 1 machine, the *mpirun* command is invoked by a user to launch the application. This command uses the underlying operating system to spawn multiple processes on different hosts, and is also responsible for assigning unique ranks to each process. Since there is no underlying operating system in our current implementation, we statically assigned each process to an embedded microprocessor and determine MPI process ranks at compile-time. A boot-loading mechanism is being developed to provide this functionality. Other minor issues include: lack of support for data types larger than 32 bits, support for only synchronous (blocking) implementations of `MPI_Send` and `MPI_Recv` primitives, and support for only multiplication and addition reduction operations in `MPI_Reduce`.

Table 1: Functionality of TMD-MPI

Utility Functions	
`MPI_Init`	Initializes TMD-MPI environment
`MPI_Finalize`	Terminates TMD-MPI environment
`MPI_Comm_rank`	Get rank of calling process in a group
`MPI_Comm_size`	Get number of processes in a group
`MPI_Wtime`	Returns number of seconds elapsed since application initialization
Point-to-Point Functions	
`MPI_Send`	Sends a message to a destination process
`MPI_Recv`	Receives message from a source process
Collective Functions	
`MPI_Barrier`	Blocks execution of calling process until all other processes in the group reach this routine
`MPI_Bcast`	Broadcasts message from root process to all other processes in the group
`MPI_Reduce`	Reduces values from all processes in the group to a single value in root process
`MPI_Gather`	Gathers values from a group of processes

6. SAMPLE APPLICATION

The first application for the TMD architecture was created to demonstrate the effectiveness of the programming model and design flow described in Section 4.2. Molecular simulations of biological systems have long been one of the principal application domains of large-scale computing [2].

6.1 Molecular Dynamics

Atomic-level simulations of biomolecular systems have become an integral tool of biophysical and biomedical research. One of the most widely used methods of computer simulation is molecular dynamics where one applies classical mechanics to predict the time evolution of a molecular system. In MD simulations, empirical molecular mechanics equations are used to determine the potential energy of a collection of atoms as a function of the physical properties and positions of all atoms in the simulation. The net force acting on each atom is determined by calculating the negative gradient of the potential energy with respect to its position. With the knowledge of both the position and the net force acting on every atom in the system, Newton's equations of motion are solved numerically to predict the movement of every atom. This step is repeated over small time increments ($\Delta t \cong 10^{-15} s$) to yield a time trajectory of the molecular system. For meaningful results, these simulations need to reach relatively large length and time scales, underscoring the need for scalable computing solutions.

There are many software-based MD simulators available including CHARMM [18], AMBER [7], and NAMD [25]. Current theoretical and technological limitations limit MD simulations to a maximum on the order of 10^5 atoms and time scales on the order of approximately 10^{-8} seconds.

The MD application developed for the TMD is patterned

after NAMD, an open-source molecular dynamics simulator for supercomputers. Specifically, the TMD employs the same spatial decomposition strategy for effective parallelization which has allowed NAMD to scale to systems containing thousands of nodes [26]. This version of the application will be used to validate the TMD architecture, programming model, and development flow. Future versions will focus on improving computational performance by leveraging the scalability of the TMD platform.

6.2 Implementation

The first version of the TMD MD application performs simulations of noble gases, which provides the advantage of simplicity since only one type of atomic interaction needs to be considered. The total potential energy of the system results from van der Waals forces which are modeled by the Lennard-Jones 6-12 equation [2].

The application was developed using the design flow outlined in Section 4.2. An initial proof-of-concept application was created to determine the algorithm structure. Next, the application was refined and partitioned into four well-defined processes: (1) force calculations between all atom pairs; (2) summation of component forces to determine the net force acting on each atom; (3) updating atomic coordinates; and (4) publishing the atomic positions. Each task was implemented in a separate process written in C++, and inter-process communication was achieved by using MPICH over a standard switched ethernet computing cluster.

The next step in the design flow was to recompile each of the four simulator processes to target the embedded microprocessors implemented on the TMD architecture. The portability of MPI eliminated the need to change the communication interface between the software processes. The simulator was partitioned onto 2 FPGA nodes as illustrated in Figure 10. Each node is implemented using the Amirix AP1100 development board [4].

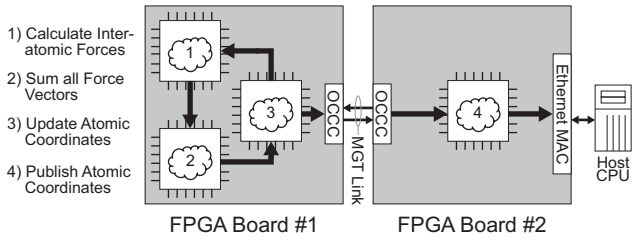

Figure 10: MD Application Implementation

The FPGA on the first board contains three microprocessors responsible for the force calculation, force summation, and coordinate update processes, respectively. All of the processes communicate with each other using TMD-MPI. The second FPGA consists of a single microprocessor executing an embedded version of Linux [33]. It also uses TMD-MPI to communicate with the first FPGA over the MGT link, as well as a TCP/IP-based socket connection to relay atomic coordinates to an external program running on a host CPU.

This initial MD application demonstrates the effectiveness of the programming model by implementing a software application on the TMD architecture. Future versions of the application will build on this foundation by adding further components necessary to simulate more complicated biomolecular systems. The final step in the design flow will be to replace the computationally-intensive processes with dedicated hardware implementations.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have proposed an architecture for a scalable high-performance computing platform, the TMD, built entirely using a flexible network of commodity FPGA hardware. The TMD is designed for applications that exhibit high computation-to-communication ratios as well as an abundance of parallelism. To this end, we have developed an abstracted, low-latency communication interface that enables multiple computing tasks to easily interact with each other, irrespective of their physical locations in the network. The network is realized using high-speed serial I/O links, which facilitate high integration density at low PCB complexity as well as a dense network topology.

We have also developed a programming model for the TMD commensurate with the scalability and parallel nature of the architecture. Using the MPI message-passing standard as the framework for creating applications, we provide parallel application developers with a familiar development paradigm. Additionally, the portability of MPI enables application algorithms to be composed and refined on CPU-based clusters. A flow is also proposed for taking a software application written for CPUs, partitioning it into a collection of parallel computing tasks, and implementing it on the TMD as hardware computing engines and embedded microprocessors. Finally, we have demonstrated the use of the architecture, programming model, and design flow by implementing a simple molecular dynamics simulator.

This work outlines the first step towards the larger goal of building a FPGA-based supercomputer. Our next immediate step is to evaluate the network topology, programming model, and design flow of the proposed architecture by implementing a subset of it using a cluster of development boards. This step also includes analyzing the performance of the internal and external communication networks, refining the TMD-MPI library and developing hardware support for passing messages, and expanding the capability of our flagship MD application. Once we have demonstrated the viability of the TMD architecture, we will investigate methods for automating our design flow.

Acknowledgements

Portions of this work were supported by CMC Microsystems, Amirix Systems, Inc., the SOCRN, NSERC, CONACYT, CIHR, The Hospital for Sick Children and Xilinx, Inc. The authors would also like to thank the reviewers and Andrew House for their helpful comments.

8. REFERENCES

[1] V. Aggarwal, I. A. Troxel, and A. D. George. Design and Analysis of Parallel N-Queens on Reconfigurable Hardware with Handel-C and MPI. In *2004 MAPLD International Conference*, Washington, DC, USA, 2004.

[2] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Clarendon Press, New York, NY, USA, 1987.

[3] Altera, the Leader in Programmable Logic. `http://www.altera.com/`. Curr. Jan. 2006.

[4] AP1000 PCI Platform FPGA Development Board. Technical report, Amirix Systems, Inc., Oct. 2005. http://www.amirix.com/downloads/ap1000.pdf.

[5] D. Brady. FPGA I/O Features Help Lower Overall PCB Costs. *FPGA and Structured ASIC Journal*, Aug. 2004. http://www.fpgajournal.com/articles/20040810_mentor.htm.

[6] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] D. Case, I. T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Jr., A. Onufriev, C. Simmerling, B. Wang, and R. Woods. The Amber biomolecular simulation programs. In *Proceedings of JCCM '05*, volume 26, pages 1668–1688, 2005.

[8] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. W. Brodersen, and B. Nikolic. Rapid Design and Analysis of Communication Systems Using the BEE Hardware Emulation Environment. In *Proceedings of RSP '03*, pages 148–, 2003.

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

[10] C. J. Comis. A High-speed Inter-process Communication Architecture for FPGA-based Hardware Acceleration of Molecular Dynamics. Master's thesis, University of Toronto, 2005.

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

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

[13] R. S. Germain, B. Fitch, A. Rayshubskiy, M. Eleftheriou, M. C. Pitman, F. Suits, M. Giampapa, and T. C. Ward. Blue matter on blue gene/l: massively parallel computation for biomolecular simulation. In *Proceedings of CODES+ISSS '05*, pages 207–212, New York, NY, USA, 2005. ACM Press.

[14] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings of PPAM '05*, Poznan, Poland, Sept. 2005.

[15] 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, Sept. 1996.

[16] 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.

[17] Handel-C Documentation. http://www.celoxica.com. Curr. Jan. 2006.

[18] Y.-S. Hwang, R. Das, J. H. Saltz, M. Hodoscek, and B. R. Brooks. Parallelizing molecular dynamics programs for distributed-memory machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.

[19] IEEE802.3 10GBASE-CX4 Study Group. http://www.ieee802.org/3/10GBCX4/, Mar. 2002. Curr. Jan. 2006.

[20] The InfiniBand Architecture Specification R1.2. Technical report, InfiniBand Trade Association, Oct. 2004. http://www.infinibandta.org.

[21] D. Ku and G. DeMicheli. HardwareC - A language for hardware design. Technical Report CSL-TR-90-419, Stanford University, 1988.

[22] T. P. McMahon and A. Skjellum. eMPI/eMPICH: Embedding MPI. In *Proceedings of MPIDC '96*, page 180, Washington, DC, USA, 1996. IEEE Computer Society.

[23] MPI - Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/. Curr. Jan. 2006.

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

[25] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. In *Proceedings of JCCM '05*, volume 26, pages 1781–1802, 2005.

[26] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: biomolecular simulation on thousands of processors. In *Proceedings of Supercomputing '02*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[27] M. Saldaña, L. Shannon, and P. Chow. The Routability of Multiprocessor Network Topologies in FPGAs. In *Proceedings of SLIP '06*, 2006. (To Appear in March).

[28] 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.

[29] The MathWorks - MATLAB and Simulink for Technical Computing. http://www.mathworks.com/. Curr. Jan. 2006.

[30] D. Soderman and Y. Panchul. Implementing c algorithms in reconfigurable hardware using c2verilog. In *Proceedings of FCCM '98*, page 339, Washington, DC, USA, 1998. IEEE Computer Society.

[31] General Purpose Reconfigurable Computing Systems. 2005, SRC Computers, Inc. http://www.srccomp.com/.

[32] M. Taiji, T. Narumi, Y. Ohno, and A. Konagaya. MDGRAPE-3: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations. In *Proceedings of PARCO '05*, pages 669–676, 2003.

[33] uClinux Embedded Linux/Microcontroller Project. http://www.uclinux.org/. Curr. Jan. 2006.

[34] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Proceedings of IPDPS '05*, page 96.2, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Xilinx - The Programmable Logic Company. http://www.xilinx.com/. Curr. Jan. 2006.

[36] M.-W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. A. Jerraya. Debugging HW/SW interface for MPSoC: video encoder system design case study. In *Proceedings of DAC '04*, pages 908–913, New York, NY, USA, 2004. ACM Press.