Applications Development with FPGAs
Simulate but Verify

G. Wenes, J. Maltby, D. Strenski
Cray, Inc.,
411 First Avenue S., Suite 600, Seattle, WA 98104-2860, USA
[geertw, jmaltby, stren]@cray.com,
http://www.cray.com

Abstract

Reconfigurable Computing (RC) exploits hardware, such as Field Programmable Gate Arrays (FPGAs), that can be reconfigured to implement specific functionality more suitable for specially tailored hardware than on a general purpose processor. However, applications development and performance optimization possible with these systems typically are dependent on the skill and experience of hardware designers. This has prevented more widespread use of RC in the High Performance Computing (HPC) community. Hence, a current challenge in this area is the establishment of a methodology that is targeted towards the HPC software engineer and application developer. We will present, with examples, such methodology.

Keywords: *Field Programmable Gate Arrays, Reconfigurable Computing, Cray XD1, Synthesis, Simulation, Verification, Programming Languages.*

1 Introduction

1.1 Reconfigurable Computing

The main characteristic of Reconfigurable Computing (RC) is the presence of logic that can be reconfigured or reprogrammed to implement specific functionality more suitably than on a general purpose processor. The term *reconfigware* (RW) has been coined to suggest how RC systems can join microprocessors and other programmable hardware in order to take advantage of the combined strengths of hardware and software.

1.2 Programmable Logic

There are three basic kinds of electronic devices: memory, microprocessors, and logic. Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform. Programmable logic devices (PLDs) are standard, off-the-shelf parts that offer customers a wide range of logic capacity, features, speed, and characteristics. Of all types of PLDs, Field Programmable Gate Arrays (FPGAs) offer the highest amount of logic density, the most features, and the highest performance.

1.3 Functionality

Field Programmable Gate Arrays (FPGAs) can be used to implement just about any functionality.

1.4 Suitability

Just a few years ago, the largest FPGA was measured in tens of thousands of system gates and operated at 40 MHz. Today, however, FPGAs offer millions of gates of logic capacity, operate at 500 MHz, and are manufactured in state-of-the-art 90nm low-k copper process. They offer a new level of integrated functions such as on-board scalar processors. These advanced devices also contain substantial amounts of memory, and offer features such as clock management systems.

FPGAs have a growing library of intellectual property (IP) or cores - these are predefined and tested software modules that customer can use to create system functions instantly inside the FPGA. Cores are similar to libraries in microprocessors: they are robust, optimized, modular, and correct implementations of some basic functionality in customers' application space.

These two trends conspire to blurr the distinction between logic and microprocessor devices: FPGAs are used increasingly to provide application performance while current processor chip-sets, for instance, may include communication devices. Industries such as manufacturing, government, research, media, and biosciences increasingly are deploying FPGAs. Their benefits are as hardware application accelerators that provide orders of magnitude ($10-100$) performance improvement over common microprocessors such as Intel's Pentium4. Not every application, however, is suitable for such performance acceleration: targeted applications are highly parallel at a fine grain level (*on-chip* parallelism), perform limited-precision fixed-point arithmetic, and have a high operation count per datum. Examples of suitable applications are: searching, sorting, signal processing, audio/video/image manipulation, encryption, error correction, coding/decoding, packet processing, and random number generation. However, floating point intensive, but highly pipelined, applications such as molecular dynamics [11] qualify too.

The remainder of the paper is organized as follows: Section (2) describes the architecture of the Cray XD1 as a reconfigurable computing platform in some detail,

Section (3) describes the Integrated Development Environment for the Cray XD1, while Section (4) describes, with examples, the applications development framework. Section (5) is more speculative in that it suggests a common development language for both high performance computing programmers and electronic device engineers. Finally, Section (6) presents the conclusions.

2 Cray XD1 Architecture and FPGA Expansion Module

The Cray XD1 high performance computer is based on the Direct Connected Processor (DCP) architecture, developed to remove both the communications bottleneck which limits application performance in 32 and 64-bit clusters and the memory contention which limits the scale of Symmetric Multiprocessor (SMP) solutions. The DCP architecture views the system as a pool of processing, logic, and memory resources interconnected by a high bandwidth, low latency network. The entry-level configuration for the Cray XD1 is a single chassis, see Figure (1).

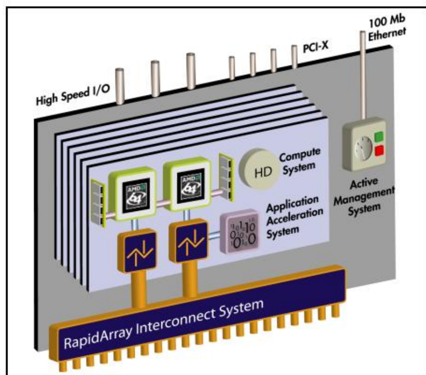

Figure 1: Schematic view of the XD1 Chassis. Each chassis contains 12 compute nodes (AMD Opteron 32/64 bit, x86 compatible processors, now also available in dual core), connected via RapidArray Interconnect (a 1 Tb/s switch fabric), and 6 Application Acceleration co-processors. See text for further description.

Each chassis consists of a main board, 6 compute blades and an optional switching fabric expansion board. Each chassis provides

- 12 AMD Opteron 248 processors, now also available in dual core, organized as 2-way SMPs;
- A 1 Tb/s per second Rapid Array embedded switch fabric;
- 12 RapidArray Communications Processors;
- Up to four independent PCI-X I/O slots;
- 0.5 to 8 GB of DDR 400 SDRAM per core;
- 24 RapidArray inter-chassis links with an aggregate 48 GB per second bandwidth;
- 6 Application Acceleration processors for reconfigurable computing.

Of most interest here is the expansion module, shown in Figure (2): parts of an application that are compute-intensive and highly repetitive can be assigned to a specific FPGA processor residing on the expansion module for application acceleration. These Application Acceleration processors are tightly integrated into the Opteron's memory map and use standard software programming primitives which treats the FPGAs as co-processors to the Opteron.

The bandwidth between the FPGA and *any* SMP is 3.2 Gbytes/s (or 1.6 Gbytes/s in each direction simultaneously) while the fabric latency to the local SMP memory is approximately a few hundred nanoseconds. Once a memory access request leaves the FPGA it travels through a single fabric processor to get to the local SMP. The delay through the fabric processor is $50 - 100$ ns. The round trip latency for read operations would be twice the fabric latency plus whatever time the Opteron memory manager requires to complete DRAM access. The latency to remote SMPs will typically involve the delay from two fabric processors ($100 - 200$ ns) plus the delay through one or more switching elements (250 ns each).

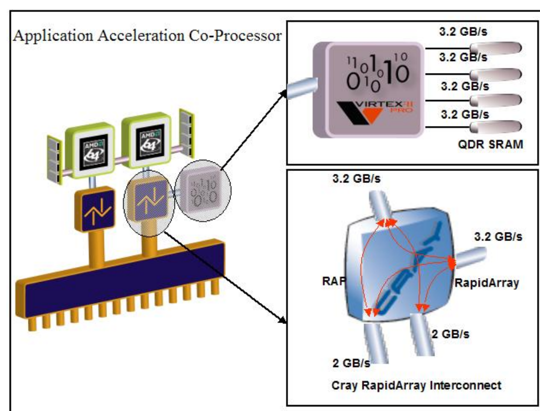

Figure 2: Schematic view of the FPGA expansion module. The expansion module currently supports three different Xilinx Virtex II Pro devices: the XC2VP30, XC2VP40, and XC2VP50. See text for further description.

3 Integrated Software Environment

3.1 Synthesis and Simulation

Programming logic devices is not common practice in HPC. Logic programmers need to have a good understanding of the physical properties (timing, size,...) and the basic workings of the device, which is contrary to standard software development practices that advocate a strict separation of hardware and software.

For low-level development, the minimum toolset required for programming logic devices on the XD1 is the Xilinx Integrated Software Environment (ISE) package. This ISE covers HDL synthesis (VHDL or Verilog) through the generation of the binary programming file. Many third party software packages are available for synthesis (Synplicity, Mentor, Synopsis, etc.) as well as for simulation (ModelSim, Cadence, etc.)[1] The Xilinx ISE package incorporates VHDL or other cores into the design. The Coregen tool – included with ISE – will create many standard cores. In addition, Xilinx and other companies provide a wide variety of more sophisticated cores for a fee. Cray provides, for free, the cores for the QDR SRAM and Rapid Array fabric interface (QDR2 Core, resp., RT Core). These are provided as pre-synthesized and placed netlists that can be incorporated by the tools into a user design. For the development of new VHDL cores, Xilinx, Mentor, Cadence, Synopsys and others provide extensive software tools for generating VHDL or Verilog designs. This is shown schematically in Figure (3).

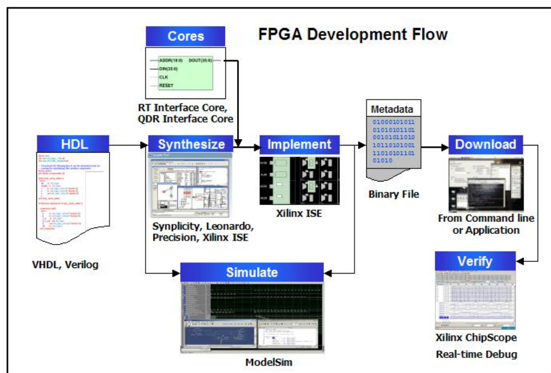

Figure 3: Program development flow for FPGAs from HDL to placed Core.

For programming using any other, or more abstract models, available tools include Mentor, Cadence, Synopsys, Celoxica, and other support for C synthesis. Xilinx offers a Forge tool that generates designs from Java code. Xilinx also supports an interface to MATLAB for DSP designs.

[1] *Simulation* executes the logic machinery, given a set of inputs; *synthesis* results in the optimized (for size, performance, power, ...) structure of the machine.

3.2 Programming Languages

High-level abstraction approaches require at a minimum:

- Multi-threaded languages;
- Channels for communications and synchronization;
- User-specified and reconfigurable target architectures;
- Good-to-excellent Quality of Results (QoR)[2].

3.2.1 VHDL

Hardware Description Languages (HDL) describe the FPGA hardware. Two of the most widely used ones are Verilog and VHDL. They bear some resemblance to standard programming languages but they are *simulation* languages rather than (sequential) programming languages, i.e. HDL statements are not executed in sequential order during simulation but may be executed in parallel. Furthermore, simulation languages must support the notion of real time, either as an absolute clock or as cycles. VHDL in particular straddles the fence between general-purpose programming languages and hardware description languages and has a rich and verbose syntax.

3.2.2 C-level Languages

C is a sequential language. To introduce parallelism in C, one can introduce manual pragma's (ImpulseC, www.impulsec.com; Catapult C, www.mentor.com), or explicit parallel constructs (Handel-C, www.celoxica.com), or use sophisticated compilers or a combination of the above. Furthermore, timing signals are added to the standard ANSI-C compilers. As compared to ANSI-C compilers, full IEEE floating point arithmetic is usually not supported, nor pointers; however, arbitrary width variables and arithmetic are, as are signals and channels.

3.2.3 Mobius

Mobius (www.codetronix.com) is a Pascal-like CSP language, based on Hoare's Communicating Sequential Processes (CSP) and as such includes channels, communications and synchronizations but also types, records, arrays, channels, timers, and floating point arithmetic. It can generate both C and Verilog code.

[2] QoR is typically measured in terms of performance, size, power consumption, ...

3.2.4 QoR

How well do abstract, higher-level languages compare to cores that were originally developed in VHDL and then further fine-tuned? The answer is mixed. In Figure (4), we show the QoR for the Data Encryption Standard (DES) algorithm. DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key (although the effective key strength is only 56 bits). DES has 16 rounds or permutations – and two additional rounds of pre- and post-permutations – meaning the main algorithm is repeated 16 times to produce the ciphertext (as the number of rounds increases, the security of the algorithm increases exponentially). Typically, DES cores are either sequential (they minimize area but produce results only every 18 cycles) or parallel (deeply pipelined cores that produce 64-bit plain text every cycle but at a cost of area). QoR for sequential DES are within a narrow range; however, the variation in QoR is much larger for the parallel DES. It is worthwhile pointing out that pipelined versions

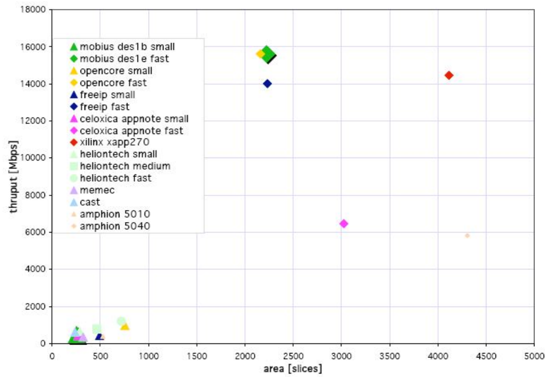

Figure 4: QoR for the DES encryption. Data collected by and courtesy of Codetronix LLC.

operate very close to peak numbers of current memory buses.

The Advanced Encryption Standard (AES), also known as Rijndael, is a block cipher adopted as an encryption standard by the US government, and is expected to be used worldwide and analysed extensively, as was the case with its predecessor, the Data Encryption Standard (DES). AES is fast in both software and hardware, is relatively easy to implement, and requires little memory. As shown in Figure (5), there are many commercial and academic cores available for it. The throughput results for Mobius are noteworthy, close to some of the best cores.

3.3 System Administration and Management

For systems adminstration and management, Cray makes available a Linux API for the FPGA with the following functionality:

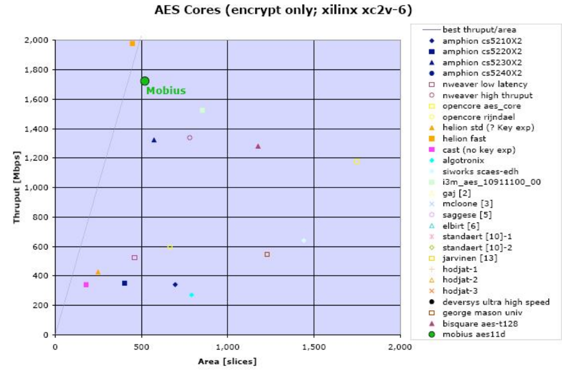

Figure 5: QoR for the AES algorithm. Data collected by and courtesy of Codetronix LLC.

- Admininstration Commands

  fpga_open: allocate and open fpga

  fpga_close: close allocated fpga

  fpga_load: load binary into fpga

- Control Commands

  fpga_start: start fpga (release from reset)

  fpga_stop: stop fpga

- Status Commands

  fpga_status: get status of fpga

- Data Commands

  fpga_put: put data to fpga SRAM

  fpga_get: get data from fpga SRAM

- InterruptBlocking Commands

  fpga_intwait: blocks process waits for fpga interrupt

4 Applications and Application Development Strategies

Our examples of applications and application development strategies were selected to illustrate different aspects of the FPGA expansion module for the Cray XD1: the interaction between the general purpose microprocessor (AMD/Opteron)and the FPGA co-processor; how to exercise the DMA engine between AMD and FPGA in high-speed fashion; and how to develop algorithmic strategies for the FPGA. A 4th example, where it all comes together, will be presented separately [5].

4.1 Basic Linear Algebra

It is well appreciated that the SAXPY (and DAXPY) basic linear algebra routines, $Y \leftarrow a \times X + Y$, test the memory subsystem on current microprocessors,

especially when exercised with non-unit stride. FPGA implementations are certainly not expected to outperform such microprocessors for two reasons:

1. Bandwidth will continue to be a bottleneck. Indeed, as shown in Table (1), the streaming AMD-to-FPGA bandwidth for write operations – while substantially better than PCI or PCI-X – are at about 70% of peak. The read bandwidths are substantially worse, due to some AMD/Opteron limitations. Together these results suggest to use write-only approaches for exchanging data between logic and processor;

2. IEEE 32 and 64-bit arithmetic is expensive on FPGAs. The best 64-bit IEEE cores (K. Underwood, private communication) run at close to 200 MHz on the Xilinx XC2VP50 and achieve 1 result (addition or multiplication) per cycle but are deeply pipelined (15 to 30 stages) and require ≈ 1000 slices. Nevertheless, current research [13, 14, 15, 3] suggests great progress in closing the gap between CPU and FPGA floating-point performance.

	array	pointer	memcpy
read	5.94	5.95	6.01
write	1260.	1320.	1320.

Table 1: Bandwidths (MB/s) for read and write operations between host and QDR SRAM. From [12].

4.2 Communication Protocols and Libraries

Cray provides pre-synthesized and placed netlists for the cores for the QDR RAM and Rapid Array fabric interface (QDR2 Core, resp., RT Core). These are low-level device drivers. On top of these interface cores, communication protocols between SMP node and FPGA have been developed that can use any or all of the following memory-based methods:

- The application maps a region on the FPGA's SRAM into its own address space and makes normal references to it;

- The application allocates a contiguous block of DRAM within its own address space that can be accessed directly by the FPGA;

- The application can read and write to individual registers provided by the FPGA logic.

Given these protocols, it is now possible to implement put/get communication libraries between SMP and FPGA that have functionality similar to the well known shmem libraries:

- Put and Get Operations: remote write (put) and remote read (get) operations;

- Non-blocking Put and Get Operations: to initiate multiple concurrent transfers between SMP and FPGA, perform useful work while the transfers are in progress and then wait or poll for their completion;

- Atomic Memory Operations: to allow atomic operation on shared variables (Atomic Memory Operations). These functions perform atomic read-and-update and swap operations on a remote data object, usually a register;

- Memory Allocation: to allocate memory at the same address in each PE and to free memory allocated previously.

Work is in progress [9] towards similar communication libraries.

4.3 Convolutions

Many FPGA implementations for image reconstruction and processing are regarded as valuable intellectual property, but the fundamental operations are similar in most cases. These algorithms include global transforms such as Radon transforms, 1D and 2D discrete Fourier transforms; semi-global transforms such as convolution filters; and simple pixel manipulations. We focus here on convolutions.

A 2D-convolution is written as:

$$Y_{i,j} = \sum_{n=-N}^{N} \sum_{m=-M}^{M} f_{n,m} X_{i-n,j-m} \qquad (1)$$

where X is the input data, f the filter coefficients, constituting a sliding window (typically, $N, M = 3, 4, \ldots 7$, but in more extreme cases of the order of a few ten) against the input data, and Y the result. With appropriate choices of the filter coefficients, convolutions have been used for edge detection, smoothing, noise reduction, etc. in image processing. Furthermore, convolutions can be used to implement fixed point multiplications: given a multiplier in the range $(2^{N+1}-1, \ldots, 2^{-N})$, its approximate binary representation $[b_N, b_{N-1}, \ldots, b_{-N}), b_i = 0, 1]$ is used as a 1D filter of size $2N+1$.

Here, we focus on 1D convolutions for three reasons:

1. It was shown [1] that one can use Singular Value Decomposition (SVD) to reduce the 2D convolution to approximate a 2D filter with 2x1D convolutions and one low complexity 2D one;

2. In many practical implementations, 2D convolutions are implemented as a string of 1D convolutions with appropriate delay lines;

3. In important applications [4] like 3D depth migration, it was shown how long, one-dimensional extrapolation filters can replace 2D filters, albeit in iterative fashion.

4.3.1 The Gather Approach

Almost all implementations – whether the target platform is a general purpose processor or logic device – of the 1D convolution (appropriately rewritten as $y_i = \sum_{n=-N}^{N} f_n x_{i-n}$) take a very literal approach: within the convolution unit, the $2N+1$ input data $x_{i-N}, \dots, x_{i+N}$ get multiplied with the appropriate filter coefficients and the results are summed together – in binary tree-like fashion – in an adder unit, Σ. We believe the main disadvantage of this approach is that the Σ unit contains multiple stages, adding to the overall depth of the convolution unit and not utilizing all available adders for a fully parallel operation.

4.3.2 The Scatter Approach

Instead we prefer a scatter approach, shown in Figure (6) for the case $N = 4$. Here, each input datum x_i is inspected for the contributions it makes to the output data $y_{i-N}, ..., y_{i+N}$, gets multiplied with the appropriate filter coefficient while the results are accumulated. When the next input datum, x_{i+1} enters the convolution unit, the accumulated result y_{i-N} leaves the unit. Clearly, the unit is not as deep as in the gather approach and all adders and multipliers are utilized in parallel.

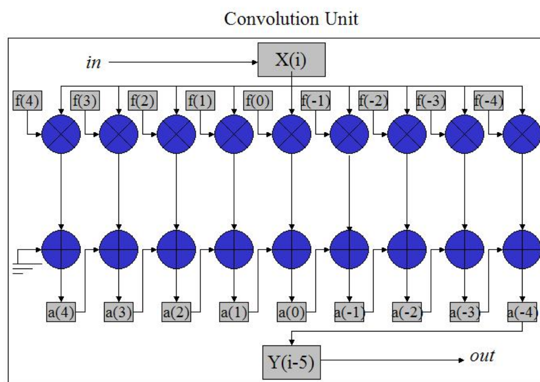

Figure 6: Schematic representation of the data flow and logic of a 1D convolution

4.4 Dynamic Programming: Smith-Waterman

This fully worked out dynamic programming example [5] will be presented at another session at CUG.

5 Linear Temporal Logic – The Logical and Computational Aspects of a Natural Language Interface for Verification

The previous sections demonstrated the need for a high level of abstraction in designing and developing for reconfigurable architectures. A number of high-level languages were described that benefits the ease-of-use, time-to-market, and QoR of programmable logic development. However, we believe that the application of formal verification techniques will yield additional and significant benefits in algorithm and system design for FPGAs.

Verification is not testing; testing is the practice of running the system and checking that it performs as expected. Testing becomes rather cumbersome, if not unreliable, for complex systems. Verification, however, connects the model with its specifications and generates a positive answer if the model complies with the requirements and a counterexample otherwise. Verification is done by a model checker. However, formal verification requires deep computer science expertise that hinders its wide-spread deployment. Furthermore, verification is a hard problem which suggests that one should aim for a methodology that satisfies the specification by construction, i.e., a correct-by-construction methodology.

We suggest that a remedy for this situation is the adaption of a natural language interface for hardware or program verification. We further suggest that such an interface already exists and, in fact, has a proven track record. Indeed, *Linear Temporal Logic* (LTL) [8] quantitatively and succinctly analyzes or synthesizes the behavior of complex systems in a language similar to natural languages. LTL has proven very effective as a specification language in the concurrency and computer-aided software engineering industries.

5.1 Linear Temporal Logic

Linear Temporal Logic (LTL) specifies the temporal behavior of tasks in a finite state machine in terms of *formulae* to construct first-order predicate logic and arithmetic assertions which evaluate TRUE or FALSE. Elementary formulae in LTL are TRUE, FALSE, and p where p belongs to the set of predicates. Those typically are events e or activities a. Propositional Logic (the mathematics of Boolean values TRUE and FALSE and the logical operators, $\vee, \wedge, \neg, \equiv, \Rightarrow$) is augmented with predicate logic (with quantifiers $\exists$ and $\forall$) applied to discrete time. The application of predicate logic on formulae results in a new formula.

It can be shown that the above procedures lead to an inductive definition of LTL formulae:

- TRUE, FALSE, and p are LTL formulae;

- If ϕ_1, ϕ_2 are LTL formulae, so are $\phi_1 \wedge \phi_2$ and $\neg\phi_1$;
- If ϕ_1, ϕ_2 are LTL formulae, so are $\circ\phi_1$ and $\phi_1 \, \mathsf{U} \, \phi_2$;
- There are no other LTL formulae.

Here the unary operation $\circ\phi$, or $\mathsf{N}\phi$, is pronounced *next* ϕ and evaluates TRUE if ϕ is true at the next instance while $\phi_1 \, \mathsf{U} \, \phi_2$ reads ϕ_1 *until* ϕ_2 and is satisfied when ϕ_1 is statisfied until ϕ_2 is satisfied.

In the simple example below, we show how numerical operations – the 1D convolution – can be expressed in terms of LTL formulae. With the definitions and formulae given as in the previous Section 4, our observations are the filter coefficients f, the incoming data stream x, and the results of additions a, and multiplications m. The output formula y is then given by:

$$\begin{aligned} m_{i,k+1} &\equiv \circ m_{i,k} = \circ(x_k * f_i); \\ a_{i,k+1} &\equiv \circ a_{i,k} = (a_{i-1,k} + m_{i,k}); \\ y_{k-5} &= (\circ(a_{-4,k}) \wedge (k > 5)) \; \mathsf{U} \; (k > N + 5). \end{aligned} \tag{2}$$

5.2 Forward Looking Statements

There is a substantial body of work, starting from the original work of Pnueli [8], demonstrating the usefulness of LTL in the verification of concurrent programs such as network communication protocols and operating systems [2, 6, 7]. Recent work [10] also suggests the applicability of LTL to complex dynamical systems and networks.

Until recently, languages used to specify design properties have been largely proprietary and limited to electronic design. One notable exception is the temporal logic languages LTL (and also CTL, Computation Tree Logic). In fact, these languages served as the basis for some of the proprietary languages that were developed later. However, electronic designers are intimidated by the scientific notation and the rather abstract nature of these academic languages. Nevertheless, the emergence of LTL in what has been traditionally high performance computing disciplines such as modeling and simulation of complex and concurrent dynamical systems suggest a common framework for application programmers and electrical engineers alike.

6 Conclusion

We described the Cray XD1 architecture with emphasis on its Reconfigurable Computing capabilities. We showed how current system software, libraries, compilers, and tools lower the effort required in programming logic devices. We developed a framework for applications development in RC that is familiar to the HPC programmer. We suggested that a natural language interface for program verification will shorten the program development cycle.

Acknowledgements

We would like to acknowledge input from Per Ljung of Codetronix, LLC, and Jim Rutt of the Santa Fe Institute. Steve Margerm of Cray Canada contributed to our understanding of the Xilinx Integrated Software Environment.

References

[1] C-S Bouganis, G. A. Constantinides, and P. Y. K. Cheung. A Novel 2D Filter Design Methodology for Heterogenous Devices. In *Proceedings of IEEE Symposium on Field Configurable Computing Machines*, 2005.

[2] E. Clarke, D. Peled, and O. Grumberg. *Model Checking*. MIT Press, 1999.

[3] Y. Dou, S. Vassiliades, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating point fpga matrix multiplication. unpublished, 2005.

[4] D. Hale. 3d depth migration via mcclellan transformations. *Geophysics*, 56(11):1778–1785, 1991.

[5] J. Maltby, J. Chow, and S. Margerm. FPGA Acceleration of Bioinformatics on the XD1: A Case Study. In *Proceedings of the Cray User Group Meeting*, 2005.

[6] Z. Manna and A. Pnueli. *The temporal Logic of Reactive and Concurrent Systems: Specification*. Spreinger-Verlag, Berlin, 1992.

[7] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[8] A. Pnueli. The temporal logic of programs. In *Proceedings of the* 18th *Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[9] D. Strenski, M. Babst, and R. Swift. Evaluation of running FFTs on the Cray XD1 with attached FPGAs. In *Proceedings of the Cray User Group Meeting*, 2005.

[10] P. Tabuada and G. J. Papas. Linear time logic control of discrete-time linear systems. unpublished, 2005.

[11] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations. In *Proceedings Supercomputing 2003*, 2003.

[12] J. L. Tripp, H. S. Mortveit, A. A. Hansson, and M. Gokhale. Metropolitan Road Traffic Simulation on FPGAs. In *Proceedings of IEEE Symposium on Field Configurable Computing Machines*, 2005.

[13] K. D. Underwood. Fpgas vs cpus: Trends in peak floating-point performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, pages 171–180, February 2004.

[14] K. D. Underwood and K. S. Hemmert. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In *Proceedings of the IEEE Symposium on Field-Programmable Custome Computing Machines (FCCM 2004)*, April 2004.

[15] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on fpgas. In *Proceedings of the* 18th *International Parallel and Distributed Processing Symposium (IPDPS'04*, pages 94–103, April 2004.