Reconfigurable Computing for High Performance Computing Computational Science

Song Jun Park, Brian Henz, and Dale Shires
US Army Research Laboratory (ARL), High Performance Computing Division, Aberdeen Proving Ground, MD
{song.jun.park, bhenz, dshires}@arl.army.mil

Abstract

Parallel computing systems with thousands of processors are now common and more affordable due to the focus on clustered commodity processors. However, both market and physical factors are converging that will limit the performance of these systems in the future. Hardware advances over the past several decades have been empirically observed with remarkable precision to obey Moore's law, predicting an increase in transistor density by about a factor of two every eighteen months. Maintaining these improvements has become problematic as power dissipation and other size-limiting factors become more pronounced at smaller feature size. Reconfigurable computing, or heterogeneous computing, is offering hope to the scientific computing community as a way to provide continued growth in computing capability. This paper discusses some of the hardware and software associated with this new technology. It also provides a discussion on the overall state of the technology for use by computational scientists. This is done by exploring a sample problem related to bit- and integer-based applications.

1. Introduction

High performance computing (HPC) is an evolving field that usually involves the use of parallel computing systems to solve difficult computational problems. Scientific research performed by the Department of Defense (DoD) can be greatly enhanced through the use of these computing systems. HPC as a research field in itself and as a means to further scientific research had a natural migration to parallel computing systems. This is somewhat obvious given that "The most powerful computer at any given time must, by definition, be a parallel machine"[1].

Parallelism provides a way to overcome clock speed limitations of serial processing units. This parallelism can be straightforward, where many distinct processing elements are tied together using fast interconnects or it can also be covert and performed within the CPU itself through instruction-level parallelism. Because of the various delays in performing complex mathematical operations and stalls waiting for memory accesses, today's CPUs have the capacity to interweave instructions at each clock cycle and hence have numerous operations scheduled simultaneously. Finding these opportunities in algorithms is complex and highly machine dependent where the process is usually relegated to high-level language compilers.

Computational science and scientific computing are fields of study exploiting these computing systems for various disciplines, have evolved and grown considerably over the past decade. These areas have made great advances using the increasing speed and performance of parallel systems as they have increased their processor counts and CPU speeds at the individual level. However, if the focus remains on using traditional computing architectures and software development paradigms, future growth and scalability into the realm of petaflop computing with traditional computing architectures looks bleak. Note the ability to achieve anything close to theoretical peak speeds has steadily declined. Furthermore, software engineering proves to be very difficult as the requirements to maintain portability and efficiency coexist with the overall requirement of mapping to complex architectures.

Reconfigurable computing is an emerging area of interest to HPC that will overcome many of these problems. Reconfigurable computing allows for the use of flexible digital design fabrics to allow for greater processing power over traditional fetch-execute-store architectures. However, programmability remains a barrier for effective deployment of this methodology to the general HPC community. This paper briefly discusses the current conditions in the HPC community that have led to an increased relevance for reconfigurable solutions. We discuss our experiences with the technology and how it was used for a target application area dealing with bit- and integer-based computing.

0-7695-3088-5/07 $25.00 © 2007 IEEE

2. The State of HPC

Hardware advances over the past several decades have been empirically observed with remarkable precision to obey Moore's law (predicting an advance in scalar performance by about a factor of two every eighteen months), but maintaining this rate has become problematic as power dissipation and other size-limiting factors become more pronounced at smaller feature size. Furthermore, the complex mix of memory bandwidth, interconnection latencies, and general algorithm scheduling at the compiler/processor level are all factors increasing the difficulty of achieving the advertised performance of massively parallel hardware.

The challenges at producing viable code for faster and more complex architectures are just as daunting, if not more, than those found in the hardware design. The difficulties range from job scheduling, task or domain decomposition, load balancing, and management of unwieldy data sets. Software engineering has become a much more time-consuming task for the computational scientist. Doubts about the ability of the message-passing model of parallelism to work well with petaflop architectures have led several vendors to begin research into new languages such as Chapel and X10 to target these next generation machines[2]. While these approaches may prove effective, it is unclear how they might affect developmental cycles. It has been noted that successful HPC software development projects routinely require a time span of about a decade[3].

The current state of HPC looks good at first glance because of the tremendous computing capacity. However, there is a widening gap in terms of capability. The gap is getting wider by the day as newer approaches in hardware, such as the multi-core chip, make their appearance on the market. While the capacity is growing, the capability (as found in the software) is still lacking. Suggestions on how best to handle the new reality focus largely on traditional software engineering practices to lessen the risk of new hardware insertion and extend application lifecycles[4].

A promising approach to push HPC to the next level is found by paying attention to both hardware and software in a unified way. Reconfigurable computing blends custom hardware design with high-level language approaches to see the process as one unified endeavor. This is a fundamentally different way of viewing HPC that certainly requires more work, but the potential reward is considerable.

3. Reconfigurable Computing for HPC

Reconfigurable computing is a rapidly maturing field that offers great hope for meeting the demanding requirements of computational scientists. In general, it is an attempt to blend custom hardware design with high-level language approaches. The use of highly flexible computer fabrics gives algorithm designers the ability to make substantial changes to both data and control paths:

> *"Reconfigurable computing is computer processing with highly flexible computing fabrics. The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the data path itself in addition to the control flow"*[5].

Of course, these lofty descriptions have to find their way into practical uses. This section discusses the hardware that is being used to achieve these goals. The technology has been in existence for some time and has matured to the point of being useful for the HPC community. We conclude this section by discussing the programming methodology for these systems and the complexity involved.

3.1. Hardware

A Field Programmable Gate Array (FPGA) is a reconfigurable device composed of Configurable Logic Blocks (CLB) and programmable interconnects. An FPGA is usually supplied as a stand-alone board or as a component that can be plugged into a host. Two manufacturers, Xilinx and Altera, have a majority of the FPGA market. The actual FPGA board can come in a wide variety of sizes with different internal and external interfaces. Figure 1 shows a simple FPGA training board that connects through serial ports to a host computer. The board has LEDs and several switches and buttons that can be polled by the interface libraries. The actual FPGA on this board is a Xilinx Spartan.

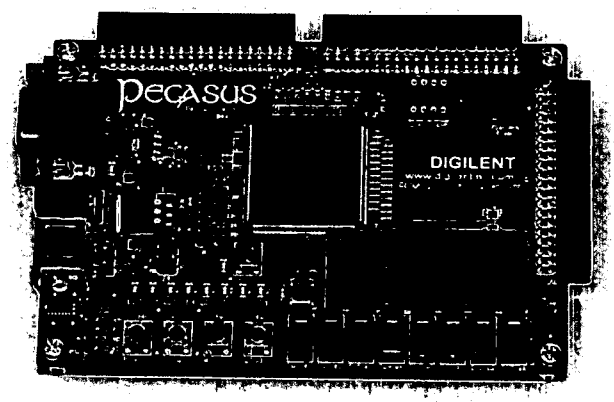

Figure 1. A sample FPGA training board

CLB can be programmed to implement any desired logic functions that use a memory technology to store desired output values. Programmable interconnects surround the individual logic blocks and allows for user-defined connections. Thus, a combination of configurable logic blocks and programmable interconnects is able to create any logic network. Figure 2 illustrates how a NAND gate operation can be constructed with transistors or with a memory element.

Figure 3 is a simplistic graphical depiction of connected gates that are formed after an algorithm is placed on a FGPA. An algorithm is encoded as a digital design and mapped to a switching fabric made up of logic cells. FPGAs are designed to emulate integrated circuits. If an algorithm is "synthesizable" and burned on a chip, it can also be "simulated" in an FPGA.

Any digital function can be mapped to an FPGA but developers have historically shown a preference for integer- and bit-based operations. This is mainly due to the limited size of the fabric, but this is changing. Since the basic component of an FPGA is static RAM, therefore, these devices roughly track the speed of SRAM technology with clock speeds currently up to 500 MHz[6].

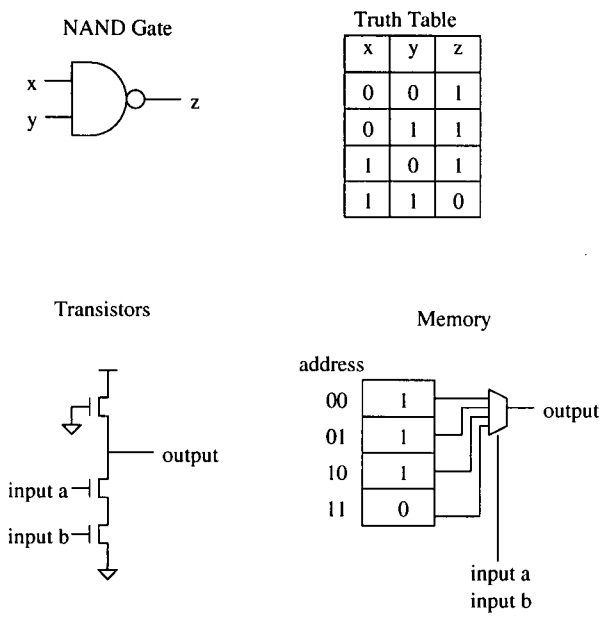

Figure 2. NAND gate implementation with transistors and with a memory unit

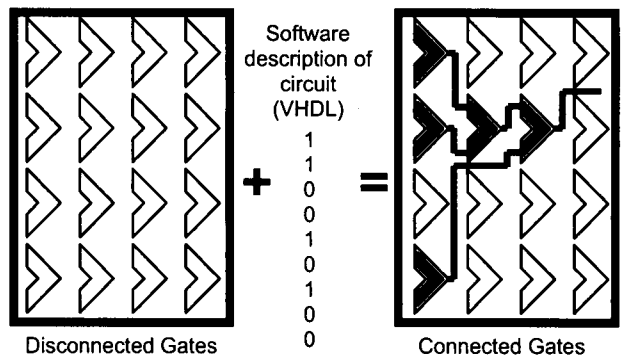

Figure 3. A graphical representation of mapping digital circuit descriptions to FPGA fabric

Traditionally, a processor executes computations on a fixed architecture. For an FPGA, the design of architecture is required before computations can be executed. Hence, FPGA technology implies the use of dedicated hardware for performing computations. The process is similar to the design of an application specific integrated circuit (ASIC) with an important difference of the reprogrammable functionality of FPGAs. However, this flexibility of FPGAs results in a lower clock frequency when compared to ASICs or processors.

While the outlook for traditional CPUs into the future is a bit uncertain, the converse is true for FPGAs. DeHon's law is the analog of Moore's law that governs performance on reconfigurable chips[7]. It observes that the computational capacity of reconfigurable hardware grows at a rate roughly twice that of general-purpose CPUs.

We have provided many examples of where FPGAs differ from traditional CPUs. CPUs essentially do one thing at a time, at very high clock speeds whereas FPGAs can do a multitude of things, at a slower clock speed. The traditional CPU in use today follows the von Neumann sequential processing paradigm. That is, instructions are fetched and executed and results are stored in a continuous sequential fashion. The fact that memory is separate from the processing unit in this architecture leads to a condition known as the von Neumann bottleneck; these are the delays from reading and writing to main memory. The addition of cache memory has been promoted over the years to alleviate some of this forced-wait, spin-idle time.

FPGAs overcome these sequential barriers by allowing circuit designers to operate both in space and time. Algorithms are laid out in space on reprogrammable hardware. Instruction-level parallelism is the most obvious form of speedup resulting from this capability. Multiple copies of the same computation can be carried out simultaneously by unrolling or strip-mining the heavily used loops in an algorithm. Incredibly deep

pipelines can also be formed for more step-wise operations.

Speedup through other means is also possible on large FPGA fabrics. Separate tasks can be executed concurrently with no data or time dependencies. There is also the potential for reduced latency as potentially large data structures can be laid out and maintained on lookup tables in FPGA fabric vice tiered memory systems. Stall cycles are removed along with the von Neumann bottleneck, essentially disappearing on the FPGA.

3.2. Development Environment

FPGAs have a somewhat complex development environment that is certainly different from the traditional code development environment in the general community. FPGAs are usually programmed using a Hardware Description Language (HDL). Two of the most prevalent languages in use are VHSIC Hardware Description Language (VHDL) and Verilog. The two vary in their approach to coding. While VHDL has a similar look and feel to that of the Ada programming language and is widely used in DoD-related efforts, Verilog has the look and feel of the C programming language. A small piece of example VHDL code is given in Figure 4.

Hardware description languages are concurrent and parallel languages. In a concurrent language environment, the order of statements is irrelevant because a code is not executed line by line. A code written in VHDL or Verilog, attempts to describe the architecture and interconnects of a hardware system, creating the hardware description language.

The development cycle for FPGA software is also different from those found in traditional software engineering for generic processors. First, parallel algorithms require a slightly different focus from designers who are used to thinking sequentially. Even current parallel programming paradigms, such as message-passing, do not map well to the underlying structure of the language. Rather, for those familiar with parallel models, it is similar to the Parallel Random Access Machine or the implied Single Instruction Multiple Data model[8]. That is, all processors are executing the same instruction during the same time slice but using potentially different data streams.

```
-- sqrt8.vhdl   unsigned integer sqrt 8-bits
computing unsigned integer 4-bits
--    sqrt(00000100) = 0010   sqrt(4)=2
--    sqrt(01000000) = 1000   sqrt(64)=8
library IEEE;
use IEEE.std_logic_1164.all;

architecture circuits of Sm is
  signal t011, t111, t010, t001, t100, td :
std_logic;
begin  -- circuits of Sm
  t011 <= (not x) and y and b;
  t111 <= x and y and b;
  t010 <= (not x) and y and (not b);
  t001 <= (not x) and (not y) and b;
  t100 <= x and (not y) and (not b);
  bo   <= t011 or t111 or t010 or t001;
  td   <= t100 or t001 or t010 or t111;
  d    <= td when u='1' else x;
end architecture circuits;  -- of Sm
```

Figure 4. Example VHDL code

The next step involves compilaation and simulation to verify the operations of a design. A simulator typically generates a waveform of signals within a design. After the process of debugging and simulation, a design is synthesized. During the process of synthesis, hardware description language code is translated into a network of logic gates. Depending on the complexity and size, a synthesis process can take hours to days to complete.

Designing an application for an FPGA using HDL requires a prerequisite knowledge of digital logic and circuit theory, however, high-level languages, similar to the programming language C, are being introduced as a substitute for HDL. In this methodology, a designer is allowed to write a code resembling a high-level language in syntax and format. Then, the code is compiled to produce the corresponding hardware counterparts in VHDL or Verilog.

One such language is from Celoxica and is called Handel-C. Handel-C is a C-like language that follows the ANSI-C syntax and semantics with extensions and restrictions to specify hardware design. It is designed to produce efficient hardware, provides for synchronization, and allows one to use arbitrary word widths. Key to the Handel-C approach is the `par` statement that expresses what should happen in parallel. Figure 5 shows example code with the serial, 4-cycle block shown in (a) as compared to the same code in (b) that can execute in parallel in 1 cycle using the `par` construct.

Another language is from Mitrionics called Mitrion-C. Mitrion-C has its own syntax and takes an algorithmic description approach[9]. Moreover, Mitrion-C is a fully parallel programming language similar to hardware description languages. The code in Mitrion-C is targeted for the highly configurable Mitrion virtual processor, which gets programmed into an FPGA. This approach simplifies the compiler's work. Currently, the virtual

processors have a fixed clock frequency set at 100 MHz. Also, Mitrionics provides support for the Cray XD1 architecture by removing the user's responsibility of having to of implement the core interface components.

```
// 4 clock cycles
  {
  i = 0;
  a = 1;
  b = 23;
  c = 99;
  }
```

(a) Standard C syntax serial code.

```
// 1 clock cycle
par {
  i = 0;
  a = 1;
  b = 23;
  c = 99;
  }
```

(b) Handel-C code representing a parallel region.

Figure 5. Standard C serial code versus Handel-C parallel constructs

Nallatech offers another high-level FGPA development approach. Nallatech provides DIME-C and DIMETalk environment to a developer with interest in programming FPGA devices. DIME-C compiles and generates VHDL output of an algorithm written in DIME-C, which is a subset of standard C. An advantage of using DIME-C is that a user only needs to learn the C statements that are supported in DIME-C. DIMETalk provides an abstraction of the PCI-X interface, translating its communication channel in a form of a network. Communication and control of the FPGA design is accomplished through Nallatech's Field Upgradeable Systems Environment application programming interface (API) function calls executing on a host side.

4. Reconfigurable Computing in Practice

For all but the most simplistic circuit designs, FPGAs will act together with traditional CPUs to form a heterogeneous, reconfigurable architecture for the next generation of parallel computers. The most compute-intensive segments of software will be offloaded to an optimized architecture of FPGAs for hardware acceleration.

4.1. Hardware Systems

Desktop solutions with FPGAs are already available for a host of problems. The traditional areas where custom-built FPGA hardware accelerated computing is making an impact include signal and image processing, software-defined radio, aerospace, bioinformatics, and cryptography. Large-scale speedups can already be seen on applications running on single FPGAs in these areas. A factor of a 100 speedup is not uncommon in open literature.

However, there is no reason to stop at single FPGAs to solve difficult problems. Recently there has been a push to extend the viability of FPGA-based solutions into the parallel world. HPC systems with FPGAs are being fielded in the DoD High Performance Computing Modernization Program. Several Cray XD1 systems are available with clusters of over 100 FPGA processors. These machines will be used by the authors to investigate the use of coupled commodity clusters and FPGA devices in a parallel system.

According to Cray, one of the three main obstacles to the adoption of reconfigurable computing is Peripheral Component Interconnect (PCI) bus bottlenecks or data starvation to the FPGA (the other two being job scheduling and programmability)[10]. Indeed data starvation is a problem in some FPGA cases where traditional bus speeds through PCI connections are lacking. Cray has implemented a proprietary system known as RapidArray to connect the compute processors to the FPGAs over high bandwidth, low-latency links. In the personal computer or workstation market, vendors are moving to PCI-Express solutions for FPGAs. Currently, the majority of boards offer Peripheral Component Interconnect Extended (PCI-X) solutions to improve bandwidth between an FPGA and a processor. This is an intermediate step prior to full PCI-Express support.

4.2. Application Areas

Whether it be one FPGA or hundreds connected in a cluster, it is safe to say that all but the simplest applications will involve some combination of generic processors operating in conjunction with FPGA hardware. Approaching computation-intensive programs for optimization is already handled in this fashion. Here, researchers generally look for a 90/10 rule in their code that is surprisingly common across domains. The 90/10 rule states that roughly 90% of the total execution time of a piece of software resides in only 10% or less of the total source lines of code. This 90/10 offloading rule will be the fastest way to identify what code segments could possibly be shipped to the FPGA for fast execution. There should be no shortage of possible application areas.

Often researchers, when first encountering FPGAs, are hesitant to investigate their use, due to the low clock speeds associated with the devices. The low clock speed is often misleading but does provide some insight on what application might be appropriate to target for FPGAs. For example, consider a CPU at 4 GHz versus an FPGA clocking at 400 MHz. Assume that the CPU takes one cycle to produce an interesting result. One can easily see that it would take an array of 10 Processing Elements (PEs) in the FPGA fabric to match the performance of the CPU. Of course, it usually takes well over one cycle for a CPU to produce an interesting result. It must serially read

the data from memory and write any results back. Other overheads such as the incrementing of induction variables on loops and associated loop overhead for branch and bound easily add to the clock quantum to produce interesting results. Overall, with today's technology, a 10 PE rule of thumb helps to determine viable FPGA applications.

5. Technology Evaluation

To evaluate the use of FPGAs for the Army, we are focusing on two distinct application areas. The first deals with integer and bit-based computing technology. Here we will be targeting encryption algorithms, steganalysis, and data mining for intrusion monitoring and protection. The ability to hide and encode messages using standard encryption and ciphers is a major problem for Army intelligence. Furthermore, the sheer size of data collected on network traffic sensors is overwhelming. The task of mining this data to uncover a potential computer security breach is extremely time-consuming using conventional computing.

The power of FPGAs to directly tackle these problems with incredible speedup is the major drawing force. We will be working with the Army Research Laboratory's Center for Intrusion Monitoring and Protection to identify applications for studying FPGA utility in data mining. We have already identified the "Blowfish" encryption code as a first-cut in analyzing FPGAs in single and parallel mode against standard implementations of this algorithm on commodity chips. We will move into steganalysis upon completion of these initial efforts.

The second main focus area will be on floating-point intensive applications. Currently, we are targeting a Classical or Quantum Monte Carlo (QMC) algorithm to implement in hardware to compare against computers having reported high FLOP-rate capabilities. QMC is a good candidate for several reasons. First, it is broadly representative of scientific computing algorithms. Second, its structure, which allows fine and coarse grain parallelization, pipelining, and calculation with integer or fixed point data representation makes it a sound fit for FPGAs. Finally, QMC is very time-consuming and can easily take advantage of savings from hardware acceleration with incredible impact.

6. Application Design

6.1. Encryption Algorithm

The Blowfish algorithm is a fast symmetric block cipher[11]. Unique and notable features of Blowfish include key dependent S-boxes and a highly complex key schedule. A time consuming key schedule process makes Blowfish an ideal hash function candidate for password authentication. Although Blowfish is known as a block cipher, the algorithm supports a hash operation by using a user key as an input, ranging from 32–448 bits and getting a fixed 64 bit output.

6.2. VHDL Hardware Design

For Blowfish, hardware structure and design choices depend strongly on the intended purpose of the algorithm. Consider a standard execution of encryption and decryption functions with constant secret key for a set of data. In this case, a pipeline structure enhances efficiency and throughput where a predetermined secret key maintains the key-dependent S-boxes constant at every level of the pipeline. The opposite is true when the algorithm is used with the intention of performing a brute force attack. With an objective to determine an unknown user key, the attack continuously guesses a different secret key. Accordingly, S-boxes must be recalculated for each particular key under examination. Due to key and S-box dependency, S-boxes are not predefined identical lookup tables, but a changing entity. Unlike the process of encrypting or decrypting messages, key recovery of Blowfish spends a majority of its time pre-computing key and S-boxes. This pre-processing is equivalent to encrypting 4,168 bytes of text.

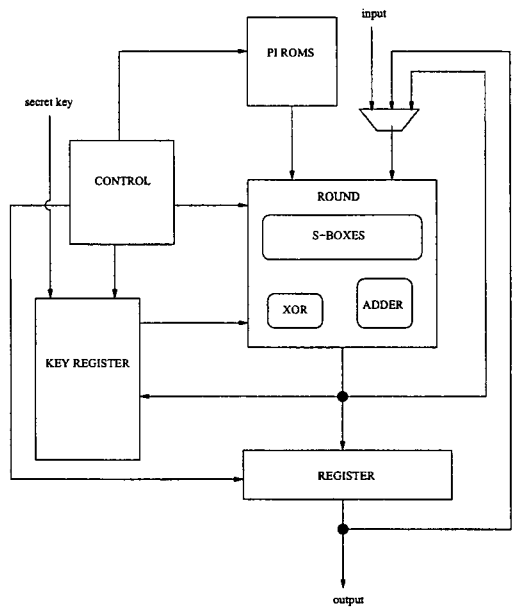

Figure 6. Top-level Blowfish architecture

The major elements of the hardware design consist of S-box, datapath, key register, and control. The design layout is shown in Figure 6. The specifications for the S-box are 32-bit output with 8-bit address inputs. During

preprocessing, the data path generates 64-bit output to replace previous values of S-box. Since a standard 32-bit RAM unit does not support writing in 64-bit within one cycle, two 32-bit S-boxes with a multiplexer controlled output are used to support an optimal 64-bit load. The seven most significant bits of the address are applied to both S-boxes and the least significant bit is connected to the select signal of the multiplexer. Basically, a 32-bit S-box capable of storing 256 entries is substituted with two 32-bit S-boxes each holding 128 entries, as shown in Figure 7. The preprocessing stage assigns new values for all data within the S-boxes which equals to a total of 1,024 entries. The impact of S-box loading capability is significant. With the ability to load 64-bit values in one clock cycle, the S-box write operation completes in 512 cycles. As for the 32-bit data loading per cycle, write operation would require 1,024 cycles.

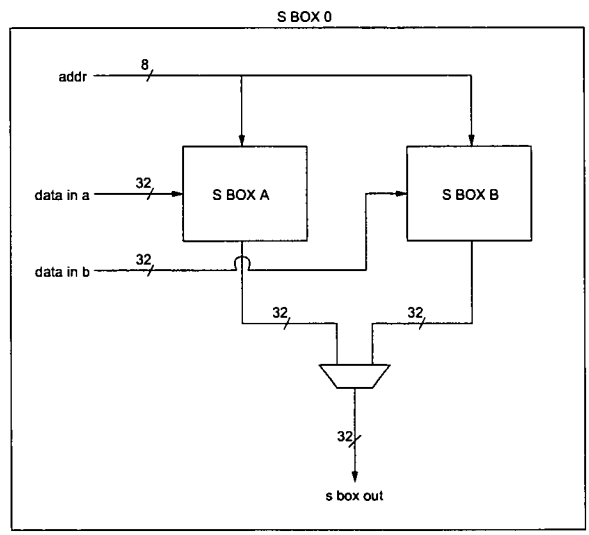

Figure 7. Optimized S-box design

6.3. VHDL Simulation

During simulation, the current state of S-boxes, output of registers, and the value of the signals at various stages need to be checked and verified with the correct values. Intermediate results are an essential element during the debugging process of a hardware design. Intermediate values were retrieved from the Blowfish code written in standard C. ModelSim, an HDL simulator, produces a window displaying waveforms that represent the inner values of the hardware design. These waveforms were analyzed and compared with the expected values obtained from the C code for validation.

6.4. Hardware Issues

The size of hardware for the Blowfish algorithm demanded large resources to implement, especially S-boxes for sixteen rounds of Blowfish. Each S-box contains 256 entries, each entry storing a 32-bit value. For sixteen rounds of the Blowfish algorithm, there are total of 64 S-boxes. Instead, a reduced version of one round was designed. This hardware is then reused to implement the 16 rounds of the Blowfish algorithm.

As previously discussed, mapping S-boxes to configurable logic blocks of an FPGA locks up large amount of FPGA resources. An alternative option would be to use the block RAMs, which are dedicated on-chip RAM modules within the FPGA. The downside of using block RAM is that it only allows one write per cycle, which slows down the loading of initial hex values of "pi". Another disadvantage of block RAM is the synchronous read characteristic: the output of the S-box appears one clock cycle after an input is applied. Thus, computing 16 rounds takes 17 clock cycles for the block RAM design. Additionally, key register values and necessary control signals must be stored and forwarded to the next clock cycle due to the one cycle delay of a block RAM. Figure 8 describes the timing relating to S-boxes mapped to block RAMs.

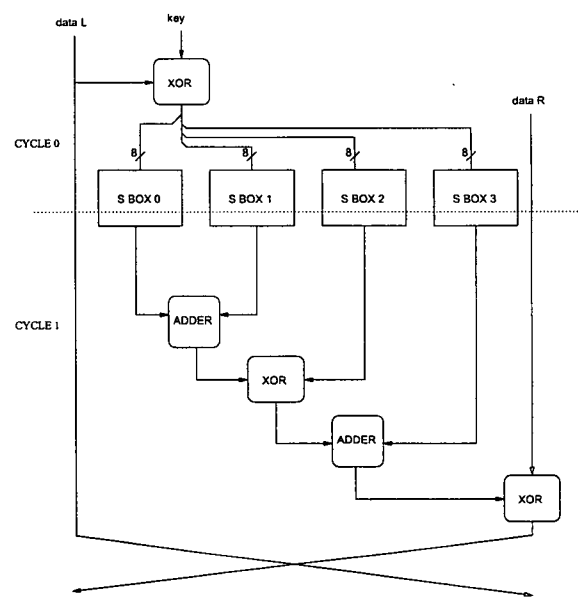

Figure 8. Timing of dataflow

The number of clock cycles needed to complete a one key test of Blowfish algorithm is composed of: the secret key load, pre-processing, and encryption/decryption. The time required for loading secret key depends on the input and output (I/O) specification. For example, 64-bit I/O for a secret key needs 9 clock cycles to finish loading a

512-bit key register. A separate exclusive-OR cycle is not necessary because input of a secret key is applied to an exclusive-OR before being latched into the key register. Preprocessing is divided into "pi" initialization, key register setup, and S-box setup. Total number of clock cycles for VHDL design equals 9,012, which is summarized in Table 1.

Table 1. Required Clock Cycles

Operation	Clock Cycles
Secret Key Load	9
Pi Initialization	129
Key Register Setup	153
S-box Setup	8704
Encryption/Decryption	17
Total	**9012**

The entries for S-boxes are key dependent such that original values are replaced with a new value derived from a secret key. For the purposes of performing a key attack, the pipeline is of no use due to the key dependent nature. Every pipeline stage would require different S-box values, which translate to different hardware for each stage. Recall that preprocessing of S-boxes is a major portion of computation for Blowfish key attack. Defining 16 rounds of execution to be 1 run of Blowfish algorithm, encryption takes 1 run, and total pre-processing takes 521 runs, which equals to about 99.8% of time spent on the pre-computing step. Therefore, a method more aligned with the key attack would be to use a single stage of Blowfish and to parallelize by duplicating processing units as permitted by available area. Here, the multiple single round designs are each reused iteratively to achieve a similar effect of a pipeline with a startup I/O limitation.

6.5. FPGA Development Flow

The overview procedure for programming FPGAs can be broken down into three tasks: the hardware design, the FPGA interface, and the host program. The hardware design refers to the process of designing an application in hardware. Traditionally, designers use HDLs, but compilers are being introduced that translate a code written in a high-level language to a code in a hardware language format. Secondly, the user application must be connected to architecture specific interface devices for FPGA communications with its interconnected components. Interface core components provide a method and functionality to control, read, write, and monitor an FPGA. Finally, a host program is written in standard C with vendor specific API functions to load and execute the hardware design on an FPGA device.

Each of previously described steps can be accomplished in many different ways. For instance, a design of hardware can be done using VHDL, Verilog, DIME-C, Mitrion-C, or Handel-C to name just a few. In addition, the architecture and interface cores differ with each company along with corresponding API functions. Thus, the lack of a dominant standard within FPGA community imposes challenges for effectively utilizing FPGA technology.

6.6. Synthesis and Performance Results

The Blowfish algorithm was written following the imposed rules of DIME-C. DIME-C does not support pointers and the concept of call by reference. Values passed as function arguments will be modified instead of a copy being created. Since DIME-C is a subset of ANSI C, debugging can be performed using a standard GNU compiler. For debugging and simulating DIME-C code, the design was wrapped around a main function which was responsible for providing the inputs.

In order to generate a binary file to program a FPGA, DIME-C code is translated into VHDL and imported into a DIMETalk network. At minimum, a DIMETalk network consists of a PCI-X interface, clock driver, memory block, and a user hardware component. After the creation of a network, DIMETalk synthesizes the design, translating VHDL code into logic gates, and builds a binary file. When this build process completes, the area and delay results along with sample host API file are placed under the current working folder. A sample API host file performs basic hardware tests, board reset, binary file programming, and execution of the design. The synthesis results of the Blowfish algorithm, designed in VHDL and DIME-C for Virtex-4 LX100, are summarized in Table 2.

Table 2. Virtex-4 LX100 Utilization Summary

		Used	Percentage Used
VHDL	Slices	3273	6%
	BRAM	40	16%
DIME-C	Slices	13715	27%
	BRAM	52	21%

Table 3 lists the measured execution time of a software version running on a Xeon 3.0 GHz microprocessor and hardware versions running on a Virtex-4 FPGA using VHDL and DIME-C languages. Execution time reflects the amount of time required to hash one key value using the Blowfish algorithm. DIME-C design is much slower than both VHDL and software versions. However, the core of the Blowfish algorithm is mostly involved with data dependent functions resulting

in highly sequential operations, allowing little room for hardware optimizations. In addition, the area of the FPGA fabric was not fully utilized, which can achieve faster execution by duplicating identical hardware units.

Table 3. Performance Comparison

	Clock Frequency	Execution Time
ANSI C	3.0 GHz	54 us
VHDL	84 MHz	90 us
DIME-C	53 MHz	2300 us

7. Conclusion

The HPC community is currently facing a capability gap that is only going to get worse. There are numerous hardware and software development challenges that lie ahead as we attempt to construct larger computer systems to focus on computational science applications to key Army requirements. Reconfigurable computing holds the promise of a solution, but it will take a substantial effort to reach maturity. Within the next three to four years we foresee more focus on this methodology with success stories coming from the many modeling and simulation codes currently running on commodity clusters.

Acknowledgements

The authors wish to thank those individuals from the User Productivity Enhancement and Technology Transfer Program of the DoD High Performance Computing Modernization Program and the Naval Research Laboratory who assisted in arranging various FPGA-related training events.

References

1. Carriero, N. and D. Gelernter, *How to Write Parallel Programs: A First Course*, The MIT Press, Cambridge, MA, p. 5, 1992.

2. HPCWire, "HPCS Languages Move Forward." Aug. 2006; http://www.hpcwire.com/hpc/827250.html.

3. Post, D., "The Coming Crisis in Computational Science." *Proc. of the IEEE Int'l Conf. on High Performance Computer Architecture: Workshop of Productivity and Performance in High-End Computing*, Madrid, Spain, 2004.

4. Meyer, R., "Emerging Multi-core Realities." *Scientific Computing*, Aug. 2006.

5. Wikipedia, "Reconfigurable computing." Sep. 2006; http://en.wikipedia.org/wiki/ Reconfigurable_Computing.

6. Fernando, J., "Using FPGAs in High Performance Computing." Ohio Supercomputing Center, Lecture at Naval Research Laboratory, 2006.

7. DeHon, A., "The Density Advantage of Configurable Computing." *IEEE Computer*, vol. 33, no. 4, pp. 41–49, 2000.

8. JaJa, J., *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, 1992.

9. Tripp, J., M. Gokhule, and K. Peterson, "Trident: From High-Level Language to Hardware Circuitry." *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

10. Cray Inc., "Cray XD1 Supercomputer for Reconfigurable Computing", 2005.

11. Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)." *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, pp. 191–204, 1994.