ARMY RESEARCH LABORATORY

Reconfigurable Computing: Experiences and Methodologies

by Song Jun Park, Dale Shires, and Brian Henz

ARL-TR-4358 **January 2008**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-4358 **January 2008**

Reconfigurable Computing: Experiences and Methodologies

Song Jun Park, Dale Shires, and Brian Henz
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
January 2008	Final	October 2006–November 2007

4. TITLE AND SUBTITLE
Reconfigurable Computing: Experiences and Methodologies

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Song Jun Park, Dale Shires, and Brian Henz

5d. PROJECT NUMBER
7UE DCL

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRD-ARL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-4358

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Reconfigurable computing refers to computations done with flexible fabrics where the data path and control flow can be customized to the application. Unlike traditional computing using the fetch, execute, and store model that is highly sequential, reconfigurable computing allows developers to program their applications both spatially and temporally. This allows for potentially great speedups with applications that might be well-suited for such approaches. However, programming in this style requires specialized hardware and a somewhat complex design flow. This report discusses background on the topic and highlights our experiences using this technology on two target applications. It also discusses the state-of-the-art high-level language approaches that have been offered to streamline the development cycle using these technologies.

15. SUBJECT TERMS
reconfigurable computing, hardware acceleration, FPGA, encryption, sorting

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Song Jun Park
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	34	19b. TELEPHONE NUMBER (*Include area code*) 410-278-5444

Contents

List of Figures v

List of Tables vi

Acknowledgments vii

1. Introduction 1

2. FPGA Background 2

3. FPGA Programming 3

4. Reconfigurable Systems 3

5. Evaluation of High-Level Languages 4

5.1 Nallatech....4

5.1.1 DIME-C....4

5.1.2 DIMETalk....5

5.1.3 FUSE API....6

5.1.4 Design Flow....7

5.1.5 Nallatech Experiences....8

5.2 Mitrionics....8

5.2.1 Mitrion-C....9

5.2.2 Simulator....9

5.2.3 Mithal....9

5.2.4 Mitrionics Experiences....10

5.2.5 Comparison....10

6. Applications 11

6.1 Blowfish Algorithm....11

6.1.1 VHDL Hardware Design....12

6.1.2 VHDL Simulation....12

6.1.3 Hardware Issues....14

6.1.4 DIME-C Implementation....15

6.1.5 Mitrion-C Implementation....15

6.1.6 Blowfish in Hardware 16
6.1.7 Blowfish Hardware Results 17
6.2 Mstack 18
6.2.1 Mstack Implementation 18
6.2.2 DIME-C Optimizations 19
6.2.3 Synthesis Setback 19
6.2.4 Mstack FPGA Results 19

7. Closing Thoughts on Hardware Acceleration 20

8. Conclusions 20

9. References 22

Distribution List 23

List of Figures

Figure 1. NAND gate construction. 2
Figure 2. Sample DIME-C source code. 5
Figure 3. DIMETalk network. 6
Figure 4. Host and FPGA communication. 7
Figure 5. Sample Mitrion-C source code. 9
Figure 6. Mitrionics simulator. 10
Figure 7. Top-level blowfish architecture. 13
Figure 8. The 64-bit S-box design. 13
Figure 9. Timing of dataflow. 14
Figure 10. Cray design hierarchy. 16

List of Tables

Table 1. Development time estimates. 11
Table 2. Comparison of the languages/approaches 11
Table 3. Required clock cycles. 15
Table 4. Performance comparison. 17
Table 5. Single-unit area utilization on FPGA 18
Table 6. Performance results on a Virtex-4 XC4VLX100 FPGA. 20
Table 7. Virtex-4 LX100 area utilization 20

Acknowledgments

The authors wish to thank individuals at the U.S. Naval Research Laboratory for their assistance with accounts and training relating to the Cray XD1 and Mitrionics software development toolkits.

Mstack was developed and provided by Mr. Daniel Pressel. He also provided the CPU performance results given in table 6.

Computing time was sponsored in part from a grant of resources by the Department of Defense High Performance Computing Modernization Program Office.

INTENTIONALLY LEFT BLANK.

1. Introduction

Over the course of the last decade, high performance computing architectures have dramatically shifted away from vector processors to scalar units. This change occurred mainly due to shifts in the commodity markets and the increase in performance of scalar processors. Whereas vector processors are designed to execute mathematical operations on numerous data items simultaneously, scalar units can process one element at a time. Superscalar architectures, such as the MIPS R8000, have also been developed over time, where a limited number of simultaneous instructions per cycle could be issued. This allowed for a small amount of instruction-level parallelism and speedup per clock cycle on scalar designs. By June 2006, 98.4% of the largest parallel computer systems were using scalar processors mainly in a cluster configuration (*1*).

Cluster computing allows for parallelism at a coarse-grained level. Applications in various fields such as bioinformatics and computational structural mechanics can potentially benefit from having numerous processors work on subsets of an overall problem domain. These results are then merged during a final step to achieve the overall problem results. However, the overall speedup from using many processors is limited by the time to distribute data and communicate among the processors. It is also limited by the individual processor speeds and algorithm execution on the scalar processors.

While scalar processors typically follow the von Neumann architecture approach of fetch, execute, and store, reconfigurable computing refers to processing with the aid of programmable logics, usually in the form of a field programmable gate array (FPGA) (the scope of this report does not include hardware-accelerated computing as found with devices such as graphical processing units [GPUs]). With an FPGA, data path and control flow can be modified as necessary to reduce an algorithm's execution time. Instead of computing through a series of instructions, a series of logic gates is created to solve a problem. By coupling an FPGA with a processor, compute-intensive applications can be off-loaded from a host central processing unit (CPU) to an optimized architecture of an FPGA. This integration allows an FPGA to function as a powerful coprocessor and collaborate with a main CPU for performance acceleration.

FPGAs are not really programmed in the traditional sense of the word. Rather, they require a hardware description language that has the concepts of concurrence built in. Due to the time-intensive development workflow required from hardware descriptions, several vendors are developing and refining procedural programming languages that down-compile to hardware implementations or descriptors like VHSIC hardware description language (VHDL). In this report, we look at DIME-C and Mitrion-C for programming FPGAs. The test bed systems used in this study include the Cray XD1 featuring Virtex-II Pro FPGAs and the Nallatech H101-PCIXM board featuring a Virtex-4 FPGA.

2. FPGA Background

An FPGA is a reconfigurable device composed of configurable logic blocks and programmable interconnects. Configurable logic blocks can be programmed to implement any desired logic functions that use a memory technology to store output values. Programmable interconnects surround the individual logic blocks and allow user-defined connections. Hence, the combination of configurable logic blocks and programmable interconnects is able to create any logic gate network. Figure 1 illustrates how the NAND gate function can be constructed with either transistors or memory elements.

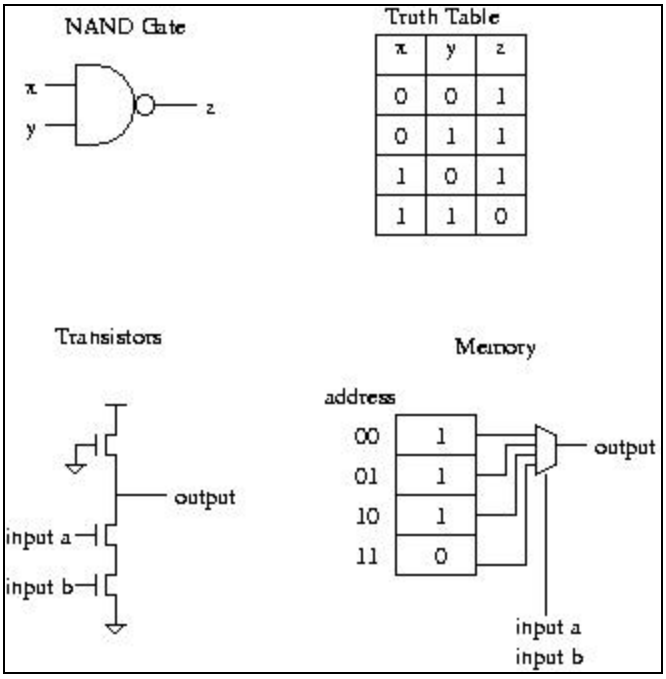

Figure 1. NAND gate construction.

For an FPGA, a specific architecture is adopted for targeted computations as opposed to computing on a fixed architecture of a scalar processor. Therefore, 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 difference of reprogrammable functionality in FPGAs. Unfortunately, lower clock frequency is the cost of having a flexible computing fabric. Mainstream FPGAs are built on static random access memory (SRAM) technology and operate at associated speeds (on average, 400 MHz). This lower speed is misleading when comparing the technology to scalar processors. The wide and deep pipelines possible with FPGAs can provide speedup to overcome the shortfall in clock rates.

3. FPGA Programming

The overall procedure for programming FPGAs can be broken down into three tasks: core hardware design, FPGA interface specification, and host program development. First, an application of interest is designed in hardware. Traditionally, designers use hardware description languages for hardware design. Second, the user application must be interconnected to an architecture-specific interface in order for the FPGA to communicate with its surrounding components. These interface core components handle communications between a host and an FPGA. Finally, a host program is written in standard C with vendor-specific application programming interface (API) functions to load and execute the hardware design on an FPGA device.

Each of the previously described steps can be accomplished in many different ways. For instance, a hardware design can be done using VHDL, Verilog, DIME-C, Mitrion-C, or Handel-C to name just a few. The two dominant hardware languages are VHDL and Verilog. Hardware description languages are inherently concurrent and parallel languages that require a different mind set than the sequential programming languages that are targeted for a CPU. In a concurrent language environment, the order of statements is irrelevant because statements are not executed in order, line by line. A code written in VHDL or Verilog simply describes the architecture and interconnects of a hardware system; hence the name hardware description language.

The design for an FPGA using a hardware description language requires the prerequisite knowledge of digital logics and circuits background. To attract broader users, high-level languages, similar to the programming language C, are being introduced as a substitute for hardware description languages. In this methodology, a designer is allowed to write a code resembling a high-level language's syntax and format. Then, the code is compiled to produce the corresponding hardware counterparts in VHDL or Verilog.

4. Reconfigurable Systems

The Nallatech H101-PCIXM hardware is equipped with a Xilinx Virtex-4 LX100 and 4 MB of DDR-II SRAM (*2*). The card is connected to a 133-MHz-capable peripheral component interconnect extended (PCI-X) interface inside a Linux workstation. The field upgradeable systems environment (FUSE) API functions, developed by Nallatech, control the FPGA board.

The Cray XD1 combines microprocessors and FPGAs with the low-latency and high-bandwidth RapidArray interconnect (*3*). RapidArray interconnect is a high-speed switch fabric that improves PCI-X bus bottlenecks and shared resource contention. Currently, the Cray XD1 at the U.S. Naval Research Laboratory is configured with 144 nodes of Virtex II Pro FPGAs and 6 nodes of Virtex 4 FPGAs (*4*).

5. Evaluation of High-Level Languages

In an apparent effort to field their technologies into newer and more diverse areas, FPGA vendors are trying to overcome the barriers found in programming their devices by using high-level language development methods. Several examples exist—Nallatech, Mitrionics, and Celoxica, to name a few. The various vendor-specific approaches used are evaluated and compared with the VHDL method. This report explores the Nallatech and Mitrionics approaches.

5.1 Nallatech

Nallatech offers the DIME-C and DIMETalk software development tools to ease the process of programming applications on an FPGA. Currently, DIME-C and DIMETalk are supported only on a Windows platform. Since a large amount of system memory is used during the synthesis process when executing Xilinx software, 64-bit Windows or Linux operating systems are recommended for the software installation.

5.1.1 DIME-C

DIME-C is the high-level language developed by Nallatech. Nallatech provides the software development environment to edit, compile, and translate a DIME-C code into VHDL. An attractive feature of DIME-C is that it is a subset of the American National Standards Institute (ANSI) C with the identical syntax. To get started, a user only needs to learn the portion of the standard C language that is not supported in DIME-C. The following ANSI C elements are not supported in DIME-C (*5*):

- pointers
- structures
- enumerated types
- switch statements
- all labeled statements (e.g., case default)
- the type qualifiers const and volatile
- the storage-class specifiers auto, register, and typedef
- the type specifiers long and double
- the unary expression sizeof
- the jump statements goto, continue, and break

- the conditional expression “?”
- string literals

These constraints allow for a minimal learning curve to start coding in DIME-C. However, to exploit hardware optimizations, a certain coding style must be followed for parallelism and pipelines to be applied to the design as described in the DIME-C user guide. Upon compiling and translating DIME-C code, a color coded graphical representation indicating parallel and pipelined structure is generated along with estimated area of FPGA slice requirement. Since the language is a subset of ANSI C, DIME-C code can be debugged using any ANSI C compiler. Figure 2 shows a sample code written in DIME-C.

```
for (i = 0; i < 18; i += 2)  {
    for (k=0; k<16; k++)  {
        data_l = data_l ^ p[k];
        data_temp = data_l;
        bit0_7   = data_temp & 0x000000FF;
        data_temp >>= 8;
        bit8_15  = data_temp & 0x000000FF;
        data_temp >>= 8;
        bit16_23 = data_temp & 0x000000FF;
        data_temp >>= 8;
        bit24_31 = data_temp & 0x000000FF;
        data_temp = s0[bit24_31] + s1[bit16_23];
        data_temp = data_temp ^ s2[bit8_15];
        data_temp = data_temp + s3[bit0_7];
        data_r = data_temp ^ data_r;
        temp = data_l;
        data_l = data_r;
        data_r = temp;
    }
}
```

Figure 2. Sample DIME-C source code.

5.1.2 DIMETalk

In order to configure an FPGA, a binary bit file must be generated through the synthesis process. But before creating a binary bit file, FPGA’s interface information needs to be specified and connected with the user design. DIMETalk provides a PCI-X host interface as a component within the DIMETalk network, allowing the connection from PCI-X to internal memory elements. Having the host interface component simplifies the implementation of the FPGA and host communication by hiding the details associated with PCI-X bus and its connections to the FPGA pins. Unfortunately, interface specifications vary among different vendors, which results in a portability issue.

The FPGA's internal and external interfaces are handled by the DIMETalk software. DIMETalk divides and represents an overall FPGA design as separate, individual components with predefined interconnects (*6*). At minimum, a clock driver, a host interface, a memory element, and user design modules are required to complete an FPGA design in Nallatech. The following list outlines and describes the minimum components of a DIMETalk network:

- Clock driver provides the clock signals for a design.
- Host interface defines the FPGA board interface to a host system allowing communication to an FPGA.
- Memory elements can store data and are visible and accessible by a host.
- User module defines an algorithm in hardware.

Basically, DIMETalk provides a workspace for placing and connecting elements of a hardware design, creating a network inside an FPGA (see figure 3). Also, this allows for a graphical method of creating multiple instantiations and managing multiple FPGA networks.

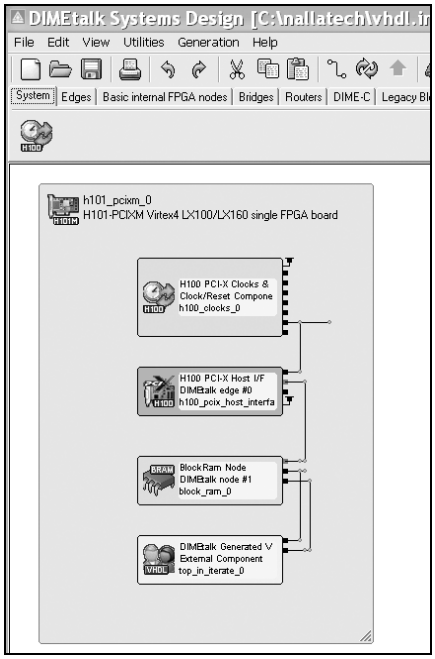

Figure 3. DIMETalk network.

5.1.3 FUSE API

The FUSE API functions, provided by Nallatech, are used for a host to communicate with an FPGA board. A standard C code, referred to as a host C file, contains these FPGA-specific API functions. Executing a host C file will load, control, and execute the hardware design on an FPGA. A generic host C file is created during the build process of a DIMETalk network, thus

relieving a developer from the details of initial FPGA setup. Besides the initial FPGA board setup, the host C file is responsible for initializing system memory and transferring data to an FPGA memory. A graphical representation of this communication is shown in figure 4.

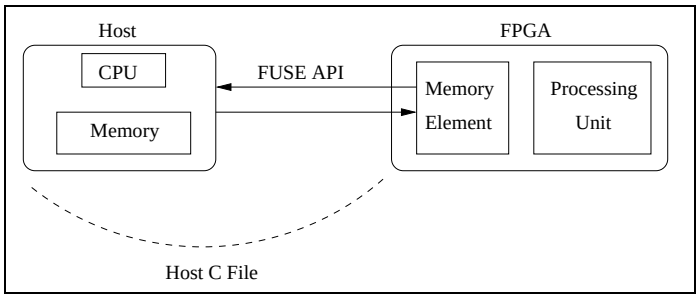

Figure 4. Host and FPGA communication.

The host C file is compiled to generate an executable that, once invoked, will start the reconfigurable computing process. In summary, the host C file is responsible for the following:

- FPGA board setup.
- Host system memory initialization.
- Data transfer from host system memory to FPGA memory.
- Triggering the FPGA process and monitoring for done signal.
- Transfer of computed results in FPGA memory to host system memory.

5.1.4 Design Flow

The components and steps required to run an application on an FPGA board are outlined below. The three design files needed to run an application in an FPGA are the hardware description file, an interface specification file, and a host C file. The major steps of Nallatech's design flow are as follows:

1. Development of a DIME-C source code describing an application.
2. Compilation of the DIME-C code into VHDL.
3. Importing VHDL that was created previously into DIMEtalk and creation of a network consisting of the minimum components which are interface, clock, memory, and imported VHDL.
4. Synthesis of the DIMEtalk network, generating the associated binary bit file.
5. Inserting data exchange and control API functions in the host C file.
6. Compilation of the host C file and start of the executable.

5.1.5 Nallatech Experiences

The tools offered by Nallatech attempt to simplify the hardware design process by offering the high-level language DIME-C, providing a PCI-X interface component, and creating a generic host C file. Furthermore, hardware optimizations are automated by the DIME-C compiler. Unfortunately, these user-friendly features result in a limited amount of hardware design flexibility for exploiting parallelism. The conversion from DIME-C to VHDL is an open-ended problem. For example, an addition operation can be accomplished using a carry-propagate adder, a carry-save adder, or a carry look-ahead adder. The ability to control low-level signals and components of hardware design gives the FPGA an edge over processors—something lost during automatic conversion. Since the DIME-C compiler is in charge of designing hardware for an algorithm written in DIME-C, hardware optimizations are limited to the ability of the DIME-C compiler to detect them.

An issue encountered with Nallatech's software is the lack of a test bench unit to verify a generated hardware design that is translated from DIMC-C. Apparently, test bench functionality for the high performance computing FPGA boards is not yet available. The only option to verify a design on an FPGA is to wait for the synthesis to complete and then test the design by running it on an FPGA card. This process for the Blowfish algorithm took an excess of 1 hr to complete for any change applied to the design. The problem was exacerbated with the mismatch of DIME-C simulation output and FPGA output. The DIME-C version would produce the correct results, but the compiled and synthesized FPGA version of working DIME-C source code would produce incorrect results. Such a situation indicates immaturity of the DIME-C compiler.

The DIME-C compiler derives a hardware design from a subset of standard C, which is a procedural language. A procedural language is intended for programming processors and is fundamentally different from designing hardware. Although the intentions and benefits of using DIME-C for programming FPGAs are attractive, achieving performance improvement directly from a language targeted for a CPU can be a limiting factor.

5.2 Mitrionics

Mitrionics development tools include the Mitrion-C language, Mitrion-C simulator, and Mithal. Mitrionics software tools are supported in various platforms such as Windows, UNIX, Linux, and OS-X. Moreover, the Mitrion-C language and simulator are free to download. Mitrionics charges a license for the virtual processors that actually run a design on an FPGA. A unique aspect of the Mitrionics approach is that rather than directly translating a high-level language down to the hardware level, highly configurable virtual processors are inserted in between the translation process (*7*). The virtual processor is a fine-grain, parallel, and configurable soft-core processor with a fixed clock frequency of 100 MHz (*8*). Mitrionic's software tools support the Cray XD1 architecture, which removes the user's responsibility of implementing the FPGA to host interface components.

5.2.1 Mitrion-C

Mitrion-C is a high-level concurrent language similar to the hardware description languages without the requirement of knowing the digital design concepts. Its syntax is different from standard C as shown in figure 5. Mitrion-C is a single assignment language resulting in many temporary variables. When a Mitrion-C source code is compiled, it is translated into VHDL.

```
(data_ll, data_rr, mem_s0_02, mem_s1_02, mem_s2_02, mem_s3_02) = for (i
in <0..15> ) {
        data_k = data_l ^ p[i];
        bits0_7 = (bits:8) data_k;
        bits8_15 = (bits:8) (data_k >> 8);
        bits16_23 = (bits:8) (data_k >> 16);
                  bits24_31 = (bits:8) (data_k >> 24);
        (s0_out, mem_s0_01) = _memread(mem_s0_00, bits24_31);
        (s1_out, mem_s1_01) = _memread(mem_s1_00, bits16_23);
        (s2_out, mem_s2_01) = _memread(mem_s2_00, bits8_15);
        (s3_out, mem_s3_01) = _memread(mem_s3_00, bits0_7);
        f_out = ((s0_out + s1_out) ^ s2_out) + s3_out;
        data_l = f_out ^ data_r;
        data_r = data_k;
}(data_l, data_r, mem_s0_01, mem_s1_01, mem_s2_01, mem_s3_01);
```

Figure 5. Sample Mitrion-C source code.

5.2.2 Simulator

The graphical user interface simulator for Mitrion-C provides an intuitive graphical representation of dataflow and design structure (see figure 6). Stepping through the visual representation, a user can compare component utilization and the amount of achieved parallelism. In conjunction with the simulator, “watch” statements can be inserted into a source code to monitor internal signals within a design.

5.2.3 Mithal

The Mitrion host abstraction layer API is a C-based API for interfacing the Mitrion Virtual Processor and a host computer. A host C file consisting of Mithal API functions load, control, and execute a design on an FPGA. An example of Mithal API functions for writing and reading data to an FPGA is as follows:

```
mem = (WORD*)mitrion_processor_reg_buffer(p,"mem_a",NULL,NUM_WORDS *
      sizeof(WORD), WRITE_DATA);
mem2 = (WORD*)mitrion_processor_reg_buffer(p,"mem_b",NULL,NUM_WORDS *
       sizeof(WORD), READ_DATA);
```

A unique feature of the Mithal API is the ability to use a Mitrion simulator to mimic a design running on an FPGA, allowing the simulation of entire design interactions. Hence, the complete FPGA design can be simulated on a machine without any FPGA installed.

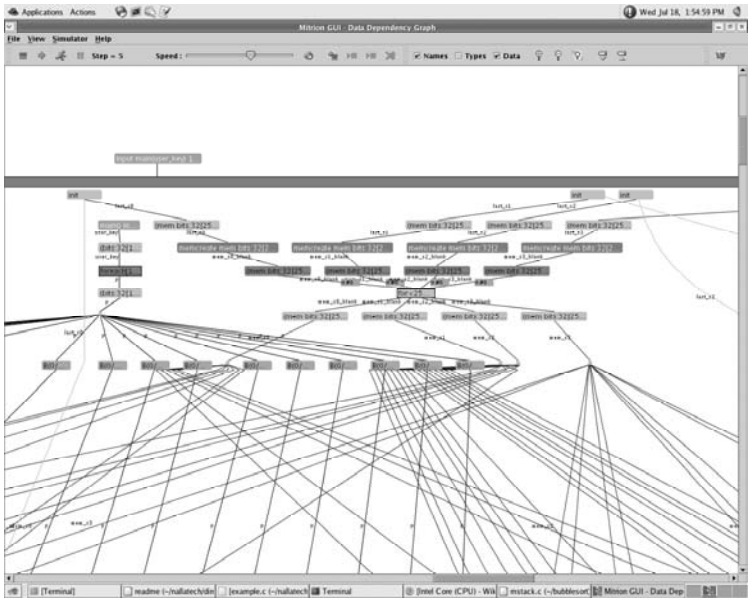

Figure 6. Mitrionics simulator.

5.2.4 Mitrionics Experiences

In the Cray XD1, communication between a processor and an FPGA is accomplished through the RapidArray interconnect. Thus, the RapidArray interface core must be included in a user design. VHDL source codes of interface core components, including RapidArray, are provided as templates with predefined port signals. Normally, a developer is responsible for processing the host or CPU's requests by including necessary interface components. However, the Mitrionics developer only needs to follow and match the bus width and the number of interface signals for a particular architecture within Mitrion-C code.

Although Mitrion-C looks similar to the procedural languages like standard C, a different mind-set is required for programming a parallel language. There is no order of execution, and parallelism is implicit in the Mitrion-C language. Initial coding experiences with Mitrion-C language indicate a complexity for implementing array manipulations and difficulty in tracking variable types. In addition, source code tends to be cluttered with temporary variables because every statement, including loops and block expressions, must return a value. Dividing and debugging portions of Mitrion-C code becomes cumbersome because the language manifests code dependency where commenting out a portion of the code can result in compile errors due to the dependencies in final memory instance tokens, final return values, and temporary variables.

5.2.5 Comparison

This section attempts to give a high-level, overall assessment of the various approaches. Naturally, each approach has its own set of advantages and disadvantages. We are mainly interested in performance and ease-of-use (time to solution)—two areas that, at times, seem to be

mutually exclusive. It should be noted that some of these assessments stem directly from the design mentality of the approach, while others reflect upon the maturity level of the approaches as they come from the various vendors.

Development time estimates are given in table 1. These categorizations follow after dealing with the approaches for 1 year. More detail is provided later in the performance of the approaches on certain target applications. At this point, the easier it is to code, the worse the sustained performance. The reader may wish to consult table 1 again after reviewing the performance data in later sections of this document. More general assessments are listed in table 2.

Table 1. Development time estimates.

Development Stage	VHDL	DIME-C	Mitrion-C
Background learning	High	Low	Medium
Writing source code	High	Low	Medium
Debug & simulation	Low	High	Medium
Applying design changes	High	Low	Medium
Maintaining	High	Low	Medium

Table 2. Comparison of the languages/approaches.

Language	Advantages	Disadvantages
VHDL	• High performance • Design flexibility • All hardware functionality available • Intended for programming FPGA	• Development time • I/O interface • Size • Unfamiliar to software developers
DIME-C	• Easy to code • I/O Interface core abstraction • Hardware optimizations automatic • Subset of standard C with same syntax	• Limited functionality • Primitive • Procedural language
Mitrion-C	• Reduced coding time • Raised abstraction level • User control of parallelism • Concurrent language (parallelism is default)	• Limited customization • Documentation • Short history

Note: I/O = input/output.

6. Applications

6.1 Blowfish Algorithm

The Blowfish algorithm is a symmetric block cipher introduced by Bruce Schneier (*9*). Details of the algorithm can be accessed at Schneier's Web site (*10*). Unique and notable Blowfish features include key dependent S-boxes and a highly complex key schedule. A time-consuming

key schedule 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 inputs in the range 32–448 bits and getting a fixed 64-bit output.

6.1.1 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 a 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-box 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. Rather, they are a changing entity. Unlike the process of encrypting or decrypting messages, key recovery of Blowfish spends a majority of its time precomputing key and S-boxes. This preprocessing is equivalent to encrypting 4168 bytes of text.

The major elements of the hardware design consist of S-box, data path, key register, and control. The design layout is shown in figure 7. The specifications for the S-box are 32-bit output with 8-bit address inputs. However, during preprocessing the data path generates 64-bit output to replace previous values of the S-box. Since a standard 32-bit RAM unit does not support writing of 64-bit in one cycle, two 32-bit S-boxes with a multiplexer-controlled output are used to support 64-bit loads. 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 8. The preprocessing stage assigns new values for all data within S-boxes—1024 entries. With the ability to load 64-bit values in one clock cycle, the S-box write operation completes in 512 clock cycles instead of 1024 clock cycles.

6.1.2 VHDL Simulation

During the simulation stage, the current state of the S-boxes, output of registers, and the value of the signals at various stages are checked and verified with the correct values. Intermediate results are an essential reference during the hardware design debugging process. Intermediate values were retrieved from the Blowfish code written in standard C. ModelSim, a Hardware Description Language 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.

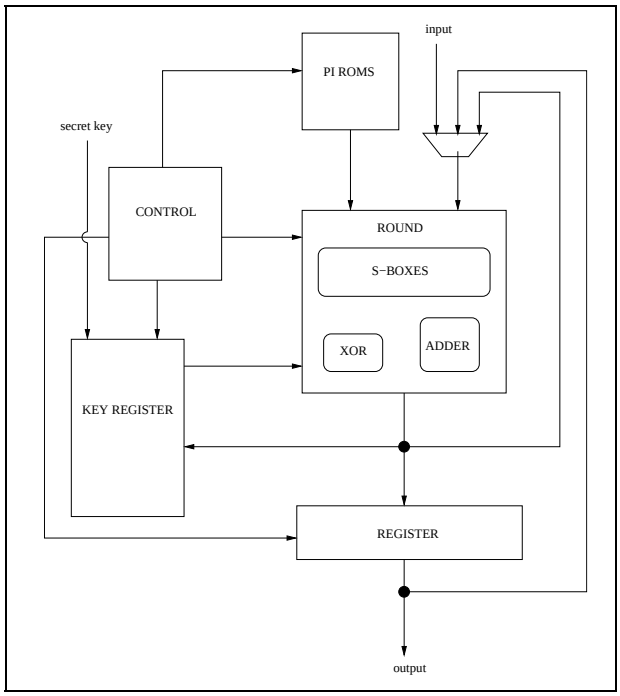

Figure 7. Top-level blowfish architecture.

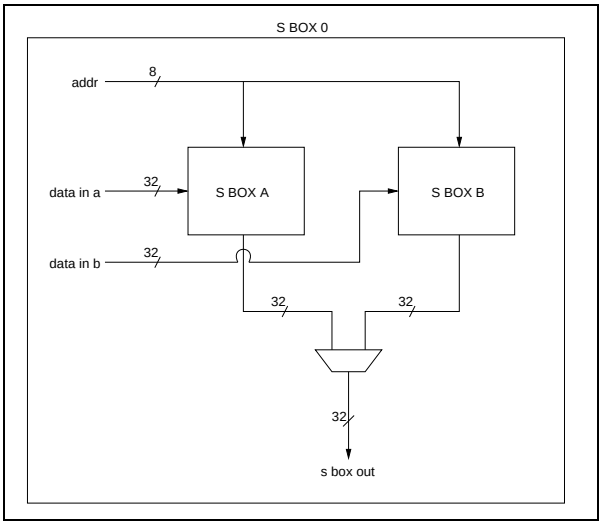

Figure 8. The 64-bit S-box design.

A testbench module tests various vectors with the help of a script file. A script file was generated with the help of Perl, which instructs the simulator by providing input vectors and expected answers. Testbench compares the hardware results with the expected answer. Error message is printed to the console window when the hardware result fails to match the expected answer. The test vectors were acquired from the Blowfish author's Web site (*10*).

6.1.3 Hardware Issues

The size of hardware for the full Blowfish algorithm demanded large resources for an FPGA, especially S-boxes for 16 rounds of Blowfish. Each S-box contains 256 entries where each entry stores a 32-bit value. For 16 rounds of the Blowfish algorithm, there are 64 S-boxes. Instead, the reduced version consisting of just one round was implemented. This hardware is then reused to implement the 16 rounds of Blowfish algorithm. Size limitation is a big concern for FPGA design. As more of the fabric is required to implement a design, the ability to replicate the design to promote parallelism and pipelining is decreased. If it is degraded to a high level, the slower clock speeds of the FPGA will not allow better performance over standard scalar CPUs with higher clock rates.

Mapping S-boxes to configurable logic blocks of an FPGA locks up a large amount of FPGA resources. An alternative option would be to use the block RAMs, which are dedicated on-chip RAM modules within FPGAs. The downside of using block RAM is that it only allows one entry write per clock cycle, which means 128 clock cycles are needed to load the initial 128 hex values of pi. Another disadvantage of block RAM is the synchronous read characteristic where 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 9 describes the timing relating to S-boxes mapped to block RAMs.

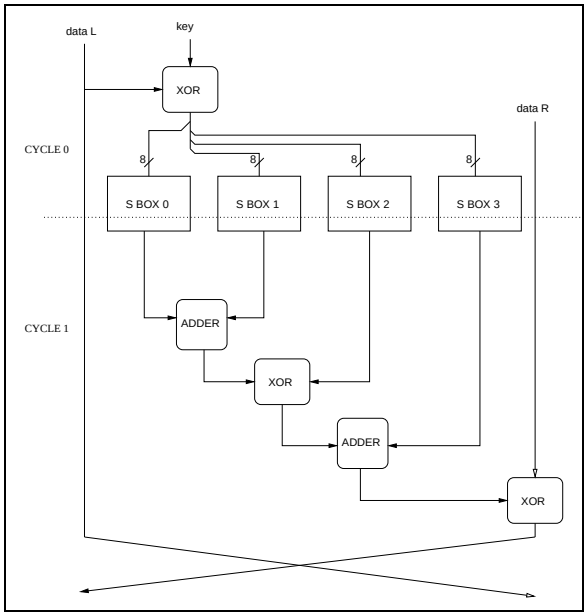

Figure 9. Timing of dataflow.

The number of clock cycles needed to complete a one-key test of the Blowfish algorithm is composed of secret key load, preprocessing, and encryption/decryption. Time required for loading the secret key depends on I/O specification. For example, 64-bit I/O for a secret key needs nine 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 exclusive-OR before being latched into the key register. Preprocessing is divided into pi initialization, key register setup, and S-box setup. The VHDL design requires 9012 clock cycles (table 3).

Table 3. Required clock cycles.

Operation	Cycles
Secret key load	9
Pi initialization	129
Key register setup	153
S-box setup	8704
Encryption/decryption	17
Total	9012

6.1.4 DIME-C Implementation

The Blowfish algorithm was written adhering to the rules specified by the DIME-C language. Mainly, DIME-C does not support pointers or the concept of call by reference, although arrays are supported. Values passed as function arguments will get modified instead of a copy being created. With pointers, it becomes more difficult for the compiler to track parallel use of hardware (*11*). Since DIME-C is a subset of ANSI C, debugging was performed using a standard GNU compiler. For debugging and simulating DIME-C code, the design was wrapped around a main function responsible for supplying input data.

In order to generate a binary file to program an FPGA, DIME-C code is translated into VHDL by the DIME-C compiler and imported into a DIMEtalk network as a user component. Within the DIMEtalk software environment, the DIME-C generated-user component is interfaced with the PCI-X core, clock driver, and memory block by connecting component ports with wires. After creating the network, DIMETalk synthesizes the design, which translates the network into logic gates and builds a binary file. When this build process completes, the area and delay results along with a sample host API file is placed under the current working folder. A sample API host file performs basic hardware tests, board reset, binary file programming, and design execution. Data transfer from a host to an FPGA is added to the sample host file, and the host file, is compiled to generate the final executable.

6.1.5 Mitrion-C Implementation

Unlike other high-level languages targeted for hardware, Mitrion-C is a concurrent language similar to hardware description languages. The Blowfish algorithm was written and debugged on a local workstation using the free Mitrionics compiler and simulator. After passing the

simulation, the Blowfish design in Mitrion-C was compiled and synthesized, following Mitrion's Cray XD1 document (*12*), on the Cray XD1 system equipped with Virtex-II Pro and Virtex-4 FPGAs.

The workflow design attempts to streamline the development process. Interface components and interconnects for the Cray XD1 are predefined by the core files in templates named Rapid Transport, User App, and quad data rate (QDR) SRAM. User design, which is the hardware design of a target application, is inserted and interfaced with the internal signals of the User App file. In other words, the outer skeleton, including interconnecting port signals of the interface core files, are preset by the vendor, and a user design is wrapped inside this predefined template. This concept is illustrated in figure 10.

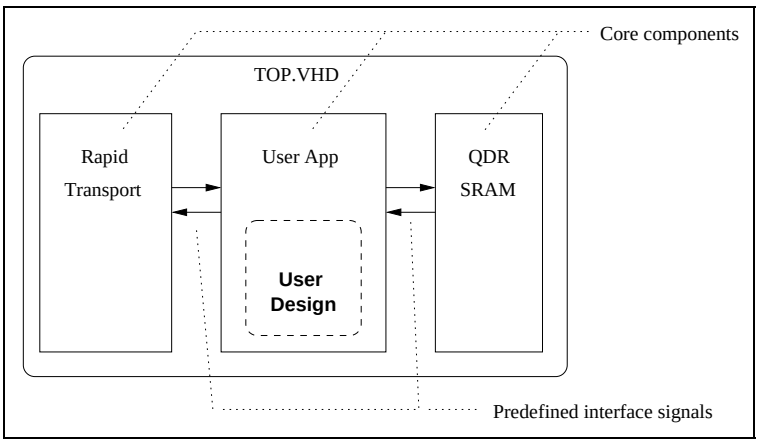

Figure 10. Cray design hierarchy.

6.1.6 Blowfish in Hardware

The S-box entries are key-dependent where original values are replaced with a new value calculated from a secret key. This S-box preprocessing is a major portion of computation for a Blowfish key attack. Suppose 16 rounds of execution are defined as one run in Blowfish algorithm. Then the algorithm can be viewed as encryption taking one run and total preprocessing taking 521 runs to complete. In other words, about 99.8% of the time is spent on the precomputing step. Key and S-box initialization of the Blowfish algorithm imposes a one-time start-up cost for encryption or decryption where the secret key remains constant. However, an attacker employing a brute force attack would need to perform the lengthy preprocessing every time for each different key. This inherently sequential process of key and S-box setup delivers extra protection and limits hardware acceleration.

Dedicated block RAM provided by Xilinx does not support reset initialization of the internal memory cell to pi; only output data of specified width is latched when reset is asserted. Normally, S-box with fixed internal values would be a good candidate for utilizing a block RAM

located inside a FPGA. Even normal Blowfish algorithm operations used for fixed-key encryption and decryption would be a good fit for using a block RAM, since start-up initialization is supported. But for the purpose of performing brute force attack, this block RAM property implies sequential reset of S-box pi internal values for every new key.

For attacking user passwords, S-box dependencies prevent gaining benefits from pipelining. Every pipeline stage would require different S-box values, which translate to different hardware for each stage. An alternative approach is to replicate one round of Blowfish such that each round independently computes its assigned input iteratively. The resulting structure is simply a collection of parallel and independent one-round design of Blowfish. The allowed number of copies depends on the available amount of configurable logic blocks and block RAMs within a particular model of an FPGA.

6.1.7 Blowfish Hardware Results

The Blowfish algorithm key search operation was implemented and executed on both the CPU and the FPGA. For programming the FPGA device, VHDL, DIME-C, and Mitrion-C languages were analyzed for comparison. Area and delay information are provided within a Place and Route file with an extension ending in .par. The Blowfish execution time focused on the core processing time, excluding host and FPGA data transfer. Performance results (see table 4) indicate that DIME-C version performs poorly in terms of execution time compared to other approaches. As for the area utilization, DIME-C and Mitrion-C designs require more space than hand-coded VHDL (see table 5), which means fewer instances of processing units can be duplicated for acceleration. Performance results show that a 32-bit data path and key dependent S-boxes of the Blowfish algorithm give a processor a distinct advantage in execution time.

Table 4. Performance comparison.

Language	Processing Hardware	Clock Frequency (MHz)	One Unit Execution Time (µs)	Processing Units	Throughput (keys/s)
ANSI C	Xeon	3000	54	1	18,518
VHDL	Virtex-4 LX100 FPGA	65	120	11	91,666
DIME-C	Virtex-4 LX100 FPGA	51	1850	4	2162
Mitrion-C	Virtex-II Pro VP50 FPGA	100	822	2	2433
	Virtex-4 LX160 FPGA	100	823	4	4860

Table 5. Single-unit area utilization on FPGA.

Language	Processing Hardware	Resource	Used
VHDL	Virtex-4 XC4LX100	Slices	3270
		Block RAMS	33
DIME-C	Virtex-4 XC4LX100	Slices	10,391
		Block RAMS	51
Mitrion-C	Virtex-II Pro XC2VP50	Slices	11,782
		Block RAMS	23
	Virtex-4 XC4LX160	Slices	12,424
		Block RAMS	23

6.2 Mstack

Mstack is an algorithm involving a large number of independent sorts to calculate respective median values. The internal data types to be sorted are represented as floating point numbers. The hardware is designed to accommodate a sort array of maximum length 128, referred to as the number of channels in the Mstack benchmark. Due to the limited fixed size of the arrays, a bubble sort is used to perform the sorting for Mstack. This section describes DIME-C implementation of Mstack. Further details of the benchmark will be described in a separate document.

6.2.1 Mstack Implementation

The core computation of Mstack involves a bubble sort algorithm. The steps for the Mstack algorithm can be divided into initialization, sorting, and median calculation. In reconfigurable computing, previous tasks are decomposed and assigned to either a processor or an FPGA. The cost of using a coprocessor is the additional steps incurred for configuration and exchanging data with an FPGA. Tasks for the reconfigurable computing model for Mstack involve initial FPGA setup, data initialization on the host side, data transfer to the FPGA, sorting on the FPGA, median calculation on the FPGA, and transferring results back to the host memory.

The bubble sort and median calculation part of the Mstack benchmark were written in DIME-C and implemented in hardware to be executed on an FPGA. The remaining initialization steps are performed by the host. As expected, this experiment showed that to reduce the time spent on data transfer, a few large blocks of data should be transmitted, rather than many smaller blocks. Agglomeration like this is common in the field to reduce the start-up and shutdown costs for performing lots of small data transfers. However, large blocks of data require a large amount of FPGA memory, which, in turn, reduces the number of possible instantiations. To fit the maximum number of multiple-processing units, the use of FPGA resources was balanced between block RAMs and slices.

6.2.2 DIME-C Optimizations

The graphical representation of DIME-C code is a helpful tool that indicates serial, parallel, and pipelined executions through color-coded visualization. Additionally, a rough estimate of area utilization is calculated by the compiler. Using this information, a designer can quickly observe the effects of design modification without performing the lengthy synthesis step. By combining the rough estimate of area utilization calculated by the compiler, the following optimizations were applied to bubble sort algorithm written in DIME-C:

- Substitute multiplication and division with addition and shift operators.
- Compute loop-independent calculations outside of the inner loop.
- Use separate block RAMs to allow parallel execution.

Few options are available for creating multiple copies of a functional unit. The first option is to create multiple instances with the `#pragma` preprocessor command. This command allows separate instances of a function. The second method is to create multiple instances by using DIMEtalk software to duplicate necessary units using its graphical interface. The third option is to create multiple sort instructions within the DIME-C code. In the case of bubble sort, multiple instances of a bubble sort function can be generated using the `#pragma` command, but logic functions that could have been shared among separate instances are also duplicated. Area utilization can be saved if parallel bubble sorts can be combined to share similar logic requirements such as address calculation and address offset. Thus, multiple instances were added within a DIME-C code and duplicated in the DIMEtalk network.

6.2.3 Synthesis Setback

Xilinx recommends 4 GB of system memory and a 64-bit architecture for the synthesis of Xilinx-4 LX100 FPGA (*13*). For small designs, the synthesis process completes without any error, but designs utilizing full FPGA area abort during the synthesis step. To circumvent the memory shortage, files were transferred to a 64-bit Linux machine containing 4 GB of memory. The synthesis process can be invoked by typing “tclsh buld.tcl”. However, before running the synthesis script, file names under the “tmpcore” directory must be all lowercase letters for the synthesis process to recognize and load these files successfully on Linux operating systems.

6.2.4 Mstack FPGA Results

Performance results were obtained using the “time” command, which measures the total run time to execute a command in a Linux operating system. This elapsed time includes FPGA initial setup, FPGA configuration, host-to-FPGA communication, FPGA computation, and reading back of median results. A Virtex-4 FPGA (device type XC4VLX100, speed grade -10) was the specific hardware used for evaluation. Performance results are given in table 6. Device utilization is shown in table 7.

Table 6. Performance results on a Virtex-4 XC4VLX100 FPGA.

No. of Channels	Elapsed Time (s)	
	FPGA (Virtex-4 LX100)	CPU (3.0-GHz Intel Woodcrest)
5	12.5	1.3
50	62.2	9.5
75	119.4	18.0
128	298.1	44.0

Table 7. Virtex-4 LX100 area utilization.

Resource	Used	Available	Percentage
DSP48s	96	96	100
RAM16s	227	240	94
Slices	41,320	49,152	84

7. Closing Thoughts on Hardware Acceleration

In general, FPGA performance improvements are gained through unit duplication and hardware optimizations. First, replicating a processing element increases the throughput. By creating parallel processing units within an FPGA, multiple executions can occur concurrently. Hence area utilization becomes important, since it determines the maximum number of parallel units that can reside in an FPGA. For example, consider an algorithm running on a standard CPU clocked at 4 GHz. Now compare this to an FPGA that, since it employs SRAM technology, is currently clocked at 400 MHz, at best. Assuming that the CPU takes about one cycle, on average, to complete some operation, 10 duplicate-functioning units on the FPGA would roughly equal the processing power of the faster CPU. Second, hardware can be optimized for a task of interest. Optimizations include various hardware techniques to reduce the steps and time required to complete an algorithm. For example, bit manipulation, pipelining, and multiple-instruction execution are just a few of the optimization methods available in hardware design. In hardware, data path, data flow, storage, and control can be custom designed to be optimal for a particular algorithm. Basically, custom design dedicated for a specific purpose gives an FPGA the advantage. Therefore, flexibility or the option to control and customize hardware components is the key aspect in achieving hardware acceleration.

8. Conclusions

After overcoming the initial learning phase, high-level languages for hardware designs reduce design time to prototype and allow faster design modifications. The integration of host and FPGA interfaces significantly simplifies the interface development. However, compared to a

hand-coded hardware design, only a limited amount of hardware optimizations can be applied to the high-level languages because designers are exempt from the underlying hardware details. FPGA elements are not visible to the programmer (*14*). As a result, performance improvements depend on the amount of vendor-specific optimizations recognized within an algorithm.

C-to-hardware languages intend to simplify the programming of FPGAs for end-users by automatically translating a C-like source code into a hardware description language. In essence, a high-level language compiler is a translator rather than a hardware design tool. By making the low-level details transparent to users, the ability to custom design hardware that is tailored for a particular application is taken away. Not all the optimization options available to a hardware designer are accessible to a high-level language user. As a result, in the majority of cases, simply translating scientific code into hardware will not result in faster performance for several reasons, such as the following:

- The large area overhead associated with automated translation.
- A much lower clock frequency in the FPGA.
- An inability to customize hardware.

Despite the growing potential of FPGA-based systems and coprocessors, their popularity in the computing community remains stagnant. This may be for several reasons. The architecture and interface cores, along with corresponding API functions, differ with each company. The lack of a universal standard, platform incompatibility, and vendor-specific development tools also seems to thwart users from considering reconfigurable computing. These limitations define the challenges for the advancement in FPGA technology.

9. References

1. Top 500 Supercomputer Sites. http://www.top500.org (accessed September 2007).
2. Nallatech Limited. *H101-PCIX Reference Guide*; Issue 1; November 2006.
3. Cray Inc. *Cray XD1 FPGA Development*; Version 1.3.1; October 2005.
4. Osburn, J.; Anderson, W.; Rosenberg, R.; Lanzagorta, M. Early Experiences on the NRL Cray XD1. *HPCMP Users Group Conference*, Denver, CO, 26–29 June 2006; IEEE Computer Society: Washington, DC, 2006; pp 347–353.
5. Nallatech Limited. *DIME-C User Guide*; Issue 1; November 2005.
6. Nallatech Limited. *DIMEtalk 3.1 User Guide*; Issue 3; November 2006.
7. HPC Wire. http://www.hpcwire.com/hpc/491730.html (accessed August 2007).
8. Mohl, S. *The Mitrion-C Programming Language*; Revision 1.2.0–001; 2006.
9. Schneier, B. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), Fast Software Encryption. *Cambridge Security Workshop Proceedings*, Springer-Verlag: New York, 1994; pp 191–204.
10. Schneier, B. http://www.schneier.com/code/vectors.txt (accessed February 2007).
11. Genest, G.; Chamberlain, R.; Bruce, R. Programming an FPGA-based Super Computer Using a C-toVHDL Compiler: DIME-C. *Second NASA/ESA Conference on Adaptive Hardware and Systems*, Edinburgh, United Kingdom, 5–8 June 2007; IEEE Computer Society: Washington, DC, 2007; pp 280–286.
12. Mitrion. *Running the Mitrion Virtual Processor on XD1*, Ver. 1.2-001.
13. Xilinx. http://www.xilinx.com/ise/products/memory.htm (accessed July 2007).
14. Kindratenko, V.; Steffen, C.; Brunner, R. Accelerating Scientific Applications with Reconfigurable Computing. *Computing in Science & Engineering* **2007**, *9*, 70–77.

NO. OF COPIES	ORGANIZATION
1 (PDF ONLY)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FORT BELVOIR VA 22060-6218
1	US ARMY RSRCH DEV & ENGRG CMD SYSTEMS OF SYSTEMS INTEGRATION AMSRD SS T 6000 6TH ST STE 100 FORT BELVOIR VA 22060-5608
1	DIRECTOR US ARMY RESEARCH LAB IMNE ALC IMS 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRD ARL CI OK TL 2800 POWDER MILL RD ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

1	DIR USARL AMSRD ARL CI OK TP (BLDG 4600)

INTENTIOINALLY LEFT BLANK.