COMPARATIVE ANALYSIS OF HIGH LEVEL PROGRAMMING FOR RECONFIGURABLE COMPUTERS: METHODOLOGY AND EMPIRICAL STUDY

Esam El-Araby[1], Mohamed Taher[1], Mohamed Abouellail[1], Tarek El-Ghazawi[1], and Gregory B. Newby[2]
[1]The George Washington Univeristy, [2]Arctic Region Supercomputing Center
{esam, mtaher, mlail, tarek}@gwu.edu, newby@arsc.edu

ABSTRACT

Most application developers are willing to give up some performance and chip utilization in exchange of productivity. High-level tools for developing reconfigurable computing applications trade performance with ease-of-use. However, it is hard to know in a general sense how much performance and utilization one is giving up and how much ease-of-use he/she is gaining. More importantly, given the lack of standards and the uncertainty generated by sales literature, it is very hard to know the real differences that exist among different high-level programming paradigms. In order to do so, one needs a formal methodology and/or a framework that uses a common set of metrics and common experiments over a number of representative tools. In this work, we consider three representative high-level tools, Impulse-C, Mitrion-C, and DSPLogic in the Cray XD1 environment. These tools were selected to represent imperative programming, functional programming and graphical programming, and thereby demonstrate the applicability of our methodology. It will be shown that in spite of the disparity in concepts behind those tools, our methodology will be able to formally uncover the basic differences among them and analytically assess their comparative performance, utilization, and ease-of-use.

1. INTRODUCTION

Developing applications for Reconfigurable Computers (RCs) requires both hardware and software programming knowledge. The unique problems of RCs come from the fact that hardware and software are traditionally described using different languages and tools. The standard way of describing software is using high-level languages (HLLs), such as C, C++, or Fortran. The standard way of describing hardware is using hardware description languages (HDLs), such as VHDL and Verilog. Describing hardware using HLLs is possible, and has been tried in several commercial products such as Xilinx Forge, Celoxica Handel-C, Impulse-C, and Mitrion-C. Dataflow program entry, based on the graphical user interface, e.g. DSPLogic, seems to offer an interesting compromise between HLLs and HDLs. These languages offer a trade-off between a shorter development time and a performance overhead imposed by high level languages.

Describing hardware in HLLs, or at least using dataflow diagrams, seems to be a major and distinctive feature of high-performance RCs. It would allow mathematicians and computer scientists to develop entire applications without relying on hardware designers. It would also substantially increase the productivity of the design process.

A compiler for RCs must combine the capabilities of tools for traditional microprocessor compilation and tools for computer-aided design with FPGAs. It must also extend these two separate set of tools with capabilities for mutual synchronization and data transfer between microprocessor and reconfigurable processor sub-systems [1].

In this study, three high level tools for reconfigurable programming were evaluated and compared with respect to their performance and ease-of-use. Impulse-C, Mitrion-C and DSPLogic were chosen for two reasons. Firstly, they are all fully developed HLL tools sharing the common goal of reaching a broader audience of potential RC users. Secondly, each one has a different and distinct vision on how to realize the latter goal, whether through imperative, functional or schematic programming.

To be able to reach a well-rounded comparison between the tools, four work loads were selected for development on each tool. Furthermore, the same RC was used as the testbed. In this study, the Cray XD1 reconfigurable computer was the environment selected.

2. HIGH-LEVEL DESIGN TOOLS

Hardware description languages (HDLs) are tailored specifically to hardware design. Because of this they provide a flexible and powerful way to generate efficient logic. However, this tailoring makes them unfamiliar territory for people outside the hardware design field. In order to communicate hardware design to a more general audience, a number of tools are emerging to support the use of other high-level programming languages (primarily C and C++) as HDLs. The outcome of C and C++ for hardware design facilitates the partitioning of resources between software and hardware, and facilitates hardware and software co-simulation and code reuse.

2.1. Impulse-C

Impulse-C represents a class of imperative languages with syntax based strongly on ANSI C [3]. The language is extended to address specific hardware concepts such as communicating sequential processes (CSP), and streams. CSPs exploit the parallelism inherent in applications while streams provide a mechanism for inter-process communications. The Impulse-C software-to-hardware compiler translates C-language processes to low-level FPGA hardware. Existing VHDL designs may also be incorporated and called from the Impulse-C code as external functions. Fig. 1 shows the Impulse-C development flow.

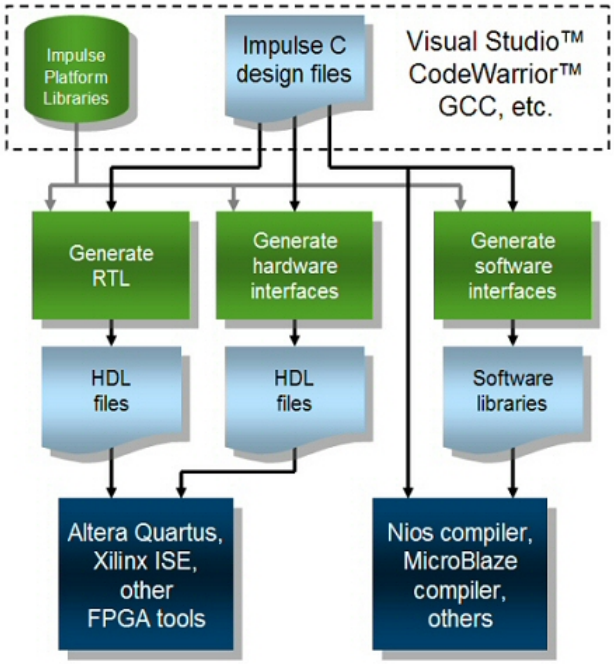

Fig. 1. Impulse-C to RTL to FPGA flow [3].

2.2. Mitrion-C

Mitrion-C is an ANSI C-based functional language [4]. Mitrion-C programming language is an implicitly parallel programming language with syntax similar to C. The language centers on parallelism and data-dependencies. In contrast, traditional languages are sequential and center on order-of-execution. In Mitrion-C there is no order-of-execution; any operation may be executed as soon as its data-dependencies are fulfilled. Mitrion-C is a Single-Assignment language (variables may only be assigned once in a scope) in order to prevent variables from having different values within the same scope. Software written in the Mitrion-C programming language is compiled into a configuration of the so-called Mitrion Virtual Processor. The Mitrion Virtual Processor is a fine-grain, massively parallel, reconfigurable soft-core processor. Fig. 2 shows the Mitrion-C programming flow.

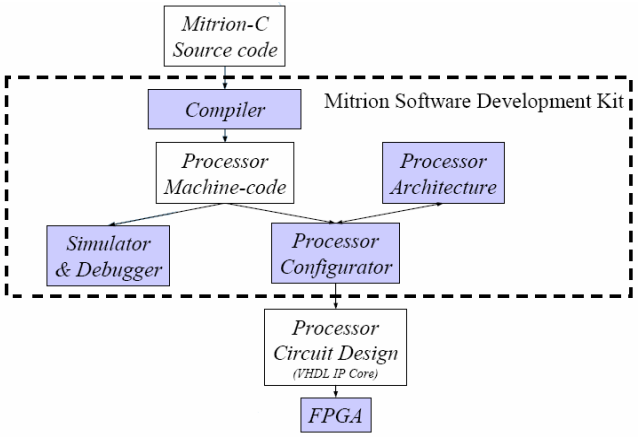

Fig. 2. Mitrion-C Programming flow [4].

2.3. DSPLogic RC Toolbox

DSPLogic RC Toolbox provides a combined graphical and text-based programming environment for RC application development based on Xilinx System Generator for DSP package [5]. It enables the designer to design RC applications with the MATLAB/Simulink package from The MathWorks. Blocks from the DSPlogic RC blockset and Xilinx System Generator are used to create a data flow diagram. Existing VHDL designs may also be incorporated using System Generator's HDL co-simulation capabilities. Fig. 3 shows the DSPLogic Programming Flow.

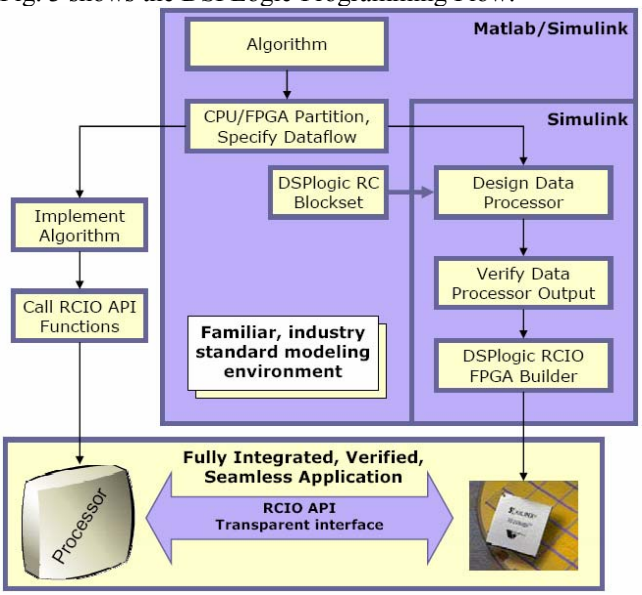

Fig. 3. DSPLogic Programming flow [5].

3. CRAY-XD1 RECONFIGURABLE COMPUTER

The general structure of the XD1 we used is as follows: one chassis houses six compute cards. Each compute card

has two AMD Opteron microprocessors at 2.4 GHz and one or two RapidArray Processors (RAPs) that handle the communication. The two Opteron microprocessors on each card are connected via AMD's HyperTransport with a bandwidth of 3.2 GB/s forming a 2-way SMP. Optionally an application acceleration processor (FPGA) can be put onto a compute board. With two RAPs/board a bandwidth of 8 GB/s (4 GB/s bi-directional) between boards is available via a RapidArray switch. This switch has 48 links of which half is used to connect to the RAPs on the compute boards within the chassis and the others can be used to connect to other chassis [6].

Users can develop their applications using either the standard HDL flow or a suite of higher-level languages such as C and C++ or the Xilinx System Generator for DSP package. Fig. 4 shows how a higher-level flow fits into the standard development flow of the Cray XD1 system [7].

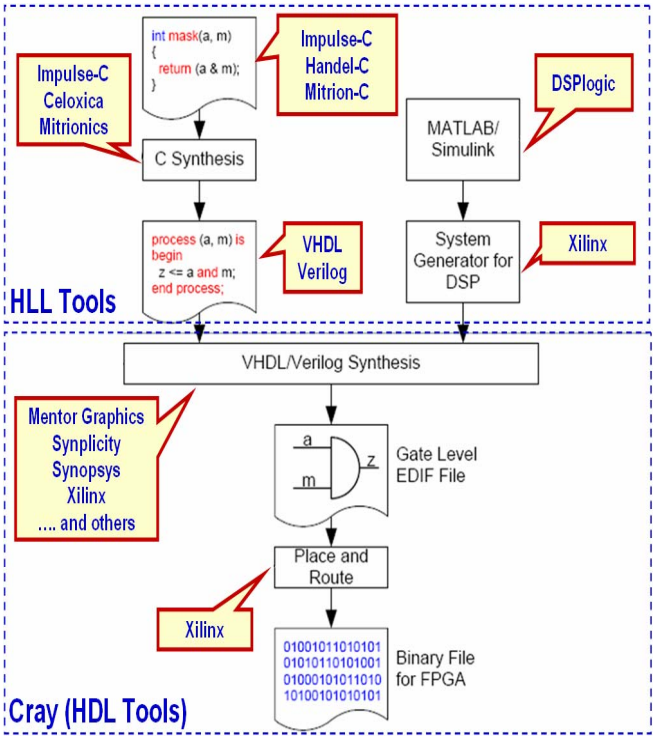

Fig. 4. Cray-XD1 development flow [7].

4. APPLICATIONS

Four workloads were selected for implementation on Cray-XD1 using the selected tools. The first workload is a simple pass-through implementation that reads input from the µP and sends it back. The purpose of this simple application is to measure the overhead caused by each tool on the FPGA with respect to the area utilization and also to measure the maximum clocking rates reached by each tool in the simplest of applications. This will give an initial and basic idea of the performance for each tool.

The second application implemented is a discrete wavelet transform (DWT). DWT is composed of two FIR filters and two down-samplers as shown in Fig. 5. The two filters are preloaded with the high-pass and low-pass coefficients defining the particular wavelet used for the transform.

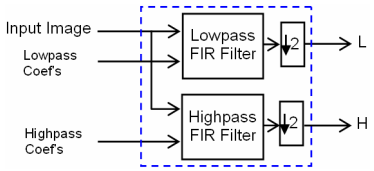

Fig. 5. 1-D DWT filter.

The third and fourth applications implemented are the data encryption standard algorithm (DES) and DES breaking. DES takes a 64-bit plaintext block (data) and a 64-bit key as inputs and generates a 64-bit ciphertext block (encrypted data). As shown in Fig. 6, DES consists of 16 identical rounds supplemented by a few auxiliary transformations. The DES breaking architecture is essentially similar to DES with constant plain and cipher texts. Because of this, the major differentiating

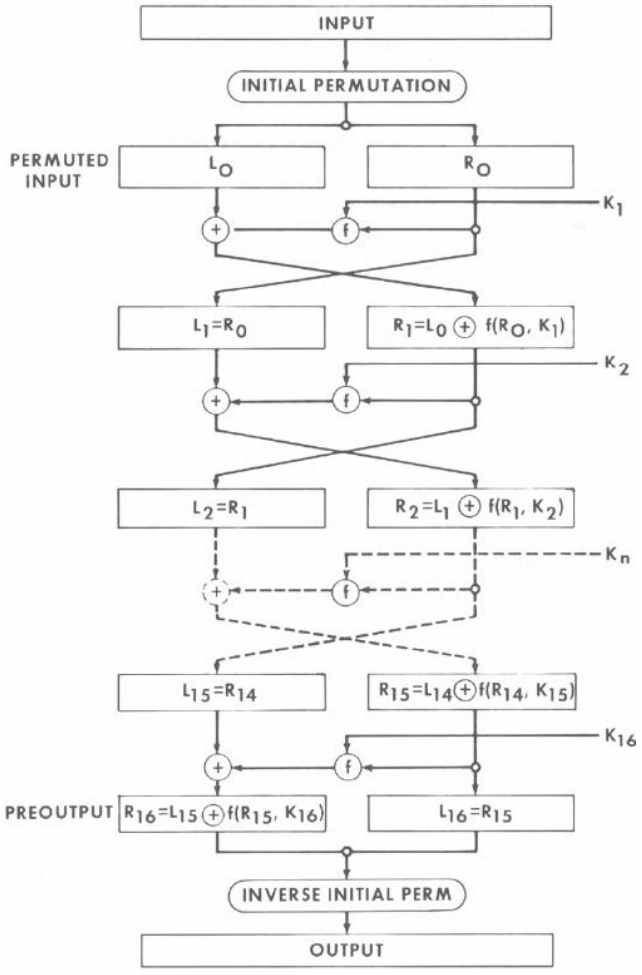

Fig. 6. DES algorithm architecture.

characteristic between DES and DES breaking, within our context, is that DES breaking is computational-intensive while DES is I/O-intensive application.

5. HLL PROGRAMMING PARADIGMS AND METRICS OF EVALUATION

The different HLL paradigms/approaches (i.e. imperative programming represented by Impulse-C, functional programming represented by Mitrion-C, and schematic/graphical programming represented by DSPLogic) were assessed in our study according to some envisioned evaluation metrics in terms of the explicitness of the programming model, ease-of-use, and efficiency of generating hardware as compared to a reference HDL approach.

The programming model can be defined as the hardware abstract view presented to the programmer by the programming tool. Thus, a programming model defines which parts of the hardware architecture will become visible to the programmer and under his/her direct control. In an RC, given a particular programming paradigm, the programming model determines whether (and how) the programmer can control data transfers between the FPGA and the onboard memory, the FPGA and the microprocessor memory, and the FPGA and the microprocessor.

Fig. 7 shows the Cray-XD1 operational environment, which illustrates all architectural modules that can be visible to the programmer and data transfers that can be under his/her control. Naturally, there is a trade off between how explicit the programming model is in making more architectural details visible and ease-of-use. The Cray-XD1 architecture allows the opteron processor to access the FPGA internal registers, internal memory, and external memory. The FPGA can access the µP memory. However, the use of HLL can disable some of these features.

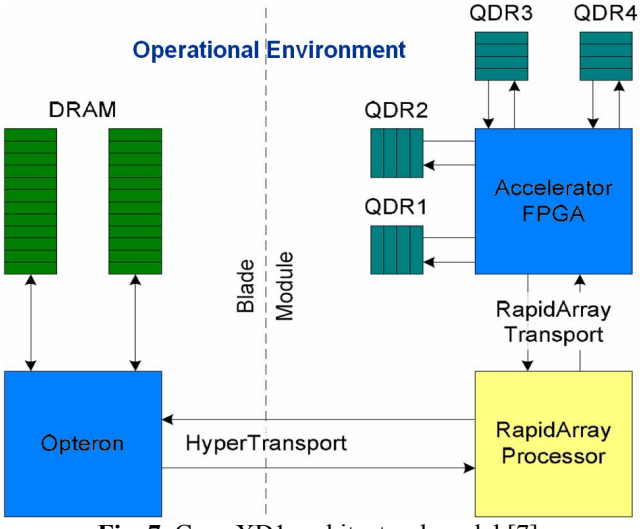

Fig. 7. Cray-XD1 architectural model [7].

The XD1 platform provides a number of transfer modes between the microprocessor and the FPGA depending on the initiator of the transfer. The microprocessor can read from and/or write to the FPGA local memory space (i.e. internal registers, internal BRAMS, and external memory). On the other hand, the FPGA can also read from and/or write to the microprocessor local memory space. The most efficient among these modes of transfer, in terms of bandwidth utilization and/or transfer throughput, is the so-called write-only mode. In this mode, the producer of the data initiates the transfer. These transfers can be either burst or non-burst. Burst transfers are useful for large amount of data.

The modes of transfer are embedded in the HLL tools with different degrees of explicitness/implicitness. Some of these tools hide from the user the details of implementation of such transfer while others leave it as the responsibility of the user. For example, DSPlogic implicitly handles the transfer scenarios utilizing the best mode (i.e. write-only architecture) guaranteeing the highest throughput possible. Table 1 shows the allowed features and modes of transfers utilized by the selected HLL tools on XD1.

The metric we devised for ease-of-use was in terms of the total acquisition (learning and experience gaining) time as well as the total development time. The easier the language is to use the faster it will take a user/developer to acquire/learn this language and develop a certain application with. In other words, the acquisition time and the development time express directly an opposite effect to ease-of-use; that is difficulty-of-use. The acquisition time is dependent on the type of paradigm being adopted, while the development time depends on both the paradigm as well as the application being developed. The acquisition time and the development time also capture the effects of the programming model explicitness. The more explicit the programming model is, i.e. the more architectural details that need to be handled by the user/developer, the longer it will take to both acquire the language and develop applications. It is also worth to mention that the user/developer experience can reduce both the acquisition and the development times. Therefore, our metric for ease-of-use, or equivalently difficulty-of-use, takes into consideration the programming model explicitness as well as the user/developer previous experience. A final note to mention with regard to this metric is that we will normalize this metric to a similar corresponding metric for a reference HDL language.

Another metric we used in our assessment is the efficiency of generating hardware. Efficiency combines the ability of each programming paradigm to extract the maximum possible parallelism/performance with the lowest cost. This is in terms of end-to-end throughput and the synthesized clock frequency, and resource usage (e.g. slice utilization) as compared to a reference conventional HDL approach; assuming optimality of the HDL approach.

Again, the user/developer experience should be taken into consideration when devising this metric. The more experienced the user/developer is the higher the throughput and frequency with less resource usage he/she can achieve from a certain language.

Finally, the sample size of the experiment population should be as large as possible to achieve accurate results and conclusions with minimum variances. By the sample size we mean the number of users being included in the experiment, the number of applications considered, the number of languages for each paradigm, as well as the number of platforms being used as testbeds. Our experiments involved three independent users with different degrees of experience in the field. As mentioned earlier we considered four different applications. Limited by the current status of the technology, we studied one language per paradigm on one supporting platform. This limitation of the sample size was beyond our control and was heavily dependent on the availability of all the languages on common platforms.

Based on the above discussion, we may introduce the following notation in order to quantify our concepts:

- Ease-of-Use
  - $N_{applications}$ is the total number of applications developed, i.e. the applications sample size
  - N_{users} is the total number of independent users involved in the experiments, i.e. the users/developers sample size
  - $T_{user.experience}$ is the individual user/developer experience in time units
  - $T_{experience}$ is the average experience of independent users in time units

$$T_{\exp erience} = \frac{\sum_{y=1}^{N_{users}} T_{user.\exp erience}}{N_{users}} \quad (1)$$

  - $T_{x.acquisition}$ is the average acquisition time of language x by independent users
  - $T_{ref.acquisition}$ is the average acquisition time of a reference language, e.g. VHDL, by independent users
  - $T_{y,x.deveolopment}$ is the average development time of application y using language x by independent users
  - $T_{y,ref.deveolopment}$ is the average development time of application y using a reference language, e.g. VHDL, by independent users
  - $T_{x.deveolopment}$ is the average development time using language x by independent users

$$T_{x.development} = \frac{\sum_{y=1}^{N_{applications}} T_{y,x.development}}{N_{applications}} \quad (2)$$

  - $T_{ref.deveolopment}$ is the average development time using a reference language, e.g. VHDL, by independent users

$$T_{ref.development} = \frac{\sum_{y=1}^{N_{applications}} T_{y,ref.development}}{N_{applications}} \quad (3)$$

  - d_x is the average difficulty factor of language x across independent users

$$d_x = \frac{T_{x.acquisition} + T_{x.development}}{T_{\exp erience}} \quad (4)$$

  - d_{ref} is the average difficulty factor of a reference language, e.g. VHDL, across independent users

$$d_{ref} = \frac{T_{ref.acquisition} + T_{ref.development}}{T_{\exp erience}} \quad (5)$$

  - δ_x is the normalized average difficulty factor of language x relative to a reference language, e.g. VHDL

$$\delta_x = \frac{d_x}{d_{ref}} = \frac{T_{x.acquisition} + T_{x.development}}{T_{ref.acquisition} + T_{ref.development}} \quad (6)$$

  - λ_x is the normalized average ease-of-use factor for language x relative to a reference language

$$\lambda_x = 1 - \delta_x = 1 - \frac{T_{x.acquisition} + T_{x.development}}{T_{ref.acquisition} + T_{ref.development}}$$

$$\lambda_x = \frac{\left(T_{ref.acquisition} - T_{x.acquisition}\right) + \left(T_{ref.development} - T_{x.development}\right)}{T_{ref.acquisition} + T_{ref.development}} \quad (7)$$

$$\lambda_x = \frac{\Delta_{acquisition} + \Delta_{development}}{T_{ref.acquisition} + T_{ref.development}}$$

- Efficiency
  - $f_{y,x}$ is the synthesized frequency achieved for application y using language x across the independent users
  - $P_{y,x}$ is the average throughput achieved for application y using language x across the independent users
  - $P_{y,ref}$ is the average throughput achieved for application y using a reference language, e.g. VHDL, across the independent users
  - $A_{y,x}$ is the average resource usage, i.e. area utilization, achieved for application y using language x across the independent users
  - $A_{y,ref}$ is the average resource usage, i.e. area utilization, achieved for application y using a reference language, e.g. VHDL, across the independent users
  - $E_{y,x}$ is the average efficiency for application y using language x across the independent users

$$E_{y,x} = \frac{f_{y,x} \cdot P_{y,x}}{A_{y,x}} \cdot T_{\exp erience} \quad (8)$$

  - $E_{y,ref}$ is the average efficiency for application y using a reference language, e.g. VHDL, across the independent users

$$E_{y,ref} = \frac{f_{y,ref} \cdot P_{y,ref}}{A_{y,ref}} \cdot T_{experience} \quad (9)$$

- E_x is the average efficiency of language x

$$E_x = \frac{\sum_{y=1}^{N_{applications}} E_{y,x}}{N_{applications}} = \frac{T_{experience}}{N_{applications}} \cdot \sum_{y=1}^{N_{applications}} \frac{f_{y,x} \cdot P_{y,x}}{A_{y,x}} \quad (10)$$

- E_{ref} is the average efficiency of a reference language, e.g. VHDL

$$E_{ref} = \frac{\sum_{y=1}^{N_{applications}} E_{y,ref}}{N_{applications}} = \frac{T_{experience}}{N_{applications}} \cdot \sum_{y=1}^{N_{applications}} \frac{f_{y,ref} \cdot P_{y,ref}}{A_{y,ref}} \quad (11)$$

- η_x is the normalized average efficiency of language x relative to a reference language, e.g. VHDL

$$\eta_x = \frac{E_x}{E_{ref}} = \frac{\sum_{y=1}^{N_{applications}} \frac{f_{y,x} \cdot P_{y,x}}{A_{y,x}}}{\sum_{y=1}^{N_{applications}} \frac{f_{y,ref} \cdot P_{y,ref}}{A_{y,ref}}} \quad (12)$$

We would like to mention that the ease-of-use metric has been mapped to the range from zero to unity, or equivalently 100%, inclusive, through the normalized average ease-of-use factor, λ_x. Unity λ_x represents the easiest-to-use languages while zero λ_x represents the most difficult-to-use languages, i.e. HDLs. Similarly, zero normalized efficiency, η_x, represents the least efficient languages while unity normalized efficiency represent the most efficient languages, i.e. HDLs.

6. EXPERIMENTAL RESULTS AND OBSERVATIONS

Table 1 shows the ease-of-use of the different paradigms, i.e. imperative, functional, and graphical, in terms of the acquisition time.

The development time of each application under each paradigm is included in Table 2. Table 2 also presents the experimental results collected for the four workloads implemented using the selected tools as well as a reference HDL implementation for each workload. It can be noticed that DSPlogic and Impulse-C, depending on the application design, can achieve the highest clock rate supported on XD1, 200MHz, while Mitrion-C is limited to a maximum of 100 MHz. On the other hand, Impulse-C could achieve the lowest resource usage for all workloads used. This is due to the fact that Impulse-C uses the simplest transfer scenario which does not utilize the FPGA external memory interface. This results in less resources being utilized as compared to the other tools. However, for I/O-intensive applications, this transfer scenario impacted the throughput dramatically, because it is the least efficient among all transfer modes on XD1 while for computational-intensive applications such as DES breaking the tool achieves comparable results to HDL. On the other hand, it can also be noted that the results for DES and DES breaking on Mitrion-C are not shown. DES, as an example of relatively large applications, proved to be problematic to the Mitrion-C compiler. This compiler showed limited capabilities in handling large designs even for simulation. We may observe from the experimental results that programming models and their explicitness to the end-user for the different approaches (i.e. imperative, functional, and graphical) is implementation dependent and differ from one vendor to another. This suggests a need for standardization with this respect. In addition, imperative approaches proved to be the easiest to acquire and use while performing reasonably and comparably with standard HDL approaches. Dataflow/graphical approaches proved to achieve the highest efficiency but not as easy to acquire and use as imperative counterparts. On the other hand, pure functional approaches proved to be the most difficult, among the three approaches, to acquire and use. Moreover, HDL approaches remain the highest in

Table 1. Explicitness of the programming models.

	µP - FPGA's External Memory	µP - FPGA's Internal Memory	µP – FPGA Internal Registers (Non-Burst Transfer)	FPGA – µP Memory	Acquisition Time[A] (days)
Impulse-C	Future Support	NA	Read / Write (Implicit)	Future Support	7
Mitrion-C	Read / Write (Burst/Non Burst) (Explicit)	NA	Read / Write (Explicit) (limited to 32 registers)	Burst Write-only (Explicit)	14
DSPlogic	Burst Write-only (Implicit)	Write-only (Implicit)	Write-only (Implicit) (limited to 8 registers)	Burst Write-only (Implicit)	7
VHDL	Explicit	Explicit	Explicit	Explicit	15

Note A
Equation (1) → $T_{experience}$= (10+3+2)/3 = 5 years

Table 2. Experimental Results.

Application	Metric	Impulse-C	Mitrion-C	DSPLogic	VHDL
Pass-Through	Clock Rate (MHz.)	200	100	200	200
	Slice Utilization (%)	13	21	28	12
	Throughput (MB/s)	1.4	376	534	620
	Development Time (Hours)	<1	1	1	1
DWT	Clock Rate (MHz.)	100	100	200	200
	Slice Utilization (%)	17	28	29	13
	Throughput (MB/s)	1.35	375	481	620
	Development Time (Hours)	4	6	5	10
DES	Clock Rate (MHz.)	100	NA	200	200
	Slice Utilization (%)	35	NA	37	22
	Throughput (MB/s)	1.38	NA	481	620
	Development Time (Hours)	10	18	15	25
DES Breaker	Clock Rate (MHz.)	100	NA	200	200
	Slice Utilization (%)	36	NA	37	23
	Throughput (MB/s)	800	NA	1600	1600
	Development Time (Hours)	10	18	15	25

Table 3. Tools Efficiency and Ease-of-Use.

	Average Difficulty factor (d_x)	Normalized Average Difficulty factor (δ_x)	Normalized Average Ease-of-Use factor (λ_x) (%)	Average Efficiency (E_x)	Normalized Average Efficiency (η_x) (%)
Impulse-C	3.9727×10^{-3}	0.4640	53.60	2.8196×10^{3}	5.72
Mitrion-C	7.9112×10^{-3}	0.9240	7.60	3.9122×10^{3}	7.94
DSPlogic	4.0383×10^{-3}	0.4717	52.83	2.2975×10^{4}	46.625
VHDL	8.5615×10^{-3}	1	0	4.9277×10^{4}	100

$T_{experience}$ = 5 years

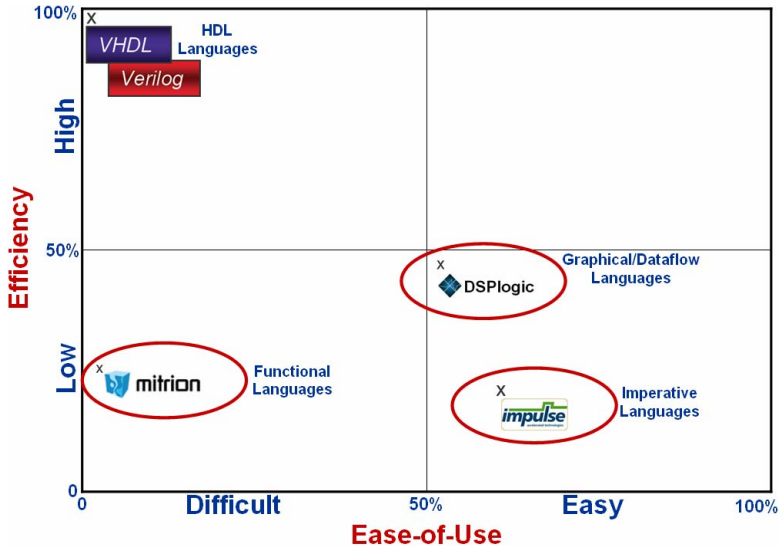

Fig. 8. Efficiency vs. Ease-of-Use of Programming Paradigms.

efficiency (optimal with this respect) but on the expense of being the hardest to acquire and use for applications developers/programmers, specifically for those with limited hardware design experience. These observations are captured both in Table 3 and in Fig. 8.

7. CONCLUSIONS

In this study, three high level tools, i.e. Impulse-C, Mitrion-C and DSPLogic, for reconfigurable programming were formally and quantitatively evaluated and compared with respect to their performance in the Cray XD1 environment. These tools were selected to represent imperative programming, functional programming and graphical/dataflow programming. We established a formal methodology and framework for evaluating different programming paradigms for reconfigurable applications and platforms. The metrics we devised were ease-of-use, and efficiency of generating hardware characterized by high throughput at the lowest cost of resource usage. The user previous experience was also taken into consideration within this framework. It was shown that in spite of the disparity in concepts behind those programming paradigms, our methodology was able to formally uncover the basic differences among them and analytically assess their comparative performance, utilization, and ease-of-use.

To be able to reach a well-rounded comparison between the tools, four work loads were selected for development on each tool. Furthermore, the same reconfigurable computer was used as the testbed. In this study, the Cray XD1 reconfigurable computer was the environment selected.

After thorough examination of each tool, one comes up with many lessons learned. First off, the designer must keep in mind that optimized hardware C isn't the same as optimizing software C. Secondly; some hardware knowledge is still needed for a better understanding of the tool when performance issues arise.

What's very encouraging with new suites of HLLs for hardware description emerging is that preliminary results achieved are close to manual HDL. Software-to-hardware porting becomes considerably easier and more efficient with every new release. These tools will definitely benefit the scientific community in developing complex and faster running designs with having significantly less knowledge of HDL.

Of course many challenges still remain. Unsupported platforms will require VHDL knowledge, however, once a wrapper is generated, it will be reusable. Another major problem is Hardware debugging. Tracing becomes difficult since the internal VHDL signals are unknown.

8. REFERENCES

[1] W. Luk, N. Shirazi, and P.Y.K. Cheung, Compilation Tools for Run-time Reconfigurable Designs, IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 1997; 56–65.

[2] K. Compton, S. Hauck, Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys 34 (2) (2002) 171-210.

[3] Impulse C – "Impulse Accelerated Technologies" web site available at http://www.impulsec.com/

[4] Mitrion web site available at http://www.mitrion.com/index.shtml

[5] DSPLogic web site available at http://www.dsplogic.com

[6] Cray website available at http://www.cray.com

[7] Cray Inc, Seattle WA, "Cray XD1 Datasheet", 2005

[8] Xilinx, Board Level Verification, http://www.xilinx.com/products/design_resources/design_tool/grouping/board_level_verif.htm

[9] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, Using Design-Level Scan to Improve Design Observability and Controllability for Functional Verification of FPGAs, FPL 2001.

[10] W. Stallings, Cryptography and Network Security, Prentice Hall, 1999.

[11] E. El-Araby, M. Taher, K. Gaj, T. El-Ghazawi, D. Caliga, and N. Alexandridis, System-Level Parallelism and Throughput Optimization in Designing Reconfigurable Computing Applications, RAW 2004.

[12] Van der Steen, Aad J. and Jack Dongarra, "Overview of Recent Supercomputers," 2004.