Performance of Data-Intensive Algorithms on FPGAs in an Object-oriented Programming Environment

Oskar Mencer, Martin Morf, Michael J. Flynn

Technical Report : CSL-TR-00-792

February 2000

This research is supported by DARPA Grant No. DABT63-96-C-0106.

Performance of Data-Intensive Algorithms on FPGAs in an Object-oriented Programming Environment

by

Oskar Mencer, Martin Morf, Michael J. Flynn

Technical Report : CSL-TR-00-792

February 2000

Computer Systems Laboratory

Department of Electrical Engineering and Computer Science

Stanford University

William Gates Computer Science Building, 4A-408

Stanford, California 94305-9040

Email: pubs@shasta.stanford.edu

Abstract

Recently, we see academic and industrial efforts to combine traditional computing environments with reconfigurable logic. Each application, or part of an application, has an optimal implementation within the design space of microprocessors, reconfigurable logic, and hardwired VLSI circuits. Programmability, Performance, and Power (P^3) are the major metrics that have to be taken into account when deciding between the available technologies. *Performance* advantages of FPGAs over processors for specific applications have been shown in previous research. We show the potential of current low-power FPGAs to outperform current state-of-the-art processors in *Performance over Power* by more than half an order of magnitude. *Programmability* remains a tough issue. As a starting point, we define a hardware object interface in C++, PAM-Blox. PAM-Blox is an open, object-oriented environment for programming FPGAs that encourages design sharing and code reuse. PAM-Blox simplifies the creation of optimized high-performance designs. Encouraging a distributed effort to share hardware objects over the internet in the spirit of open software, is a first step towards improving the programmability of FPGAs.

Key Words and Phrases: reconfigurable computing, digital signal processing, power, object-oriented

Contents

1 Introduction **1**
1.1 Object-Oriented Hardware Design 2

2 Programmability **3**
2.1 PAM-Story 3
2.2 PAM-Blox: PamBlox and PaModules 4
2.2.1 Hierarchical Naming 5

3 PAM-Blox: Performance **5**
3.1 Jacobi Relaxation 6
3.2 DES Encryption 6
3.3 Integer Matrix Multiply 6
3.4 Constant(K) Coefficient Multipliers - KCM 8
3.5 Compile Time and Xilinx Place-and-Route 8

4 Power **10**
4.1 Microprocessor versus FPGA versus ASIC 11

5 Conclusions **11**

List of Figures

1	Active PAM-Blox methods	2
2	Hardware/Software interface	4
3	PamBlox Hierarchy	7
4	Constant(K) coefficient multipliers	9
5	IDEA encryption data-flow graph	12
6	Performance over Power	13
7	CORDIC arithmetic unit	14

List of Tables

1 Lines of Code 7
2 RAW Benchmarks 7
3 Performance and Power 12

1 Introduction

Adaptive computing has been an active area of research for more than a decade [16]. While it has been shown that for certain applications Field-Programmable Gate Arrays (FPGAs) can achieve an improvement in performance over existing microprocessors, competitive FPGA designs have been created mostly on a very low, structural level. The first FPGA based computing machines, the PAM [5] and the Splash-2 [9], were build shortly after the introduction of FPGAs by Xilinx in 1985. Both projects investigated the feasibility of FPGAs as computing platforms. Both groups found that there was a certain set of highly parallelizable and/or pipelinable algorithms for which FPGAs outperform microprocessor based computing systems. Recently, we see academic[13, 22] and industrial[24, 21] efforts to combine traditional computing environments with reconfigurable, FPGA-like, logic. In such a combined computing system, the programmer will have the choice between fixed function VLSI, the microprocessor, and reconfigurable logic. Each application, or part of an application, has an optimal implementation within the design space of the available technologies. In this paper we focus on Programmability, Performance, and Power (P^3) of FPGAs. Besides economic considerations, P^3 are the major metrics that have to be taken into account by the designer. By improving P^3 we improve the applicability of reconfigurable technology within a conventional computing system. While the improvement of performance and power can be measured directly, the improvement in programmability of high-performance FPGA designs lies within the object-oriented environment – especially the specific hardware object interface.

Currently, FPGAs are programmed with CAD tools that have been optimized for traditional VLSI design. Some attempts have been made towards creating programming languages for FPGAs [12, 11]. Currently, the drawback of direct compilation to FPGAs is that most of the performance, and significant area are lost in the process. Designs compiled from high-level, behavioral languages such as C rarely compete with current general-purpose microprocessor technology. In order to exploit the efficiency of hand design while simplifying the design process, we suggest a bottom-up approach (see also [14]) to compilation for FPGAs.

Conventional general purpose processors consist of a fixed, general data-path, and programmable control (instructions) for that data-path. On these processors, a few bit-parallel arithmetic units are highly optimized for low latency[1, 23]. On FPGAs data-path and control are fully programmable, allowing the designer to tailor the architecture of the computer to the structure of the algorithm. Flexibility, or reconfigurability help to achieve high throughput, but this high throughput comes at the expense of latency (i.e. longer cycle time) and logic density on the chip.

Power consumption of FPGAs is basically dependent on the specific circuit and clock frequency. Given the same circuit, FPGAs always use more power than the equivalent custom VLSI implementation. On the other hand, a microprocessor also does not always make efficient use of all it's power consuming resources.

In the following sections we show that FPGAs can outperform microprocessors for specific applications in terms of P^2, justifying the additional effort in programming. Before considering the issues of P^3, we examine object-oriented hardware design in general, leading

```
class HWobject:public parent{
public:
  <internal wire declarations>

// constructor
  HWobject(input parameters, optional){
    <initialization of inputs>
  }

  out(output parameters, optional){
    <internal logic>
  }

  <additional methods called by 'out'>

  place(absolute placement parameters){
    <absolute placement>
  }
  place(){
    <relative placement>
  }
}
```

Figure 1: Active methods of a general PAM-Blox hardware object described in C++.

to the motivations for our approach to improve programmability: PAM-Blox.

1.1 Object-Oriented Hardware Design

In object-oriented software the structure of the data defines the structure of the program. From this perspective it seems to be a natural fit to use the object-oriented paradigm for structural description of VLSI circuits.

In an object-oriented sequential language the description, and instantiation of the hardware object has to be efficient in terms of code-size in order to minimize the complexity of creating designs. Also the specification of the hardware object should provide a very explicit interface (API). In general, such an API should be scalable, intuitive, and extensible.

Figure 1 shows the structure of a hardware object described in C++. All data and methods inside a hardware object are declared public in order to allow maximal visibility. Inputs and outputs are syntactically separated by passing inputs to the constructor and outputs to the 'out' method.

It is convenient to think of all the internal wires of a hardware object as the major part of the state of the object. The wires can be declared as the variables of the object. All methods can access the wires and implement active or passive actions. Active methods are methods describing circuits or specifying placement constraints. Passive methods are used to query an object, returning information about the object.

Templates provide an elegant implementation for circuit generators. Instantiation of an object from a template results in automatic range checking. In order to write a template for a hardware object, the micro-architecture has to be simple enough to enable parameterizable circuit generation and placement.

Inheritance significantly reduces code-size through code-reuse. Inheritance is also the basis for scalability and extensibility. As chips get larger due to improvements in VLSI fabrication technology, designs get more complex. Being able to cleanly reuse code written by others, simplifies cooperation and teamwork.

2 Programmability

PAM-Blox consists of open, object-oriented circuit generators on top of the PCI Pamette design environment, PamDC. *PAM-Blox* were first introduced in [18]. *PAM-Blox* are intended to be part of an open repository that enables design sharing between members of the adaptive computing community.

High-performance FPGA design for adaptive computing is simplified by using a hierarchy of optimized hardware objects described in C++. Programmability is improved by using object-oriented techniques such as templates, virtual functions, function overloading, and inheritance, in the specific way outlined in this section. As with object-oriented software, the design of the hardware object interface is critical to the usefulness of the system. In fact, more effort was spend on the iterative design of the interface, than was later necessary to design the hardware objects themselves.

In order to bridge the space between the algorithmic representation of an application and the gate level (lookup table level) we add levels of abstraction, starting at the register transfer level (RTL) which is equivalent to the PamDC level. The "big picture" is shown in figure 2. The boundary between hardware and software for processors, FPGAs, and ASICs, defines the interface between programmer/compiler and the computing elements. The low-level boundary between software and hardware for FPGAs requires the software to bridge a large space from algorithm down to the gate level.

2.1 PAM-Story

PAM stands for Programmable Active Memories. The first PAM, PeRLe-0, developed at DEC PRL in France[5], is one of the first FPGA based computing machines. Next to the hardware efforts the PAM team also developed a C++ class library, PamDC, for creating designs for Xilinx FPGAs. The most impressive result obtained with PamDC and the PeRLe-1 board is RSA encryption at Cray speeds[26].

The most current PAM is the PCI Pamette board developed by Mark Shand[27] at DIGITALs Systems Research Center. The PCI Pamette consists of 5 Xilinx XC4000 series FPGAs.

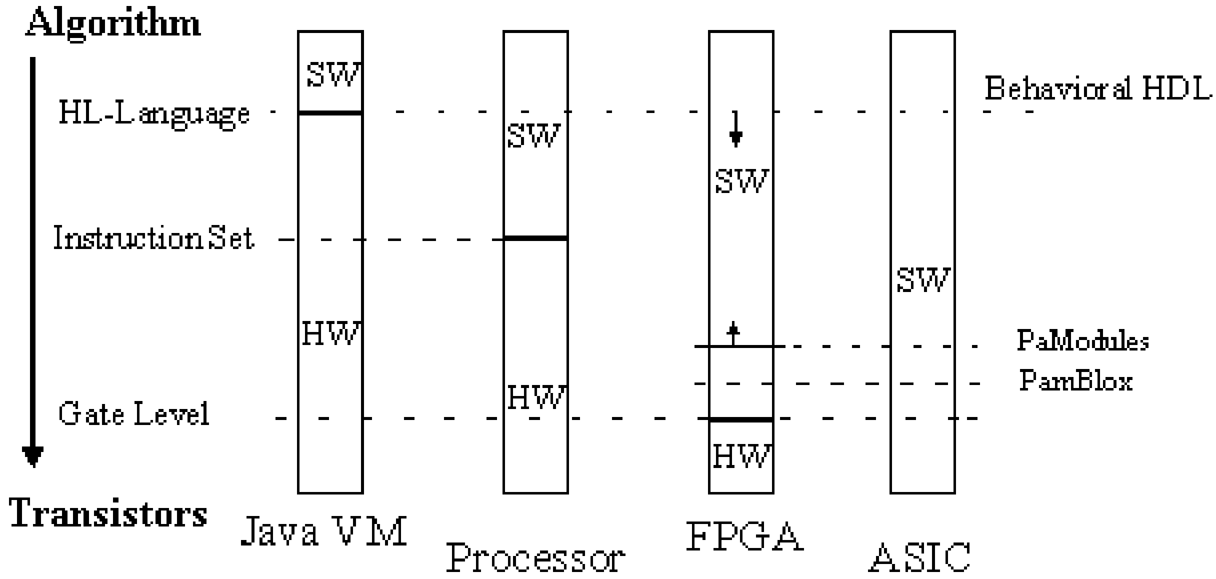

Figure 2: The figure shows the Hardware/Software interface for different technologies: a Java microprocessor(JavaVM), a conventional microprocessor, FPGAs, and ASICs. In general, the programmer has to bridge the gap between algorithms and transistors.

2.2 PAM-Blox: PamBlox and PaModules

PAM-Blox consist of two major layers of abstraction. First, PamBlox are parameterizable simple elements such as counters and adders. Automatic placement of carry chains and flexible shapes are supported. PaModules are more complex elements possibly instantiating PamBlox. PaModules generally have fixed shapes and are usually optimized for a specific data-width. Examples for PaModules are multipliers, Coordinate Rotations (CORDICs) [20], and special arithmetic units for encryption. A detailed example of a CORDIC PaModule can be found in Appendix A.

The key difference of our approach to most other design tools for FPGAs is that the designer has *optional* control over placement at each level of the design hierarchy, which is the key to high-performance FPGA design. As mentioned before, the object interface was chosen carefully to encourage code-reuse and simplify code-sharing between designers.

A subset of the PamBlox hierarchy is shown in Figure 3. The top object, *PBtop*, consists of a vector of registers, and a set of placement functions which handle different carry-chain configurations. As an example of code reuse, every child of PBtop inherits the placement functions and can overwrite them if necessary.

PaModules are more complex, generally fixed circuits implemented as C++ objects. PaModules can include multiple PamBlox and are optimized for a specific data-width. Examples are constant(K) coefficient multipliers (KCMs), Booth multipliers, Coordinate Rotation Digital Computer (CORDIC) circuits [20], and special purpose arithmetic units such as a constant multiply modulo $(2^{16}+1)$ operation for encryption(see also section 4 and [17]).

The table in figure 1 shows the code-size of PAM-Blox version 1.0 circuit generators. Code-size is given in PamDC / C++ lines necessary to implement the objects.

2.2.1 Hierarchical Naming

Commercial synthesis tools make it difficult to optimize circuits on a low level (e.g. after place-and-route) by flattening the design and changing the naming of the wires. PamDC enables direct control over the naming of wires. PAM-Blox are implemented to support a hierarchical naming scheme that creates a unique name for each wire in the design similar to paths in a file-system. The name of each wire contains all the ancestors (parent objects) of the wire. The top name can be specified by the designer, e.g. a PaModule multiplier with the name "multy" containing one adder with a carry-chain, results in the following name for the third element of the carry-chain, e.g.

$$multy/add0/carry < 3 > \tag{1}$$

The naming scheme enables designers to use additional tools for debugging and still be able to trace the source of each wire found in the final netlist. For example, the naming hierarchy is preserved for simulation (within PamDC) and low-level tools such as Xilinx fplan.

3 PAM-Blox: Performance

Applications that have been shown to execute favorably on FPGAs are data intensive applications which can be executed in very deep pipelines (e.g. encryption, pattern matching, etc.) and applications with a huge amount of fine grain parallelism such as Jacobi relaxation and lattice gas simulation[15].

Given today's technology, FPGA based computing machines can compete with general purpose processors on latency tolerant applications that require a relatively small amount of logic during a specific period of time, which we refer to as *persistence* of the associated task. *Persistence* has to be an order of magnitude larger than reconfiguration time. Due to large reconfiguration times of today's devices single FPGAs do not scale easily to large problem sizes or large data-flow graphs. Multiple FPGAs can be used to compute larger problems. The major drawback is the very high complexity of partitioning a design onto multiple FPGAs given a limited amount of pins. Overcoming the pin-limitation in software – with design tools – is investigated in the Virtual Wires[4] project. Eliminating the pin limitation with multi-chip modules of FPGAs is explored in the Teramac project[7].

First, we compare the original implementations of the RAW benchmarks[3] Jacobi, Matrix Multiply and DES encryption synthesized by Synopsys FPGA Express II with implementations using PAM-Blox. The PAM-Blox implementations of these RAW benchmarks have been designed by trying to keep the design effort within order of magnitude of the design effort for the behavioral implementations in Verilog. Note that PAM-Blox supports optional, hierarchical placement, while the design methodology of FPGA Express does not enable placement by the designer. Ideally, object-oriented hardware generators will be combined with object-oriented behavioral CAD tools.

Our metrics of comparison are *minimal cycle time*, *area requirement* in configurable logic blocks (CLBs), and *compile time* (from C++ / PAM-Blox to the Xilinx Netlist Format). As we will see later, area is directly proportional to power consumption, giving us an additional perspective to the data presented below. Compile time was measured on an Intel Pentium PC at 120 MHz. PAM-Blox are compiled with Microsoft Visual C++ 4.0. The objective of this section is to put real results from PAM-Blox into perspective with a state-of-the-art, commercial CAD tool – not to be mistaken for an argument for structural (PAM-Blox) versus behavioral (FPGA Express/Verilog) design – while acknowledging that it took more effort to create the PAM-Blox designs.

We show the performance of the following three RAW benchmarks.

3.1 Jacobi Relaxation

Jacobi relaxation is an iterative method for solving differential equations of the form:

$$\nabla^2 A + B = 0 \tag{2}$$

The basic operations for this benchmark are shift and add. The implementations compared in table 2 consist of a 4x4 array with 2x2 active cells and 8 bit values. During each clock cycle, each active cell takes the values of cells neighboring east, south, west and north, adds them together and divides the result by four. The arithmetic operations 'shift and add' map easily onto the Xilinx XC4000E library used by FPGA Express II. Therefore there is not much room for area improvement. Clock frequency of the PAM-Blox design is only about 15% higher than the design optimized by Synopsys HDL compiler. The improvement in area is about 20%.

3.2 DES Encryption

DES encryption is well suited for implementation in hardware. The basic primitives are fixed permutations and exclusive-or. The results for the PAM-Blox DES design in table 2 show a 30% increase in performance (clock frequency) using half of the area.

The superior results obtained with PAM-Blox are due to partially manual placement and technology mapping, i.e. the careful design of logic that fits into 4 bit lookup tables.

3.3 Integer Matrix Multiply

The Matrix Multiply benchmark multiplies two 4x4 matrices with $4^2 = 16$ multipliers and an adder tree. FPGA Express II uses simple bit-serial shift-and-add multiplication. A full matrix multiply therefore takes more than 50 clock cycles. For this benchmark we chose to create a more efficient computational structure, i.e. arithmetic unit to show how PAM-Blox can be used to adapt the arithmetic units to the specific requirements of the application. By implementing multiple bit-serial multipliers using Booth encoding, we are able to trade area

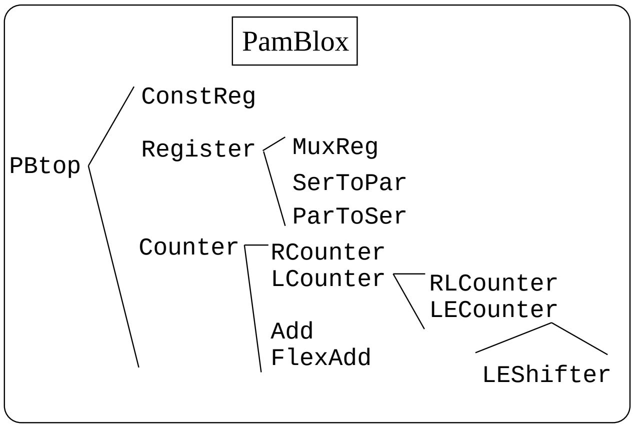

Figure 3: A subset of the PamBlox hierarchy. The top object PBtop consists of a minimal amount of logic and a set of placement functions which are inherited by all PamBlox objects. Prefix 'R' stands for "Resetable", 'L' for "Loadable", and 'E' for an "Enable".

Table 1: The table shows the initial size of the PAM-Blox generators, given in lines of PamDC / C++ code.

	PamBlox	PaModules
No. of Objects	28	6
Lines of Code	1370	750
Av. Lines per Object	~ 50	~ 120

Table 2: *R*AW benchmarks compiled with Synopsys FPGA Express II (FE II) are compared to PAM-Blox implementations. Compile time stands for the time to compile a design description to a Xilinx netlist file. FPGA Express results are reported for the completely placed and routed system. The performance for matrix multiply is given in matrix multiplications per second (mmps).

	Compile Time	Area [CLBs]	Frequency
JACOBI 4x4 (8 bit)			
FE II	80 s	164	30 MHz
PAM-Blox	45 s	129	35 MHz
DES (1)			
FE II	1,510 s	828	15 MHz
PAM-Blox	86 s	398	22 MHz
MATMULT 4x4 (8 bit)	Compile Time	Area [CLB]	Mega-mmps
FE II	350 s	609	0.38
PAM-Blox 1	77 s	604	1.23
PAM-Blox 2	98 s	954	1.52

for performance. Obviously Booth multipliers are more efficient for this specific application. The idea is to use the PAM-Blox environment to choose the arithmetic unit – in our case the multiplier – that is best suited for the specific application.

Table 2 shows two PAM-Blox designs, PAM-Blox 1 and PAM-Blox 2, differing only in the selection of the multiplier PAM-Blox 1 multiplies the matrices in 27 clock cycles while PAM-Blox 2 takes 19 clock cycles for a full 4x4 matrix multiplication including data transfer. Clock cycle times for the PAM-Blox designs are around 33 MHz. The original design synthesized with FPGA Express II runs at 15 MHz and requires 39 clock cycles for a full matrix multiply. The table above shows the throughput in matrix multiplications per second. With the right multiplier we see an increase in throughput of up to 4 times, compared to the original RAW benchmark compiled with Synopsys FPGA Express II, using a generic shift-and-add multiplier. Of course selecting a faster multiplier yields better results, but PAM-Blox designs can compete with commercial CAD tools.

3.4 Constant(K) Coefficient Multipliers - KCM

Constant(K) Coefficient Multipliers (KCMs) are of interest for many applications including filters and encryption. KCMs are implemented as PaModules. We compare 16 bit KCMs with a throughput of 16, 4 and 1 bits per clock cycle respectively, in figure 4, in order to show the time-space tradeoff for KCMs on Xilinx XC4000 FPGAs. With increasing throughput, we increase the area requirement and decrease latency – trading area for latency and throughput. While this is not surprising, the generator framework allows the designer/CAD software to easily choose and modify the arithmetic unit that fits the specific requirements of the design problem. First, we implemented the fully-parallel KCM proposed in[6] with PAM-Blox, achieving the same performance and area values reported in the application note from Xilinx. Second, we created a digit-serial design for a 16 bit KCM which takes 4 bits at a time at about 1/3 the area of the fully parallel version. The design of this multiplier is related to distributed arithmetic [28], combining multiply-adds into table lookups. While performance over area for this multiplier is worse than for the fully parallel case, the small area of this multiplier allows us to map an entire multiplication-based encryption algorithm onto around 3200 CLBs. More details on the performance and power requirement of this design are discussed later. The relatively small size of the bit-serial KCM (17 CLBs) allows us to fit more than 45 such multipliers on a Xilinx XC4020E with 800 CLBs.

3.5 Compile Time and Xilinx Place-and-Route

The improvement in compile time in figure 2 is a consequence of interpreting Verilog versus direct execution of C++, and optimizations within FPGA Express II. Except for DES, the benchmarks are simple and require almost no optimization. FE II increases structural-like compile time by about half an order of magnitude.

We expected the manual pre-placement to decrease the remaining automated place-and-route time. Instead we found that Xilinx place-and-route performance is dominated by routing. Place-and-route performance therefore varies depending on how easy or hard it is to route the placed design.

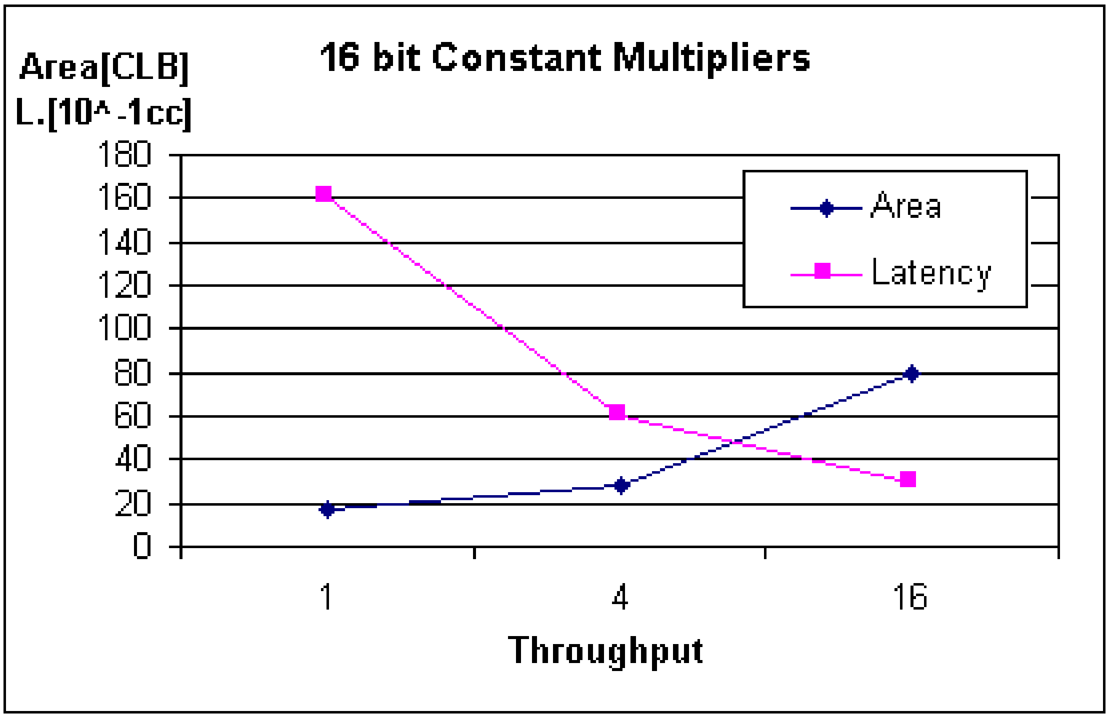

Figure 4: The figure shows Area and Latency for 16 bit constant(K) coefficient multipliers with a throughput of 1,4 and 16 bits per clock cycle. Latency is shown in units of 10^{-1} clock cycles i.e. 160 means 16 clock cycles.

While hand-placement improved circuit performance, place-and-route times varied depending on the specific design, FPGA size and seed number for the non-deterministic place-and-route algorithm.

4 Power

Given a specific application, it is not obvious if an FPGA implementation will use more or less power than an implementation with microprocessors. In order to show the potential of FPGA technology for low-power, programmable systems, we implement an arithmetically challenging, stream-oriented, parallelizable application: IDEA. IDEA – the International Data Encryption Algorithm – is a strong encryption based on (modulo $2^{16}+1$)-multiplication. IDEA was specifically targeted at microprocessors with low-latency multiplication units. Figure 5 shows the data-flow graph of the kernel loop of IDEA. We use the ratio of Performance to Power, or Operations to Energy as the basic metric for comparison of microprocessors, FPGAs, and ASICs. More specifically, the measures for evaluating each design option are Operations per Second per Watt or MOPS per Watt, and Mbits/s per Watt.

A simple method to estimate the power consumption of an FPGA design is shown in [29]. We use a simple model for estimating internal power:

$$InternalPower \ \propto \ numberofCLBs * Frequency \tag{3}$$

$$(Performance/Power) \propto (numberofCLBs)^{-1} \tag{4}$$

First, consider the implementation of IDEA on two recent microprocessors: TMX 320C6x from Texas Instruments and StrongARM SA-110 from DIGITAL. The 'C6x microprocessor is a high performance microprocessor with 2 multipliers, 4 ALUs, and a 8-issue VLIW architecture, requiring peak power of 6 Watt at 200 MHz. The StrongARM has only 1 three-stage multiplier and in-order execution, requiring peak power 1 Watt at 200 MHz. While average power consumption of a microprocessor is considerably lower than peak power consumption, compute intensive tasks such as encryption tend to use above average power. Given the available resources on each microprocessor, the lower bound on completing the computation of one round of IDEA encryption on the 'C6x is 30 clock cycles, compared to 50 clock cycles on the StrongARM. A performance estimate based on resource limitations gives us an upper bound on the performance of IDEA on the microprocessors. Comparing real performance numbers on FPGAs to this optimistic upper bound gives us a good lower bound on the improvement in terms of $\frac{Performance}{Power}$.

We implemented[1] a high-throughput design on the PCI Pamette. Maximum pipelining and a custom designed constant(K) coefficient multiplier with minimal area requirements, lead to a high-performance and low-power FPGA design. The high performance is achieved by complete loop unrolling of the kernel loop. All the multiplications in IDEA are multiplying a data word with a word from the fixed key, resulting in small, constant(K) coefficient multipliers. Maximum pipelining leads to a 56 stage pipeline with a latency of 4 clock cycles

[1] A study based on a paper design of IDEA was presented in [17].

per stage. The eight iterations of the IDEA kernel loop fill four Xilinx XC4020 FPGAs or 3200 Configurable Logic Blocks.

In Figure 3 performance with respect to enciphering with IDEA is given in MBits/s, while the performance with respect to computation of IDEA is given in mega operations per second (MOPS).

4.1 Microprocessor versus FPGA versus ASIC

The high throughput implementation of IDEA on FPGAs outperforms the older ASIC "VINCI" [8], since FPGAs use a 0.35 μm CMOS processes and "VINCI" (in 1993) used a 1.2 μm CMOS process. More recent(1999) results from ASCOM[2] show IDEA encryption at 720/240 Mbits/s, at 0.25μm technology, and 100MHz, requiring 35K Synopsys gate equivalents. Although power numbers are not available, we see that our IDEA implementation compares favorably even with this recent VLSI implementation.

Figure 6 shows the final comparison of performance over power. Trading latency for throughput results in a very efficient design for FPGAs. The low-power FPGA is more than 6 times more efficient in terms of performance over power than the most optimistic execution of IDEA on current state-of-the-art processors. One design limitation is that we have to load the key into the lookup table prior to ciphering. The latency of loading 128 lookup tables with 16 bytes each, is limited by the available bandwidth to the design. We assume a relatively infrequent change of the encryption key.

Due to the heavy use of multiplications, IDEA turned out to be a challenging example to demonstrate the advantages of FPGAs for high throughput and latency tolerant applications.

5 Conclusions

We show a first step towards improving the programmability of FPGA based computing machines. We believe that the object-oriented circuit generator framework offers a good starting point for a high-level compiler for reconfigurable logic.

In order to select FPGAs over a processor or custom VLSI for the computation of part of the application, we have to look at performance, power, and programmability together. As a rule of thumb for predicting high performance of an application on FPGAs, we confirm the following indicators: a large set of data or long runtime to mask reconfiguration time, pipelinability (no critical feedback loops), very high degrees of bit- or ALU-level parallelism, application specific numbers systems or arithmetic functions (e.g. multiply mod $2^{16} + 1$). We show that FPGAs can indeed lower the power consumption of at least one class of applications, namely encryption. The power advantage is inherent to the way computation is done on an FPGA and therefore can be expected to be useful for many other compute intensive applications.

Experience with PAM-Blox has solidified the assumption that object-oriented design of hardware units can improve design time of optimized circuits, simplify cooperation, and enhance maintainability of complex hardware designs. The next goal is to further simplify the programming model without sacrificing any of desired parameters described above.

IDEA Kernel Loop Data-Flow Graph

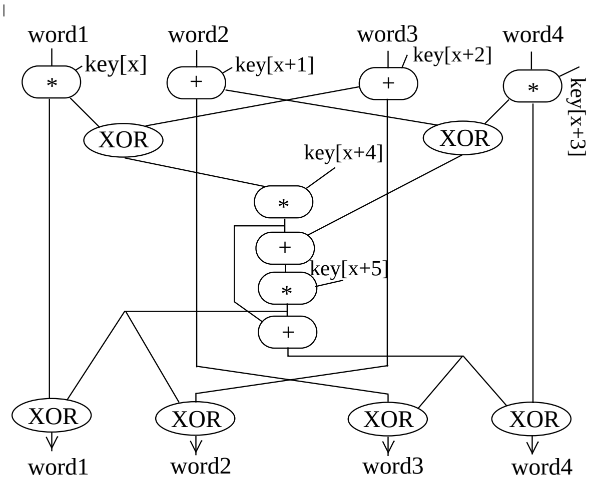

Figure 5: Four 16 bit words of data start in *word1-4*. *key* is a pointer to the array of 52 sub-keys, 16 bits each. The encoded block is returned in word1 to word4 after 8 rounds. '*' stands for multiplication modulo $2^{16}+1$.

Table 3: Performance and Power

The table shows the maximal bit-rate, Mega Operations per Second MOPS for 4 different technologies. One operation corresponds to one circle in Figure 5. Processors, ASICs and FPGAs use CMOS technology. Power for processors are based on published peak power consumption. Power estimates for FPGAs are based on the methodology in section 4.

Design	Processor	Processor	FPGA	ASIC
	TI C6x	SA-110	XC4000XL	"VINCI"
Process	0.25 μm	0.35 μm	0.35 μm	1.2 μm
Area	1 chip	1 chip	3200 CLBs	$107.8mm^2$
Mbits/s	53.1	32	528	180
MOPS	93	56	924	315
[MHz]	200	200	33	25
Watt	6	1	3.15	1.5

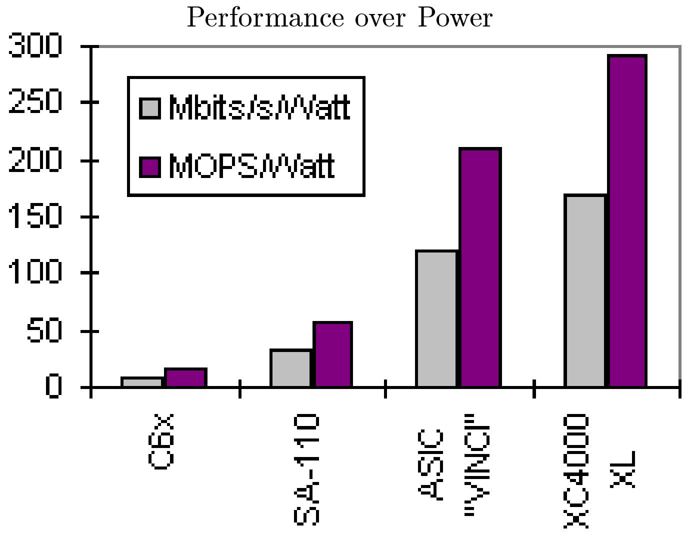

Figure 6: MOPS/Watt determines the power consumption of the technology for a fixed data rate, e.g. 56 Kbits/s modem speed.

A natural extension into behavioral synthesis is to include behavioral methods within the hardware object. A synthesis step can then transform the behavioral method to multiple structural methods , exploring the design space within the hardware object.

Acknowledgments

This research is supported by DARPA Grant No. DABT63-96-C-0106 and equipment by DIGITAL. We would like to thank Mark Shand and Stephan Ludwig of DIGITAL Systems Research Center in Palo Alto for their help and support. DES encryption was implemented by Hon-Cheong Leung. Matrix multiplication was implemented by Hyuk-Jun Lee.

Appendix A

We include a detailed example of a PaModule object, a CORDIC arithmetic unit. The CORDIC takes a vector (x, y) and rotates it by the given angle (phi) in a pipelined fashion resulting a throughput of 33 million rotations per second (at 33MHz).

The relative placement of the adders is inherited from the parent adder object. The add/sub object inherits most of it's functionality from the parent adder, and passes this functionality on to the CORDIC object, hiding the complexity of the carry-chains of the adders.

In fact, the given PAM-Blox code for a CORDIC is shorter than the corresponding behavioral description in Verilog for Synopsys FPGA Express II. For details about our CORDIC work see [20].

```
Cordic(WireVector<Bool, N>& Xval,   //contructor-inputs
       WireVector<Bool, N>& Yval,   // vector(X,Y)
       WireVector<Bool, N>& PhiVal, // rotation angle
       Bool *clock=NULL,            // [optional]
       const char *name=NULL,       // [optional]
       int *sequence=NULL):         // [optional]
    PMtop(clock,name){
    ...
  for (i=0;i<N;i++){
    X[0][i]=Xval[i]; Y[0][i]=Yval[i]; Z[0][i]=PhiVal[i];}
}
void out(WireVector<Bool, N>& Xout,  // method-outputs
         WireVector<Bool, N>& Yout){
  for (i=0; i<N;i++){                // atan table
      tablevals[i]=(int)(pow(2,(N-2))*
           atan(pow(2.0,(double)-ShiftSequence[i])));}
// shifted values
  for(i=0;i<S;i++){
    W.ShiftRight2sComp(X[i],shiftedX[i],ShiftSequence[i]);
    W.ShiftRight2sComp(Y[i],shiftedY[i],ShiftSequence[i]);
    alias(signZ[i],Z[i][N-1]);
    notSignZ[i]=~signZ[i];}
// CORDIC pipeline
  for(i=0;i<S;i++){
    ASX[i]=new AddSub<N>(X[i],shiftedY[i],notSignZ[i]);
    ASX[i]->out(X[i+1]);
    ASX[i]->place();
    ASY[i]=new AddSub<N>(Y[i],shiftedX[i],signZ[i]);
    ASY[i]->out(Y[i+1]);
    ASY[i]->place();
    if (i<(S-1)){
      W.ConstantVector(TVal[i],tablevals[i]);
      ASZ[i]=new AddSub<N>(Z[i],TVal[i],notSignZ[i]);
      ASZ[i]->out(Z[i+1]);
      ASZ[i]->place();}
}
void place(){                        // relative placement
  int i;
  for(i=0;i<=S;i++){
    Y[i][0]<<=Z[i][0]+OFFSET(0,-(N/2)-1); // place X,Y pipe
    X[i][0]<<=Z[i][0]+OFFSET(0,-N-2);
    if (i<S){
      notSignZ[i]<<=signZ[i];
      if (i>0){
    Z[i][0]<<=Z[0][0]+OFFSET(i,0);}
    }
}
...
}
```

Figure 7: The constructor and structural method of a CORDIC arithmetic unit.

References

[1] H. Al-Twaijry with M. Flynn, "Area and Performance Optimized CMOS multipliers," PhD Thesis, Stanford, Aug. 1997

[2] ASCOM, "IDEAcrypt kernel," http://www.ascom.ch/infosec/idea/kernel.html, 1998.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," IEEE Symposium Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997.

[4] J. Babb, R. Tessier, A. Agarwal, "Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pages 142-151, Napa, CA, 1993.

[5] P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: A Performance Assessment," ACM FPGA, February 1992.

[6] K. Chapman, "XApp054: Constant Coefficient Multipliers for the XC4000E," http://www.xilinx.com/xapp/xapp054.pdf

[7] W.B. Culbertson, R. Amerson, R.J. Carter, P. Kuekes, G. Snider, "Defect Tolerance on the Teramac Custom Computer," IEEE Symposium Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997.

[8] A. Curiger, H. Bonnenberg, R. Zimmermann, N. Felber, H. Kaeslin, W. Fichtner, "VINCI: VLSI Implementation of the New Secret-Key Block Cipher IDEA," IEEE Custom Integrated Circuits Conference, 1993.

[9] Duncan A. Buell, Jeffrey M. Arnold, Walter J. Kleinfelder, "Splash-2, FPGAs in a Custom Computing Machine," IEEE Computer Society Press, 1996

[10] S. Gehring, S. Ludwig, "The Trianus System and it's Application to Custom Computing," 6th International Workshop on Field-Programmable Logic and Applications, FPL, September 1996.

[11] M. Gokhale, R. Minnich, "FPGA Computing in Data C," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1993.

[12] S. Guccione, M. Gonzalez, "A data-parallel programming model for reconfigurable architectures," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1993.

[13] J. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, 1998.

[14] Brian von Herzen, "Signal Processing at 250MHz Using High-Performance FPGA's," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 6, No. 2, June 1998.

[15] F. F. Lee with M. Flynn "A Scalable Computer Architecture for Lattice Gas Simulation," PhD Thesis, Stanford, June 1993

[16] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, H. Spaanenburg, "Seeking Solutions in Configurable Computing," IEEE Computer Magazine, December 1997.

[17] O. Mencer, M. Morf, M. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," International Conference on Application Specific Signal Processing, Seattle, May 1998.

[18] O. Mencer, M. Morf, M. J. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, 1998.
http://umunhum.stanford.edu/PAM-Blox/

[19] O. Mencer, M. Shand, M. J. Flynn, "FireLink: A Reconfigurable Firewire Testbed," DIGITAL Systems Research Center, Technote 1998-012, June 1998.

[20] O. Mencer, M. Morf, M. J. Flynn, "Parallel CORDICs for Reconfigurable Computing," The Sixth FPGA / PLD Design Conference and Exhibit, Pacifico Yokohama, Yokohama, Japan, June 24-26, 1998.

[21] Microprocessor Report, "Triscend E5 Reconfigures Microcontrollers," page 12, Nov. 16, 1998.

[22] T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, 1998.

[23] S. Oberman with M. Flynn, "Design Issues in High Performance Floating Point Arithmetic Units," PhD Thesis, Stanford, Jan. 1997

[24] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale, "The NAPA adaptive processing architecture," Proc. IEEE Symposium on FPGAs for Custom Computing Machines, Napa, CA, 1998.

[25] E. Schwarz with M. Flynn, "High-Radix Algorithms for High-Order Arithmetic Operations," PhD Thesis, Stanford, Apr. 1993

[26] M. Shand, J. Vuillemin, "Fast Implementations of RSA Cryptography," 11th IEEE Symposium on Computer Arithmetic, Windsor, ONT, Canada, 1993

[27] M. Shand, *The PCI Pamette FPGA board at DEC Systems Research Center*, http://www.research.digital.com/SRC/pamette/

[28] S. White, "Applications of Distributed Arithmetic to Digital Signal Processing," IEEE ASSP Magazine, p4-21, July 1989

[29] Xilinx, "Application Brief: A Simple Method of Estimating Power in XC4000XL/EX/E FPGAs," http://www.xilinx.com/xbrf/xbrf014.pdf