A Quantitative Analysis of the Speedup Factors of FPGAs over Processors

Zhi Guo
Electrical Engineering
University of California Riverside

Walid Najjar Frank Vahid
Computer Science and Engineering
University of California Riverside

Kees Vissers
University of California Berkeley

ABSTRACT

The speedup over a microprocessor that can be achieved by implementing some programs on an FPGA has been extensively reported. This paper presents an analysis, both quantitative and qualitative, at the architecture level of the <u>*components*</u> *of this speedup. Obviously, the spatial parallelism that can be exploited on the FPGA is a big component. By itself, however, it does not account for the whole speedup.*

In this paper we experimentally analyze the remaining components of the speedup. We compare the performance of image processing application programs executing in hardware on a Xilinx Virtex E2000 FPGA to that on three general-purpose processor platforms: MIPS, Pentium III and VLIW. The question we set out to answer is what is the inherent advantage of a hardware implementation over a von Neumann platform. On the one hand, the clock frequency of general-purpose processors is about 20 times that of typical FPGA implementations. On the other hand, the iteration level parallelism on the FPGA is one to two orders of magnitude that on the CPUs. In addition to these two factors, we identify the efficiency advantage of FPGAs as an important factor and show that it ranges from 6 to 47 on our test benchmarks. We also identify some of the components of this factor: the streaming of data from memory, the overlap of control and data flow and the elimination of some instruction on the FPGA. The results provide a deeper understanding of the tradeoff between system complexity and performance when designing Configurable SoC as well as designing software for CSoC. They also help understand the one to two orders of magnitude in speedup of FPGAs over CPU after accounting for clock frequencies.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.3 [**Special-purpose and Application-based Systems**]: Signal processing systems; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Measurement, Performance, Experimentation, Languages

Keywords

RECONFIGURABLE COMPUTING, FPGA, VHDL, PERFORMANCE, ANALYSIS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA '04, February 22-24, 2004, Monterey, California, USA.
Copyright 2004 ACM 1-58113-829-6/04/0002…$5.00.

1. INTRODUCTION

The speedup over a traditional CPU that can be achieved by implementing a computation as a circuit on an FPGA has been reported many times in the technical literature. The objective of this paper is NOT to report yet another speedup. Rather, it is to present an analysis, both quantitative and qualitative, of the *components* of this speedup at the architecture level. Parallelism is the most notorious of these factors: Large FPGAs allow the user to implement multiple copies of the same computation by unrolling or strip-mining loops. Typical applications that are mapped onto FPGAs, such as signal, image and video processing applications, involve complex computations on a large volume of streaming data. Such applications can benefit tremendously from on-chip parallelism. However, the parallelism by itself does not account for the whole speed-up which is often much larger. Furthermore, when we factor into this equation the ratio of clock frequencies, the speed-up appears to be even larger when counted in number of clock cycles.

The last few years has seen the introduction of a number of Configurable System-on-a-Chip (CSoC) platforms that combine one or more CPU cores, an FPGA-based reconfigurable fabric, as well as memory blocks on a single chip. These amazing computing devices have the flexibility of software yet can approach computing speeds of custom hardware. The earliest example is that of the Triscend [12] E5 followed by the Triscend A7, the Xilinx [13] Virtex II Pro, and the Altera [14] Excalibur. The capabilities of these platforms span a wide range. At the low end, the Triscend A7 consists of a 60 MHz ARM CPU with about 20,000 programmable gates. At the high end, the Xilinx Virtex II Pro

2VP125 consists of about 10 million gates, four PowerPC 405 CPUs each running at 400 MHz, 10 Mbits of BlockRAM, 556 18x18-bit multipliers and 3.125 Gbps off-chip bandwidth. Paradoxically, the same advances in process technology that made CSoCs a reality have made it less economically feasible to develop large-scale ASICs. Mask costs alone are about $1 million in today's technologies, and are expected to double or triple with every new technology node.

FPGAs, whose capacities have become truly massive, have been shown to achieve huge speedups over microprocessors for a wide variety of applications [1][3][16][17][18][19]. Recently, BDTI [15] reported that an Altera Stratix EP1S20-6 could accommodate more than 60 times the channel capacity of a Motorola MSC8101 DSP running at 300 MHz.

With the introduction of these platforms, it is now feasible to combine on a same chip the two styles of computations: temporal, on the CPU, and spatial, on the reconfigurable fabric. CSoCs are therefore ideal platforms for embedded applications that combine both control and data intensive computations. Image and video processing applications, among many others, fall into this category.

The objective of this study is a quantitative evaluation of the various factors that contribute to the speedup achieved by FPGA implementations over traditional, and less traditional, CPUs. We believe that a better understanding of these issues can be of great help for current and future applications of CSoC, for the design of more efficient configurable fabrics and in general for a more targeted hardware/software codesign of embedded systems.

Ours is not an exhaustive analysis – we have limited ourselves, for convenience, to only one FPGA-based reconfigurable platform, the Xilinx Virtex architecture, and to one type of application: data parallel compute intensive programs drawn from image processing applications. It is well known that the clock speeds that can be realized on FPGA platforms fall well behind the ones that are customary on microprocessors, even low cost microprocessors intended for large volume embedded applications. A typical embedded CPU runs at a frequency 6 to 15 times that of a typical FPGA implementation.

DeHon [10] compares the computational densities that can be obtained on CPUs, FPGAs and ASICs. He shows that while the computational density on FPGAs is lower than ASICs, it is still much larger than what is achieved on a CPU. Of course, FPGAs, compared with ASICs, have the advantage of being programmable.

The rest of this paper is organized as follows. In Section 2, we describe the benchmarks used as well their implementation on a CSoC. Section 3 reports on the comparison with three general-purpose processors: MIPS,

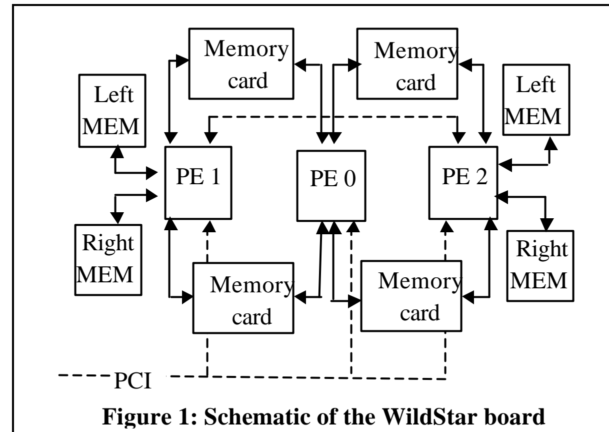

Figure 1: Schematic of the WildStar board

Pentium III and a VLIW machine. Section 4 gives an in-depth analysis of the factors accounting for the performance difference. Section 5 provides conclusions.

2. CSOC IMPLEMENTATION

In this section, we describe the three benchmarks used and their implementation on the reconfigurable platform.

A reconfigurable computing system usually consists of a number of reconfigurable devices with local memory chips and a bus to the host microprocessor if any. We use as our platform the Annapolis Microsystems WILDSTAR board [8]. The WILDSTAR board has three Xilinx Virtex XCV2000E FPGA chips, synchronous SRAM as local memory, and connects with the host by a PCI bus (Moll and Shand [11] measured the performance of the interface between a reconfigurable computing platform and its host, particularly on the PCI bus.). Standard VHDL modules can be used to design the interfaces to access and control the on-board components. Each of the FPGAs has an equivalent of two million programmable gates. In all applications, we have used only one FPGA. A schematic of our platform is shown in Figure 1.

2.1 The Benchmarks

We have selected, intentionally, benchmarks from video and image processing applications. These types of programs are computationally intensive and therefore favor an FPGA-based implementation. Control flow intensive applications would obviously favor CPU-based platforms. Our objective is an analysis of the "why" and "how" of the FPGA advantage over CPU platforms on compute intensive applications.

We have used three image processing benchmarks:

- Prewitt: an edge detection algorithm.
- Wavelet transform: used in the JPEG 2000 image compression standard.

- Max filter: computes the maximum pixel in a window of an image.

2.1.1 Prewitt

Prewitt edge detection is a common algorithm in the image processing area. An n×n mask window slides over an 8-bit image. For each window, a convolution is computed with both an n×n vertical mask and an n×n horizontal mask. The result is the geometric average of these two values. The algorithm is described in the following equations, where n is 3.

$$mask_{vert} = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{pmatrix} \quad (1)$$

$$mask_{hort} = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{pmatrix} \quad (2)$$

$$pixel_{output} = \sqrt{mask_{vert}^2 + mask_{horz}^2} \quad (3)$$

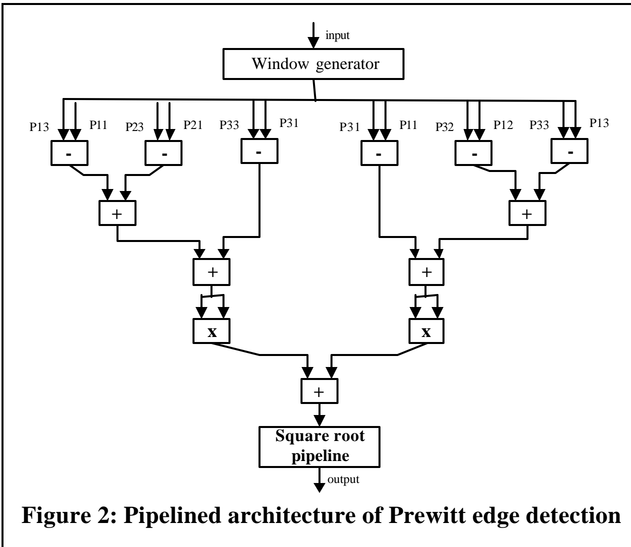

Figure 2: Pipelined architecture of Prewitt edge detection

Obviously, we don't have to do array multiplications. We only need to do 3 subtractions and 2 additions per mask to compute the convolutions. The geometric average consists of two multiplications, one addition and one square root operation. The implementation of the square root operation is based on the pipelined scheme described in [2]. The pipeline of the entire calculation consists of 12 stages and is shown in Figure 2. The square root pipeline accounts for eight stages, with one subtraction in each stage. The total computations for every output pixel consist of 19 additions/subtractions plus two multiplications.

2.1.2 Wavelet

The wavelet transform algorithm we have used is based on a 5×5 sliding window with a 2-pixel step in both the horizontal and vertical directions. The calculations are shown in Equations (4) and (5), where d_0 and d_{01} are the outputs for every column of five pixels (P_{11} to P_{15}). So for five columns, we get 5 d_0 and 5 d_{01}. Then both the columns of d_0 and d_{01} are calculated using the same equations again. Therefore, each 5×5 window generates 4 output pixels. Since the window is sliding in a 2-pixel step both in horizontal and vertical directions, we still get one output pixel per input pixel on average.

$$d_0 = \begin{pmatrix} -1 & 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \end{pmatrix}$$

$$d_1 = \begin{pmatrix} -1 & 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} p_{13} \\ p_{14} \\ p_{15} \end{pmatrix} \quad (4)$$

$$d_{01} = (d_0 + d_1) >> 3 \quad (5)$$

2.1.3 Maximum Filter

The maximum filter program is similar to the window in Prewitt edge detection. A 3×3 window slides over an image. The maximum value of the 9 pixels is the output pixel of the current window.

On the WildStar board, the data bus between the FPGA and memory is 64 bits wide, which allows the fetching of eight pixels in parallel.

In both the Prewitt edge detection and the maximum filter cases, eight windows are computed simultaneously, which means that eight pipelined iterations, as shown in Figure 2, are mapped on the FPGA.

In the wavelet transform case, four windows are computed concurrently. Each window generates four 8-bit data items every two clock cycles. Therefore all three circuits achieve an output of eight pixels per cycle. Notice that none of these applications has loop carried data dependencies.

As with most image and video processing applications, all three benchmarks rely on sliding windows over an image. This implies every pixel value is used by more than one iteration. For example, every column of output in Prewitt edge detection, which has eight pixels, depends on both the present eight rows of input data and the last two rows of input data of the previous eight rows. We store the last two pixels of each 64-bit input in a configurable dual-ported memory on the FPGA. Therefore no pixel has to be read twice from memory and we get eight output pixels every

cycle. Furthermore, all the on-chip memory access can be done within one clock cycle. Therefore, we achieve seamless connection between memory access and execution benefiting from the device's distributed memory and post-fabrication programmability characteristic.

Table 1: Hardware performance of the FPGA

	Freq. (MHz)	Number of clock cycles	Cycles/output pixel	Ops/pixel
Prewitt	33.3	131072	0.125	61
Wavelet	35.8	130944	0.125	15.75
Max filter	41.2	131072	0.125	8

For the FPGA platform we use the Xilinx Virtex XCV2000E FPGA chip. The VHDL codes are synthesized by Synplify 7.0 [4] and compiled using the place-and-route tools of Xilinx Project Navigator 4.1i [5].

The statistics for all the benchmarks are shown in Table 1. The input image size is 1024×1024 for each application. Each application has the same throughput: eight pixels per cycle. Note that the data is given per output pixels. Even though the input image is the same size, the number of output pixels depends on the filer size: 3x3 for Prewitt and Max Filter and 5x5 for Wavelet.

3. PROCESSOR PLATFORMS

We compare the same benchmarks written in C on the following general-purpose platforms: MIPS, Pentium III and VLIW. In all three cases we assume a perfect cache to factor out the effects of cache misses. We also make the input data set size large enough to ensure that the effects of other computations are negligible.

For the MIPS processor we use the Simplescalar simulator [6]. We generate the timing statistics using Simplescalar out-of-order simulation.

We use the VTune Performance Analyzer 7.0 [7] from Intel to evaluate the performance of the code running on a Pentium III. VTune collects data of the entire system with a very low intrusion level. We make sure that the target application program has over 99.5% of the processor resources, while other services, including operating system and VTune itself, use the rest. The programs are compiled using the Microsoft Visual C++ compiler.

For the VLIW platform we use the VEX compiler and simulator system from HP Labs[1]. The scalability of the VEX ISA enables users to change the number of clusters, functional units, registers and memory ports. We use the following configuration in the VEX system: four dusters (which is the maximum) with four ALUs, two multipliers and three memory units per cluster. Using a pragma in the source code, the loops are unrolled four times.

Table 2: Statistic information of the three applications on MIPS with perfect cache

	Instructions executed	Clock cycles	CPI	Instr/ pixel
Prewitt	550125282	261121454	0.476	527
Wavelet	205502954	101263275	0.455	198
Max filter	390267442	171050813	0.438	374

Table 3 - Statistic information of the three applications on Pentium III

	Instructions executed	Clock cycles	CPI	Instr/pixel
Prewitt	394475180	327414400	0.83	378
Wavelet	123136713	221030400	1.79	118
Max filter	219726901	236865600	1.08	210

Table 4 - Statistic information of the three applications on VLIW with perfect cache

	Instructions executed	Clock cycles	CPI	Instr/pixel
Prewitt	112015182	75722263	0.676	107
Wavelet	61350478	49877939	0.813	59.0
Max filter	57607148	81226079	1.41	55.2

Table 2, Table 3 and Table 4 show the execution statistics of the three applications on the MIPS processor, the Pentium III processor and the VLIW, respectively.

4. ANALYSIS OF SPEEDUP FACTORS

The clock frequency is obviously a very important factor in comparing the performance of microprocessors and FPGA-based implementations. Typical embedded microprocessors have frequencies ranging from 100 to 600 MHz whereas many FPGAs are rated for clock frequencies in the 30 to 100 MHz range. Most typical applications that are mapped onto FPGAs achieve frequencies ranging from 30 to 60 MHz as shown in Table 1. The clock frequency alone, therefore, accounts for a factor ranging from 3 to 10 in favor of microprocessors. However, the speedup in terms of clock cycles, as shown in Table 5, ranges from 381 to 2498. This

[1] VEX (VLIW Example) is a VLIW compiler and simulator with a configurable machine model developed at HP Labs, Cambridge MA. It is not yet publicly available.

leads us to conclude that after accounting for the clock frequency difference, *the FPGA implementations are one to two orders of magnitude faster than the CPUs.*

The objective of our analysis is to identify and quantify the parameters that determine this speedup. We will therefore not consider the clock frequency issue any further in our discussion.

Table 5: Speedup of FPGA implementation in clock cycles

	Prewitt Edge Detection	Wavelet Transform	Maximum Filter
MIPS	1992	773	1305
Pentium	2498	1688	1807
VLIW	578	381	620

Table 6: Iteration level parallelism for all four platforms

	FPGA	MIPS	Pentium III	VLIW
ItLP	8	1	1	4

Table 6 gives the iteration level parallelism for all four platforms. The remainder of this section is an analysis of the components of this speedup.

4.1 The Factors of the Speedup

In this section we present an analysis of the factors that contribute to the speedup. The type of applications that we are interested in consist of a loop nest that accounts for a very large percentage of the computation. We will therefore focus the analysis on this loop nest ignoring the remainder of the computation. Equation (6) gives the definition of total number of clock cycles on a general-purpose processor for a given computing task, where

$Instr_{iterat}$ = Total number of instructions per iteration.

N_{iterat} = Total number of outer iterations, in our case the number of pixels in the output image.

CPI = Average number of clock cycles per instruction.

$$Cycle_{CPU} = Instr_{iterat} \times N_{iterat} \times CPI \qquad (6)$$

Some of the instructions in a loop iteration are directly related to the pixel operations while others are "support" instructions. These instructions are used to load/store the data in memory, to manipulate the program counter (PC) to make sure the computation to be carried out in a correct sequence. Thus, we can divide the total number of instruction per iteration $\boldsymbol{Instr_{iterat}}$ into two parts, shown in Equation (7),

$$Instr_{iterat} = Instr_{sppt} + Instr_{operat} \qquad (7)$$

where $Instr_{operat}$ is the number of instructions that carry out the calculation directly, and $Instr_{sppt}$ is the number of support instructions.

Each loop in the FPGA yields one output pixel every clock cycle. Equation (8) gives the total number of clock cycles to perform the same calculation in the FPGA,

$$Cycle_{FPGA} = \frac{N_{iterat}}{P_{iterat}} \qquad (8)$$

where P_{iterat} is the number of parallel loops in the FPGA, listed in Table 6. We define the FPGA overall speedup over general-purpose processor in Equation (9).

$$Speedup_{overall} = \frac{Cycle_{CPU}}{Cycle_{FPGA}} \qquad (9)$$

If we substitute Equation (6), (7) and (8) into Equation (9), we can get Equation (10)

$$\begin{aligned} Speedup_{overall} &= \frac{Instr_{iterat} \times N_{iterat} \times CPI}{\frac{N_{iterat}}{P_{iterat}}} \\ &= Instr_{iterat} \times CPI \times P_{iterat} \qquad (10) \\ &= (Instr_{sppt} + Instr_{operat}) \times CPI \times P_{iterat} \end{aligned}$$

We can see that the overall speedup can be divided into three factors: iteration level parallelism ratio (P_{iterat}), CPI, and the summation in the parentheses. In order to make the speedup analysis more clear, we define the instruction efficiency in Equation (11), which is the percentage of the pixel operations to support instructions.

$$\begin{aligned} Efficiency &= \frac{Instr_{operat}}{Instr_{iterat}} \\ &= \frac{Instr_{operat}}{Instr_{sppt} + Instr_{operat}} \end{aligned} \qquad (11)$$

For convenience, we define the reciprocal of instruction *efficiency* as *instruction inefficiency* in Equation (12)

$$\begin{aligned} Inefficiency &= \frac{Instr_{iterat}}{Instr_{operat}} \\ &= \frac{Instr_{sppt} + Instr_{operat}}{Instr_{operat}} \end{aligned} \qquad (12)$$

Substituting Equation (12) into Equation (10) yields Equation (13).

$$Speedup_{overall} = P_{iterat} \times Instr_{operat} \times Inefficiency \times CPI \qquad (13)$$

We have three factors in Equation (13):

1. P_{iterat} is the iteration level parallelism *ratio*,

2. $Instr_{operat}$ is usually greater than the number of operations of one loop in FPGA since the implementation of some instructions, such as a shift, in FPGA doesn't take any clock cycle. $Instr_{operat}$'s lower bound is the number of operations of one loop in FPGA

3. *Inefficiency* reflects the fact that CPUs has to execute a number of support instructions to carry-out the computation.

We will discuss these three factors in the following subsections.

4.2 Iteration Parallelism

One immediate advantage of FPGAs is the ability to run several concurrent iterations on the hardware. In all three programs, eight pixels are fed into the circuit and eight output pixels are retrieved every cycle. Note that the memory data bus is 64-bits wide, which results in eight pixels per memory access. For all three benchmarks, the iteration level parallelism on the FPGA is eight. On the MIPS and Pentium it is one, and it is four on the VEX. These values are summarized in Table 6.

Table 7 shows the FPGA speedup when the iteration level parallelism advantage is factored out. One can observe that the speedup over all the platforms and on all the benchmarks is now within the same order of magnitude.

Table 7: Speedup of FPGA factoring out iteration level parallelism

	Prewitt Edge Detection	Wavelet Transform	Maximum Filter
MIPS	249	96.7	163
Pentium	312	210	226
VLIW	289	190	310

4.3 The Number of Necessary Operations

As mentioned above, the number of arithmetic and logic operations per pixel running on a CPU for a given computing task is greater than that on a FPGA for the same task. Instructions such as shift, or a multiplication by a power of two do not take any cycle time in hardware. Bit extracting takes a group of instructions in CPUs, while is implemented only by wires, and of course, doesn't take any clock cycles.

In Table 8 we can see that for Maximum Filter, both the FPGA and the CPU have the same number of operations per pixel because the only effective arithmetic operation in these two architectures is comparison. However, for Prewitt Edge Detection, the FPGA saves many operations, which are all shift operation that are used to do square root.

Table 8 - ALU Operations/pixels of CPUs and FPGAs

	Prewitt Edge Detection	Wavelet Transform	Maximum Filter
CPU	61	15.75	8
FPGA	21	10.5	8

4.4 The Efficiency Advantage

Table 2, Table 3 and Table 4 show the numbers of executed instructions for each benchmark on the three CPUs. The output image sizes are known, so that we can get the numbers of executed instructions for each iteration and list in the third column of Table 9. The second row of Table 8 lists the numbers of ALU operations/pixel on CPUs per iteration. The numbers of support instructions per iteration are listed in the forth column of Table 9. By using Equation (12), the instruction inefficiencies are listed in the last column of Table 9.

Table 9 - The Instruction Inefficiency

Benchmarks	CPU	Instr./pixel	Support instr./pixel	*Inefficiency Factor*
Prewitt Edge Detection	MIPS	527	466	8.64
	Pentium III	378	317	6.19
	VLIW	428	367	7.02
Wavelet Transform	MIPS	198	180	12.5
	Pentium III	118	102	7.51
	VLIW	236	175	15.0
Maximum Filter	MIPS	374	366	46.7
	Pentium III	210	202	26.3
	VLIW	221	160	27.6

The most fundamental difference between a general-purpose processor and a reconfigurable computer is that the former time-multiplexes the operation of the entire task on one datapath while the later can be programmed to perform the same operation repeatedly on a stream of data. In the latter, we build a hardwired datapath, plus distributed memory if needed, to carry out just one computing task efficiently. Special interconnections define the order of the operations on the data stream. No data item needs to be re-read from memory since configurable data storage can be customized for each data path separately.

Below is a summary of the factors that cumulatively form the instruction efficiency associated with the von Neumann model as compared to an FPGA implementation:

- *Sequential execution*. The von Neumann model is inherently sequential. Extracting parallelism, at compile

time or run-time, involves a substantial overhead. Pipelining is an example of an architecture mechanism that exploits instruction level parallelism at run-time. It suffers from pipeline stalls and delays when instructions are dependent.

- *Control flow*. In the von Neumann model, the control flow and dataflow instructions are embedded in the same program and executed sequentially. In the hardware implementation, the two "mechanisms" are separate. Instructions such as branches and jumps are not implemented on the FPGA. Instead, all branches of a control path are implemented and the correct outcome selected (i.e., "if conversion").
- *Large temporary storage* can be implemented on the FPGA. This storage can be used dynamically and selectively without having a large impact on the clock cycle time. This provides powerful support to implement a very large degree of parallelism, specifically iteration level parallelism.

4.4.1 An Example: Square Root

```
lw $v0[2],0($s8[30])
addiu $v1[3],$v0[2],-1
addu $v0[2],$zero[0],$v1[3]
sw $v0[2],0($s8[30])
addiu $v1[3],$zero[0],-1
bne$v0[2],$v1[3],00400280 <sq_root+90>
j 00400370 <sq_root+180>
lw $v0[2],4($s8[30])
sll $v1[3],$v0[2],0x1
sw $v1[3],8($s8[30])
lw $v0[2],16($s8[30])
sll $v1[3],$v0[2],0x2
sw $v1[3],20($s8[30])
lw $v1[3],8($s8[30])
sll $v0[2],$v1[3],0x1
ori $v1[3],$v0[2],1
lw $v1[3],0($s8[30])
sll $v0[2],$v1[3],0x1
lw $v1[3],12($s8[30])
srlv $v0[2],$v1[3],$v0[2]
lw $v1[3],24($s8[30])
sltu $v0[2],$v0[2],$v1[3]
bne$v0[2],$zero[0],00400348 <sq_root+158>
lw $v0[2],8($s8[30])
ori $v1[3],$v0[2],1
sw $v1[3],4($s8[30])
lw $v0[2],24($s8[30])
sw $v0[2],16($s8[30])
j 00400368 <sq_root+178>
lw $v0[2],8($s8[30])
sw $v0[2],4($s8[30])
lw $v0[2],20($s8[30])
sw $v0[2],16($s8[30])
```

Figure 3: Assembly code, in MIPS, of square root

Figure 3 shows the assembly code of square root subroutine loop body in the Prewitt edge detection on a MIPS processor. This loop is the dominant loop of the entire calculation task. It is invoked once per pixel and iterates eight times for each invocation. There are 11 arithmetic instructions in the loop body, while only one (the *addu* – underlined in the figure) corresponds to the hardware operator we employ in one pipeline stage to carry out the same square root algorithm. As we mentioned above, each square root stage has only one subtraction/addition operator. In fact, it does have two OR operators, but the operators are combined with other circuits and are done with the subtraction/addition together within one clock cycle. The algorithm also needs five shift operators per stage. Reconfigurable device gets them for free using wires, while as shown in Figure 3, a general-purpose processor requires a number of shift instructions. Note also that the MIPS code must use arithmetic instructions to update the loop counter, while the FPGA implementation can accomplish that in parallel. Most of the rest are memory load and store instructions, which are used to store and upgrade the current calculation status since the operators for one or one group of data are sequenced in time. We also have a number of branch instructions in Figure 3. Beside the clock cycles these instructions take, they also partly account for pipeline bubbles.

Notice that all these overhead instructions are in the major loop of the executable, which is expensive.

4.5 Memory Accesses

Memory accesses are amongst the most notorious overhead operations on CPUs. Reducing the total number of memory accesses always has a positive impact on performance and energy consumption. FPGAs allow the user to configure on-chip storage at will and customize it for each loop. In particular, this storage can be used to efficiently reduce the number of memory accessing by reusing data.

Table 10 shows the numbers of the load and store operations used to calculate one output pixel on each of the four platforms. Note that for the three CPUs the compiler optimization levels were set at the highest available level.

Table 10 - Prewitt loads and store per output pixel on the four platforms

		FPGA	MIPS	Pentium III	VLIW
Prewitt	Load	0.125	8	13	8
	Store	0.125	1	7	1
Wavelet	Load	0.125	12	14	8.75
	Store	0.125	7	7	1
Max	Load	0.125	9	9	9
	Store	0.125	1	1	1

FPGA reads and writes eight pixels from/to memory in parallel since the data bus is 64-bit and the pixel is 8-bit in our implementation. For the max filter code, which is the simplest code, all three CPUs have the same accesses: nine pixels read for each pixel written.

The communication between storage units and function units is always one of the most important effects of

computation performance. The analysis above shows the importance of loop level data reuse.

From Table 10 we can see that an extensive optimization of data reuse on the FPGA can impact the number of memory accesses by a factor ranging from 64 to 112 compared with hand optimized VHDL implementation on FPGA. Even if we factor out the difference between FPGA's bit-precision and CPUs' word-precision, which is eight for these three benchmarks, loop level reuse still can reduce memory accesses by 8 to 14 time in our benchmarks.

4.6 Comparing the Speedup Factors

In this section we compare the contribution of the important factors to the speedup. The two that constitute the advantage of the FPGA implementation are the iteration level parallelism and the instruction efficiency they are listed in Table 11. We can see that, in our implementations, instruction efficiencies are comparable to iteration level parallelism for Prewitt Edge Detection and Wavelet Transform. For Maximum Filter, the efficiency is much more significant than iteration parallelism because this benchmark has relative simple calculation and relatively high control density. In the FPGA implementation, the latency of control operations is hidden.

Table 11 - The Efficiency Factor Compared with the Iteration Level Parallelism Factor

Benchmarks	CPU	Iteration level parallelism ratio	Inefficiency Factor
Prewitt Edge Detection	MIPS	8	8.64
	Pentium III	8	6.19
	VLIW	2	7.02
Wavelet Transform	MIPS	8	12.5
	Pentium III	8	7.51
	VLIW	2	15.0
Maximum Filter	MIPS	8	46.7
	Pentium III	8	26.3
	VLIW	2	27.6

The goal of the speedup analysis is to guide the design of reconfigurable systems. This is particularly relevant for configurable systems-on-a-chip (CSoC) where the designer has the option of a software or hardware implementation. The analysis in this section exposes the following observations as related to Equation 13:

1. The iteration level parallelism is one of the speedup factors. It is limited only by device area and the available I/O or memory bandwidths.
2. Instruction efficiency is another important factor that reflects the architectural difference between FPGA and CPU. This factor is even more important in simple codes (maximum filter) than it is in complex ones (Prewitt and wavelet).
3. Hiding the latency of the supporting operations in parallel with pipelined calculation maximizes the instruction efficiency. If iteration level parallelism is limited by I/O bandwidth, trading area for instruction efficiency is worthwhile. For example, memory accesses ought to be done in parallel with the necessary ALU operations when possible.
4. The streaming of data from memory or I/O to the datapath on the FPGA is a very big advantage that eliminates a large number of support instructions.

5. CONCLUSION

In this paper, we analyzed three image-processing applications (Prewitt edge detection, wavelet transform, and maximum filter) implemented both on an FPGA-based reconfigurable platform and on general-purpose processor platforms (MIPS, Pentium III and VLIW). The objective of our analysis was to identify and quantify the factors that contribute to the speedup achieved on the FPGA over the processors, and to guide the design and implementation of reconfigurable systeams. We show that in spite of the clock cycle advantage of CPUs the instruction efficiency of the FPGA is an important factor. This factor ranges from 6 to 47 on our benchmarks. The instruction efficiency factor can be considered the inherent advantage of FPGAs over the von Neumann model architectures and affect reconfigurable computing systems' performance dramatically. We also show that FPGA implementations are very efficient in term of loading and storing data to/from memory or I/O. This is a result of the streaming computation that is usually implemented. We believe that this quantitative analysis will help shed some light on the 20 to 100 speedup factors that can be achieved by FPGA implementations over general-purpose processors.

6. REFERENCE

[1] J. Villarreal, D. Suresh, G. Stitt, F. Vahid and W. Najjar. Improving Software Performance with Configurable Logic, *Kluwer Journal on Design Automation of Embedded Systems,* November 2002, Volume 7, Issue 4, pp.325 -339.

[2] Y. Li and W. Chu. A New Non-Restoring Square Root Algorithm and Its VLSI Implementations. *ICCD'96, International Conference on Computer Design*, Austin, Texas, October 7 - 9, 1996.

[3] J. Frigo, M. Gokhale and D. Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. *9th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, California, February 2001.

[4] http://www.synplicity.com/

[5] http://www.xilinx.com/

[6] http://www.simplescalar.com/

[7] http://www.intel.com/software/products/vtune/

[8] Annapolis Microsystems Inc. *WILDSTAR hardware Reference Manual.* (http://www.annapmicro.com)

[9] W. Böhm, R. Beveridge, B. Draper, C. Ross, M. Chawathe, and W. Najjar. Compiling ATR probing codes for execution on FPGA hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines,* Napa Valley, California, April 21-24, 2002.

[10] A. DeHon, The Density Advantage of Configurable Computing, *Computer, vol.33.No.4, April 2000*, IEEE Computer.

[11] L. Moll and M. Shand, Systems performance measurement on PCI Pamette, *In FPGAs for Custom Computing Machines (FCCM'97),* April 1997

[12] Triscend Corporation: http://www.triscend.com/

[13] Xilinx, Inc. http://www.xilinx.com/

[14] Altera Corporation. http://www.altera.com/

[15] Berkeley Design Technology, Inc. (BDTI): http://www.bdti.com/

[16] G. Stitt, R. Lysecky and F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. *Design Automation Conference (DAC'03),* Anaheim, California, June 2003.

[17] J. Hauser, J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97),* pages 12-21, Napa Valley, California, April 1997.

[18] G. Brebner. Single-Chip Gigabit Mixed-Version IP Router on Virtex-II Pro, *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02),* Napa, California, September 2002.

[19] F. Cardells-Tormo, J. Valls-Coquillat, V. Almenar-Terre, and V. Torres-Carot. Efficient FPGA-based QPSK Demodulation Loops: Application to the DVB Standard, *12th International Conference on Field Programmable Logic and Applications (FPL'02),* Montpellier, France, September 2002.