Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 518602, 19 pages
doi:10.1155/2011/518602

Research Article

A Dynamic Dual Fixed-Point Arithmetic Architecture for FPGAs

G. Alonzo Vera,[1] Marios Pattichis,[1] and James Lyke[2]

[1] *Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87101, USA*
[2] *Space Electronics Branch of the Space Vehicles, Directorate of the Air Force Research Laboratory, NM 87117-5776, USA*

Correspondence should be addressed to G. Alonzo Vera, alonzo@ieee.org

Received 17 July 2010; Revised 30 November 2010; Accepted 18 January 2011

Academic Editor: Scott Hauck

In FPGA embedded systems, designers usually have to make a compromise between numerical precision and logical resources. Scientific computations in particular, usually require highly accurate calculations and are computing intensive. In this context, a designer is left with the task of implementing several arithmetic cores for parallel processing while supporting high numerical precision with finite logical resources. This paper introduces an arithmetic architecture that uses runtime partial reconfiguration to dynamically adapt its numerical precision, without requiring significant additional logical resources. The paper also quantifies the relationship between reduced logical resources and savings in power consumption, which is particularly important for FPGA implementations. Finally, our results show performance benefits when this approach is compared to alternative static solutions within bounds on the reconfiguration rate.

1. Introduction

In the realm of embedded systems, a designer often faces the decision of what numerical representation to use and how to implement it. Particularly, when using programmable logic devices, constraints such as power consumption and area resources must be tradedoff with performance requirements. Floating point is still too expensive in terms of resources to be intensively used in programmable logic devices. Fixed-point is cheaper but lacks the flexibility to represent numbers in a wide range. In order to increase numerical range, several fixed-point units—supporting different number representations—are required. Alternatively, numerical range can be increased by a single fixed-point unit able to change its binary point position.

In this paper, runtime partial reconfiguration (RTR) is used to dynamically change an arithmetic unit's precision, operation, or both. This approach requires intensive use of partial reconfiguration making it particularly important to take into consideration the time it takes to reconfigure. This time is commonly referred to as the reconfiguration time overhead.

Usually, runtime reconfigurable implementations involve the exchange of relatively large functional units that have large processing times. This, along with low reconfiguration frequencies, significantly reduces the impact of the reconfiguration time overhead on the performance.

Unlike common runtime reconfigurable implementations, the exchangeable functional units in this approach are smaller, and reconfiguration frequencies are larger. Smaller exchangeable functional units are possible by using a dual fixed-point (DFX) numerical representation [1] which provides larger dynamic range than classical fixed-point representations with little extra cost in terms of hardware. We introduce a dynamic dual fixed-point (DDFX) architecture that allows changes in precision (binary point position) based on relatively small changes at runtime in the hardware implementation. Although at a higher reconfiguration time cost, DDFX also allows the dynamic swap between different arithmetic operations.

The improvements in dynamic range and precision allow this approach to find potential applications in problems where only a floating-point solution made sense before. Numerical optimization algorithms are examples of such applications. The iterative nature of these algorithms makes them especially susceptible to numerical issues arising from the use of fixed-point arithmetic. Furthermore, these algorithms usually require extensive calculations, making them good candidates for performance speed-up through parallelization. In that sense, the smaller hardware footprint

of our approach is an advantage as it allows a larger number of DDFX units as opposed to a reduced number of larger floating-point units.

The architecture is compared with hardware implementations of floating-point, fixed-point, and a gcc software emulation of floating point. These alternatives were chosen for comparison because of their widespread use in the industry and their availability for testing. Comparisons are made in terms of resources, power consumption, performance, and precision. Data width is kept constant across comparisons. When comparing resources and power consumption, implementations with similar precision are used. When comparing performance, implementations with similar precision and resource consumption are used. For performance, the architecture is evaluated in the context of linear algebra, which is widely used in scientific calculations. Linear algebra operations are broken into vector operations, and performance is measured in terms of operations per second. The architecture is ported to two of the latest Xilinx device families in order to compare how the families' architectural differences impact the effectiveness of the dynamic approach.

We also present simulation results on the use of DDFX for inverting large matrices. For the Jacobi method, for inverting large linear systems, it is shown that DDFX's results closely approximates that of double precision floating-point.

The paper is organized into six sections. In Section 2, we provide an overview of numerical representations and provide further motivation for dynamic precision. In Section 3, we provide an overview of reconfigurable computing with a particular emphasis on dynamically reconfigurable architectures. We present the proposed dynamic arithmetic architecture in Section 4. A summary of testing platforms and methodology is given in Section 5. Results are given in Section 6. Concluding remarks are provided in Section 7.

2. Numerical Representations

2.1. Dynamic Range and Precision

2.1.1. Dynamic Range. Dynamic range is a quantitative measurement of the ability to represent a wide range of numbers, and it is defined by the relationship between the largest and smallest magnitudes that a numerical format can represent.

$$\text{Dynamic range} = \frac{\text{Largest magnitude}}{\text{Smallest, nonzero magnitude}}. \quad (1)$$

For instance, a 16-bit wide fixed-point format with no binary point can represent a number as large as 2^{15}–1 and as small as 1. Thus, the dynamic range for this specific format is $\approx 2^{15}$.

2.1.2. Precision. Precision is the accuracy with which the basic arithmetic operations $+, -, *, /$ are performed [2]. In floating-point arithmetic, precision is measured by the unit roundoff u such that

$$fl\left(x \ \text{op} \ y\right) = \left(x \ \text{op} \ y\right)(1+\delta), \quad (2)$$

for some δ satisfying $|\delta| \leq u$. Here, $fl()$ denotes the evaluation of an expression in floating point arithmetic, for all of the basic arithmetic operations: $+, -, *, /$. This definition is extensible to fixed-point representations as well.

2.2. Word Length Optimization Techniques and Software. A numerical algorithm's precision and convergence characteristics can benefit from a variable or mixed arithmetic precision implementation [3–5]. Constantinides demonstrated in [6] that determining the optimal word-length under area, speed, and error constraints is an NP-hard problem. There are, however, several published approaches to word length optimization. They can be categorized into two main strategies [7].

(1) Analytical Approach. Classical numerical analysis of the algorithm and modeling of the worst-case error as a function of operand word lengths. Analytical methods attempt to determine the optimal range and precision requirements for each operation based on the input variables representations. Starting from the representation of the input variables, analytical methods generate approximate representations for each operation, in a consecutive order. The goal is to then provide minimum precision and range requirements for each variable so as to guarantee a certain level of accuracy in the final result.

(2) Bit-True Simulations. The statistics of the signals are determined via simulations, allowing the estimation of signal dynamic range requirements at each node.

Using one of these two approaches (or a mixture of them), a number of algorithms/tools have been reported to automate the conversion of a floating-point-based algorithm into a fixed-point hardware-ready implementation. A comparative study of the performance of these techniques is presented in [8]. A number of practical applications, where one (or several) optimal word lengths have been calculated using either analytical or simulation approaches have been reported in the literature. In [9, 10], the authors describe QRD-RLS algorithms in which the precision was evaluated through an iterative method to fit the application requirements and resource constraints. In [11], an optimal word-length implementation for an LMS equalizer is presented. In this case, a variable-precision multiplier is implemented such that the word length can be adapted according to different modulation schemes. In this case, not only resource constraints have to be met, but also energy (power consumption) and datapath (frequency of operation) constraints.

An alternative to defining a priori the optimum word length for a specific implementation is to have resources available with multiple word lengths. In [5], the authors present a linear algebra platform based on the use of floating point arithmetic with different formats, in an effort to exploit lower precision data formats whenever possible to reach higher performance levels. In [12, 13], the authors formulate analytical and heuristic techniques to address the scheduling, allocation, and binding of resources

under latency constraints of multiple precision resources. A common limitation of the approaches in [5, 12, 13] is an area penalty.

Analytical approaches also include methods on interval arithmetic, affine arithmetic, and Handelman representations. In interval arithmetic [14], numbers are represented by an interval of the minimum and maximum values: $[x_{\min}, x_{\max}]$. A basic problem with interval arithmetic comes from the fact that it does not capture any correlations between the variables. Affine arithmetic provides a model that can describe correlations via the use of first-degree polynomials to represent the numbers. For example, an affine form for variable $\hat{x}$ is given by [15, 16]:

$$\hat{x} = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n, \tag{3}$$

where $\epsilon_i = [-1,\ 1]$ for $i \in \{0, 1, \ldots, n\}$. The key idea here is that first-order correlations can be modeled, allowing for much tighter bounds. An example of the advantages of affine arithmetic over interval arithmetic is given by MiniBit in [16]. Even more promising is the recent use of polynomial representations due to Handelman [17]. Beyond first-order correlations, the use of polynomial representations allows for an effective model for multiplications and nonlinear functions. While the use of Handelman representation is still emerging, preliminary results in the hardware application of the conjugate gradient algorithm show a slice reduction of over 40% over traditional methods [18].

2.3. Iterative Algorithms: A Special Case. The main challenge for optimal word-length calculation in iterative algorithms is the fact that required precision is dependent on the number of iterations (loops). In [19, 20], a *precision variation curve* is defined as a sequence of pairs $\langle Li,\ Pi\rangle$, where P_i is the minimum required precision and L_i is the number of iterations. For example, the precision of a variable X after N iterations of a loop which contains the statement $X = X + Y$ is upper bounded by:

$$\text{Precision } (X) \leq \text{Precision } (Y) + \log(N+1). \tag{4}$$

Figure 1 shows how X's precision requirement increases with the number of iterations for a starting precision of 8 bits. This curve represents an upper bound for a full precision arithmetic operation.

For instance, numerical optimization algorithms are a specially complex subgroup of iterative algorithms. They require the same increase in precision as the number of iterations increases, but they can also benefit from low precision in early iterations [4, 5]. Thus, a *dynamic precision arithmetic* can improve both numerical stability (reduce quantization errors) and convergence speed. We will present a related application in the inversion of large matrices using an iterative method.

Alternatively, there is also a tradeoff between the number of iterations and the precision used in each iteration [21]. In [21], the authors show that the use of a larger number of iterations at a lower precision can yield to significant speedups over the standard practice of using double precision in a smaller number of iterations. For a model predictive

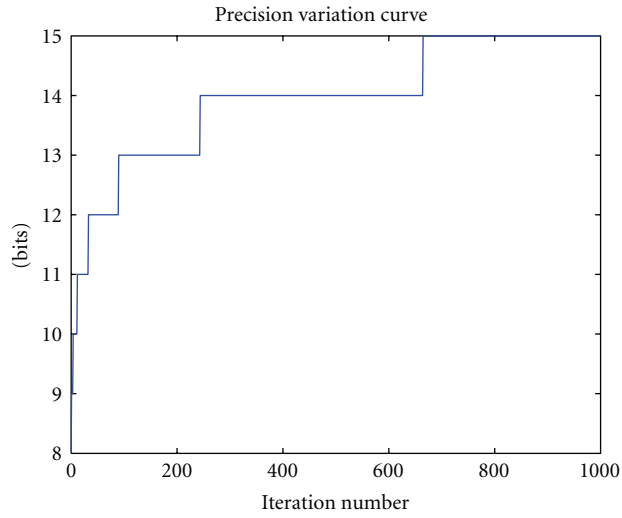

Figure 1: Precision variation curve for the accumulator in (4) [19, 20].

control application, the authors report an average speedup of 26× of a Virtex 5 implementation compared to a high-end CPU running at 3.0 GHz. The authors present a conjugate gradient method implementation for solving the generated linear system of equations. The authors also investigated the effects of a range change of the floating-point mantissa from 4 to 55 bits.

In this paper, our focus is on the solution of large linear systems using a dynamic precision that can change after each iteration. While our general framework allows changes in the number of significant bits (as in [21]), for Jacobi iterations, we show that a much simpler approach of simply changing the range of the dual-fixed-point representation can actually deliver the same accuracy as a double-precision floating-point representation.

2.4. A Case for Dynamic Precision. By dynamic precision we refer to a scheme in which a hardware implementation of an arithmetic operation changes in time to adapt its precision (change on the binary point position) according to its needs. This scheme can be accomplished by using runtime partial reconfiguration to reconfigure arithmetic modules as long as the reconfiguration time overhead is small as compared to the algorithm execution time. The reconfiguration time overhead can be decreased by reducing the amount of hardware changes required to varying precision, and by reducing the number of times precision changes are required (reconfiguration frequency). Thus, we want to consider numerical representations with small hardware footprint and with a large dynamic range.

The dual fixed-point (DFX) representation—first proposed by Ewe et al. [1]—presents both a small hardware footprint and a comparatively large dynamic range. These characteristics are also used in this paper to create an adaptation of DFX, labeled *dynamic dual fixed-point (DDFX)*. The adaptation consists of a selective reconfiguration process that significantly reduces the amount of hardware changes

required to scale in precision. This fact, combined with an improved reconfiguration rate, is used to perform arithmetic operations in an extended dynamic range at rates of operation that are comparable with other methods. This approach allows arithmetic cores with low resource requirements to be used in instances where expensive floating-point implementations were the norm.

3. Reconfigurable Computing Architectures

In 2001, Hartenstein published an exhaustive survey [22] on reconfigurable computing architectures. Complementary information can be found in [23–25]. For reconfigurable DSP architectures, surveys are available in [26, 27].

An important architecture to mention, as it is relevant to this paper, is *DISC*: the dynamic instruction set computer. DISC was first proposed by Wirthlin and Hutchings [28]. It used a medium grain, configurable logic array (CLAY) device from National Semiconductors. This computer had an instruction set made of independent hardware units that were configured into the device as needed. DISC was the first attempt to implement a truly dynamic architecture. Although in a different device family, the architecture's performance was also limited by the reconfiguration time overhead. A fundamental difference between DISC and our approach is the way we deal with the reconfiguration time overhead. DISC based its approach on a large number of resources available to add new functional units as the system required it. Only when resources were exhausted the system proceeded to replace existing modules by new ones. This sort of hardware cache reduced the impact of the reconfiguration overhead by decreasing the number of instances in which reconfiguration was required. In comparison, our approach is built upon the premise that resources are limited, requiring an existing module to be replaced at every instance.

3.1. Reconfiguration Time Overhead. The main limitation for systems considering RTR is the performance penalty paid because of the reconfiguration time overhead. This overhead is quantified by taking into consideration the bitstream size and the data transfer speed of the configuration circuitry [29–31]. SelectMap and ICAP are the external and internal parallel reconfiguration ports for Xilinx FPGAs, respectively [31]. SelectMAP provides an 8-bit or 32-bit bidirectional data bus interface to the Virtex 4 configuration logic that can be used for configuration and readback at an operation frequency of 100 MHz [32]. ICAP has 32-bit wide input and output data buses and is also set to run at a maximum frequency of 100 MHz. Thus, a maximum theoretical speed at which data can be transferred into the configuration memory using ICAP or SelectMAP is 3.2 Gb per second.

The smaller unit of reconfiguration is called a frame. In Virtex 4 devices for instance, a frame corresponds to a bit-wide column of 16 CLBs. All Virtex 4 configuration frames consist of forty-one 32-bit words resulting in a total of 1312 bits per frame. The bitstream size per each reconfigurable region in a device is calculated by the number of frames (CLB columns) it contains. Figure 2 shows the time it takes to reconfigure fully or partially (horizontal axis) different devices in the Virtex 4 family (vertical axis) assuming the maximum reconfiguration speed of to be 3.2 Gb per second.

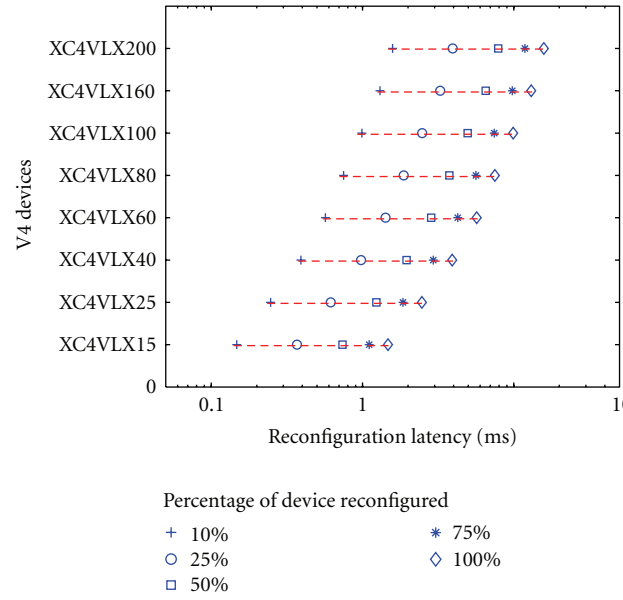

Figure 2: Virtex 4 family partial reconfiguration times. Points were calculated using the bitstream sizes reported in the device's datasheets and a theoretical maximum reconfigurable speed of 3.2 Gbits/sec [31].

4. Dynamic Dual Fixed-Point (DDFX)

4.1. Dual Fixed-Point. In this format, an n-bit number is divided into two components. The most significant bit is used to represent the exponent while the other $(n-1)$ bits are a 2's complement significant X. The exponent bit is used to indicate the radix of the signed significant X. In what follows, an n-bit DFX number whose exponent can be either p_0 or p_1 radix will be denoted as $n\_p_0\_p_1$, where $p_0 > p_1$. The value of a number is defined as:

$$D = \begin{cases} X \times 2^{-p_0} & \text{if } E = 0, \\ X \times 2^{-p_1} & \text{if } E = 1, \end{cases} \tag{5}$$

where

$$E = \begin{cases} 0 & \text{if } -\beta \le D < \beta \\ 1 & \text{if } D < -\beta \quad \text{or } D \ge \beta. \end{cases} \tag{6}$$

By choosing a boundary value, β, as the next incremental value after the maximum positive number of the radix p_0 scaling, a continuous range of numbers can be represented as shown in Figure 3. In the rest of the document, a number with radix p_0 will be said to belong to a P_0 range while a number with radix p_1 will belong to the P_1 range.

4.1.1. DDFX Adder. When performing addition of DFX numbers, both operands must be aligned with respect to the binary point. Thus, a prescaler section is required to

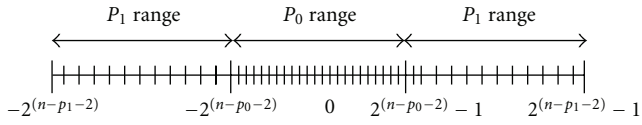

Figure 3: Range of an $n\_p_0\_p_1$ DFX number.

analyze the operands and to determine in which range—P_1 or P_0—one must perform the addition. Depending on the range in which addition is performed, one may need to scale an operand to a different range. This is done by executing an arithmetical shift to the right on the operand. A set of multiplexers are then used to select between an unchanged operand or its shifted version. Note that a truncation is performed when an operand is scaled. The bits discarded by the truncation are saved using a third multiplexer. These bits can later be recovered into the final result if the output range allows it. A block diagram is shown in Figure 4.

The prescaler produces two operands that are handed to the second stage: a full binary adder. This adder does not allow for a carry in, but it does produce an overflow signal. The adder's output is given to the third and last stage: the postscaler. In this stage, one decides in which radix the result will be presented and scales the output if needed. Scaling is again performed via shifting and multiplexers. Note that the adder described in this section can easily be transformed into a subtractor by including a 2's complement block for the subtrahend operand.

When analyzing the block diagram in Figure 4, one realizes that precision is handled only by the control blocks shown in gray. The adder itself does not deal with precision and only requires the operands' binary point to be aligned. Thus, it is possible to change the adder's precision and dynamic range by dynamically changing the control blocks. The segmentation of the functional unit in a control (dynamic) and operational (static) parts is the basic idea behind the architecture.

Since the dynamic part represents less than 40% of the whole adder, this segmentation allows us to reduce the amount of logic required to reconfigure the precision. As a consequence, the reconfiguration time overhead is also reduced.

4.1.2. DDFX Multiplier. In the case of multiplication, DFX operands do not need to be aligned with respect to the binary point before performing the operation. Thus, a prescaler section as the one in addition is not required.

A block diagram of the multiplier is shown in Figure 5. The diagram is divided in two stages. The first stage is a full precision 2's complement, binary multiplier. The second stage—a postscaler—takes the multiplier output and performs an analysis similar to the adder's postscaler case. However, in the case of the multiplier, no shifting is required. Only bit slicing is performed.

Similar to the adder, the blocks that control precision are shown in gray. The multiplier itself does not deal with precision. The blocks in Figure 5 can be rearranged in static and dynamic parts. The section of the multiplier that one needs to change in order to change the multiplier's precision represents less than 30% of the overall logic resources used by the multiplier.

4.2. Extension. A logical extension of DDFX is to also allow dynamic reconfiguration for the static parts. Although such action defeats the purpose of trying to reduce the amount of logic to reconfigure, it adds an extra degree of flexibility to the architecture by allowing a change in the operation. Since the reconfiguration time overhead is expected to be larger, these operation exchanges will have a greater impact on the architecture's performance.

5. Testing Platforms and Methodology

The systems under test (SUT) were implemented in Virtex II Pro and Virtex 4 devices with the purpose of gaining insight on the impact that the differences between both families have over reconfiguration time overhead. The general characteristics of the SUTs defined are

(i) CPU speed of 100 MHz,
(ii) CPU's cache enabled,
(iii) DDR2 memory as main memory,
(iv) UART for communication and control,
(v) access to the device's internal configuration ports (ICAP),
(vi) CPU's capacity to support user-defined coprocessors,
(vii) floating-point unit (FPU) support,
(viii) peripheral bus speed of 100 MHz, and
(ix) timer with one clock cycle resolution.

The speed of 100 MHz for the CPU and the peripheral bus was chosen to facilitate fair comparisons between the devices. Virtex 4 devices in general support a larger frequency of operation than Virtex II Pro. A more accurate comparison is feasible at lower operating frequencies since the level of effort the compiler requires to accomplish this frequency is similar for both devices. A major difference between the Virtex II Pro and the Virtex 4 SUTs is that a MicroBlaze microprocessor has been used for the Virtex II Pro while a PowerPC is used for the Virtex 4. The MicroBlaze was picked in the case of the Virtex II Pro because the PowerPC 405D5 available on these devices do not support a tight coprocessor integration as the PowerPC 405F6 on the Virtex 4 does.

5.1. Partial Bitstream Loading. In what follows, we assume that a variety of possible partial bitstreams are available to the system for reconfiguration at run time. Thus, they need to be stored somewhere accessible to the system. The solution provided by Xilinx, is to store the partial bitstreams in a compact flash memory (CF), which the system under test sees as a file system. This approach proved to be extremely slow, due to the CF high access latency. Testing shows bitstream loading speeds of about 100 Kbits/sec, way

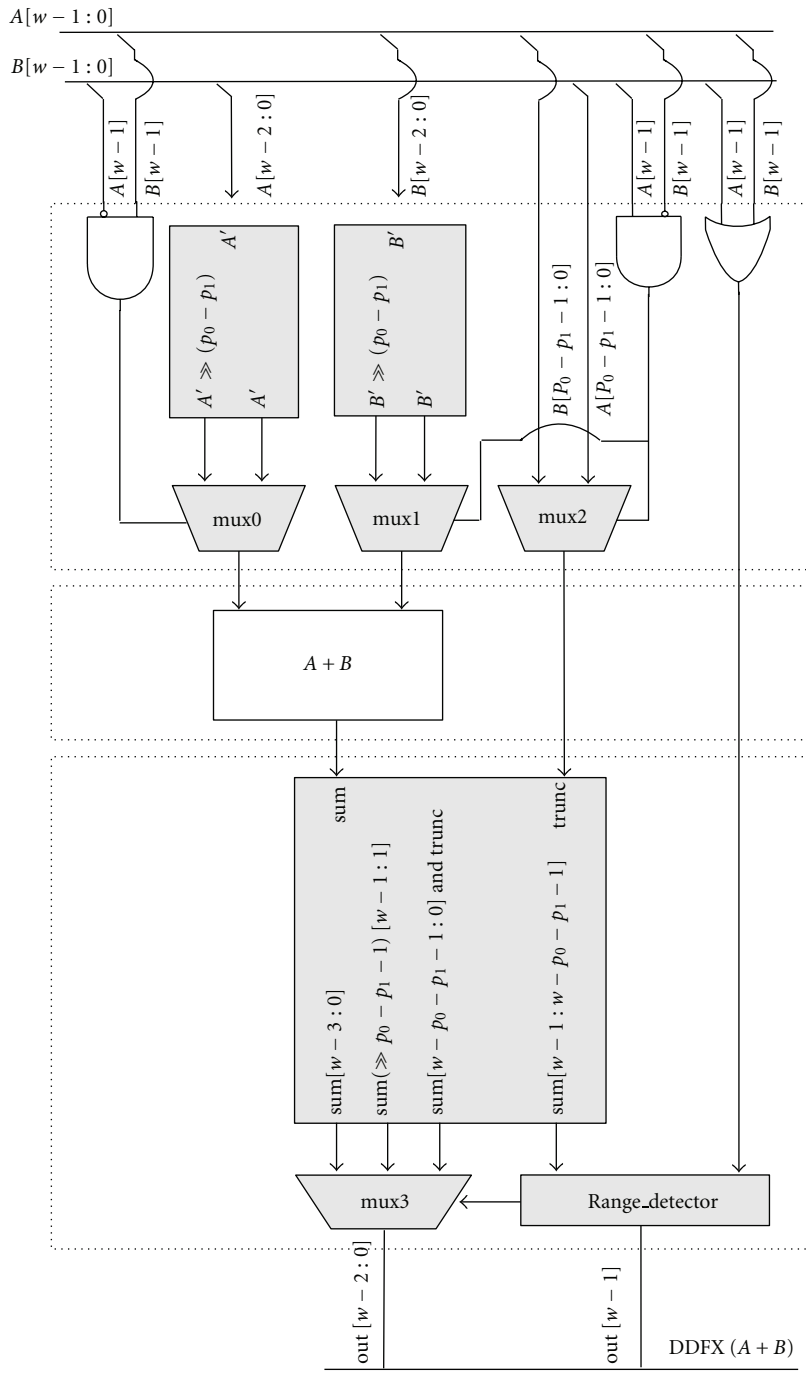

Figure 4: Dual fixed-point adder's block diagram. The three stages of the adder are depicted by the dashed lines. Expressions like $a_0 \gg n_0$ represent an arithmetical shift (with sign extension) to the right of n_0 bits for the operand a_o. The components that need to be reconfigured to enable a change in precision in a dynamic dual fixed-point adder are shown in gray.

below the maximum possible data rate of 3.2 Gbits/sec. An alternative considered was to store the partial bitstreams in a network file system. This solution, although slightly faster than the CF solution, was more costly in terms of logical resources (an ethernet supporting core is comparatively larger than a CF supporting core). Moreover, loading speed in this case would be nondeterministic and affected by the ethernet communication channel.

The preferred approach consisted of implementing a bitstream-cache sort of memory. In this approach, partial bitstreams are loaded from CF into random access memory (RAM) with the system's start-up process. Their locations in memory are stored as pointers. In this scheme, the more expensive and slower part of the partial bitstream loading is performed before any algorithm is executed. When a partial bitstream is required by an algorithm, it can be fetched

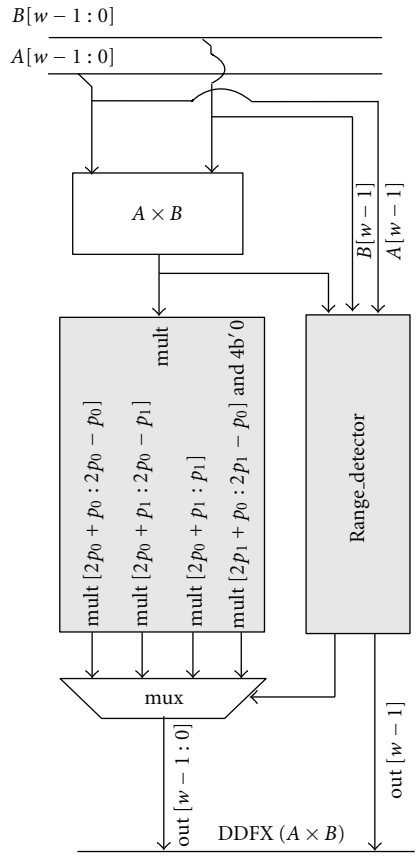

Figure 5: Dual fixed-point multiplier's block diagram. The components that need to be reconfigured to enable a change in precision in a dynamic dual fixed-point multiplier are shown in gray.

from a faster memory than the CF. This slight alteration to the standard procedure improved performance by 60 times, giving average data rates of 6 Mbits/sec.

5.2. Benchmarking the System. The test programs used to obtain the performance results in the next section targeted basic operations in linear algebra. Linear algebraic operations can be built upon a hierarchy. Dot products involve the scalar operations of addition and multiplication. Matrix-vector multiplication consists of dot products. Matrix-matrix multiplication amounts to a collection of matrix-vector products [33]. Based on these operations, more complex linear algebraic tasks can be built. Performance is measured in terms of operations per second and as a function of the *reconfiguration frequency*. Here, we define the *reconfiguration frequency* as the ratio between the number of arithmetic operations and the number of reconfigurations performed. It is straightforward to infer from this methodology that performance is directly related to the reconfiguration time overhead and thus to the reconfiguration speed and partial bitstream size.

5.3. Power Measurement. Several tools are available to estimate power consumption. For Xilinx FPGAs, we have the Xilinx power estimator (XPE) and the Xilinx power analyzer (XPA). XPE provides an estimate based on high-level resource consumption statistics and average switching activity. XPA provides a more accurate estimate based on simulated switching activity and exact utilization statistics [34]. However, the most accurate way to determine FPGA power consumption is still through hardware measurements. Moreover, there is no tool available that can provide an estimate of the power consumption during reconfiguration.

Virtex II Pro and Virtex 4 devices use three power sources commonly known as VCCINT, VCCAUX, and VCCO. Two different testing setups were implemented to measure the current for each power source. For the first setup, we measured the average current through the FPGA during operation (active power) and during standby (static power). The active power measured is equivalent to the sum of the static and dynamic power consumed by the device during operation. The measurements were taken using an ammeter as shown in Figure 6. The results of these measurements are presented in Section 6.4.3.

For the second setup, we measured the current during reconfiguration (see Figure 7). The resistor used had different values for the Virtex II Pro and the Virtex 4, but in both cases it was very small (0.1 Ohms and 0.2 Ohms, resp.). Larger resistors generate a large voltage drop and prevent the FPGA from functioning properly.

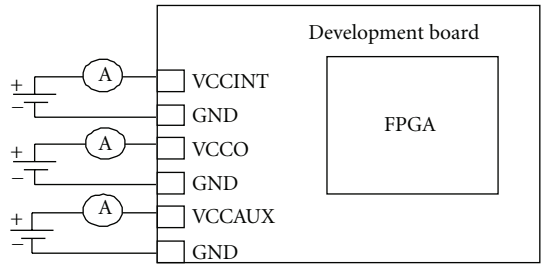

Figure 6: Measurement of average current drawn from VCCINT, VCCAUX, and VCCO.

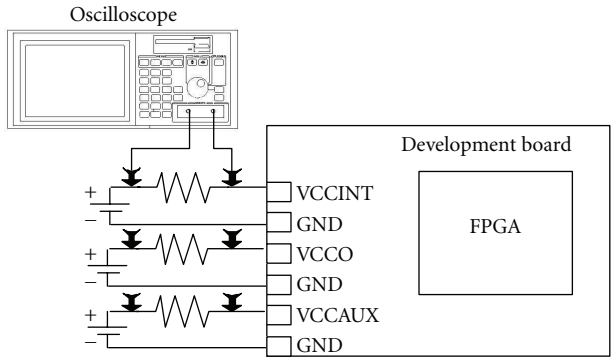

Figure 7: Measurement of current drawn during reconfiguration occurrences.

6. Results

Results are presented in terms of precision, dynamic range, resources, performance, and power consumption. Comparisons are made against hardware implementations of single-precision floating point (SFPU) and fixed-point (FX) with similar precision and fixed data width (32 bits). Results for gcc software emulation of single-precision floating point (SWFP) are also presented when appropriate. This is a special case. It is presented throughout the results not as a fair comparison, but as an example of an alternative easily available to the designer. Throughout the results in this section, the notation $n\_p_0\_p_1$, previously defined in Section 4.1, is used to specify the word length and binary point position of the different numerical representations used.

6.1. Precision and Dynamic Range. For a fixed precision and data width (32 bits), dynamic ranges vary between the hardware implementations compared. Table 1 shows dynamic range and precisions for each of the numerical formats chosen for comparisons purposes. The values were calculated using the definitions presented in Section 2.

6.2. Logical Resources. Implementation results for addition and multiplication are shown in Tables 2 and 3. Resources and maximum operation frequency values were found by compiling the cores on an XC4VFX12 device for the Virtex 4, and on an XC2VP30 device for the Virtex II Pro using ISE 9.2.4i with default settings. All but the DDFX implementation were created using the provider's optimized cores (CoreGen). In Tables 2 and 3, the term *equivalent gate count* refers to the number of 2-input NAND gates that would be required to implement the same number and type of logic functions. The number of equivalent gates cited in these tables was obtained from the ISE 9.2.4i tool after synthesis, map, place, and route had been executed.

Table 1: Precision and dynamic range results. Also see [35].

	Dynamic range	Precision
FX (32_24)	≈187 dB	$\approx 10^{-8}$
DDFX (32_24_4)	≈308 dB	$\approx 10^{-8}$
SFPU	≈1529 dB	$\approx 10^{-8}$
SWFP	≈1529 dB	$\approx 10^{-8}$

Table 2: Implementation results for Virtex 4 (XC4VFX12) using ISE 9.2.4i with default settings.

	FX (32_24)	DDFX (32_24_4)	SFPU
Addition			
Equiv. gates	1421	3508	11297
Max. freq.	≈302 MHz	≈309 MHz	≈270 MHz
Multiplication			
Equiv. gates	23290	16497	14896
Max. freq.	≈219.6 MHz	≈250.8 MHz	≈248.9 MHz

Table 3: Implementation results for Virtex II Pro (XC2VP30) using ISE 9.2.4i with default settings.

	FX (32_24)	DDFX (32_24_4)	SFPU
Addition			
Equiv. gates	1158	3514	11388
Max. freq.	≈163 MHz	≈204 MHz	≈197 MHz
Multiplication			
Equiv. gates	17802	19820	20395
Max. freq.	≈178 MHz	≈179 MHz	≈168 MHz

In terms of logical resources, for both families, a 32-bit single-precision SFPU adder is approximately equivalent to 4 DDFX or 8 fixed-point cores. Thus, in terms of resources, for the cost of one SFPU addition, we can perform 4 DDFX additions. In the case of multiplication, the DDFX footprint is smaller than SFPU by about 3% for the Virtex II Pro case. The smaller difference in size can be explained by the size of the multiplier used in every case. The single-precision SFPU requires a 24-bit multiplier (the other 8 bits are the exponent) while the fixed-point and DDFX cores require a 32-bit multiplier.

6.3. Performance. We measured the performance of the arithmetic units set as coprocessors running a series of test programs targeted towards vector operations (addition, substraction, and multiplication). The average performance is measured as

$$\text{Performance} = \frac{\#\text{Operations}}{\left(t_{\text{op}} + t_{\text{reconf}}\right)}, \tag{7}$$

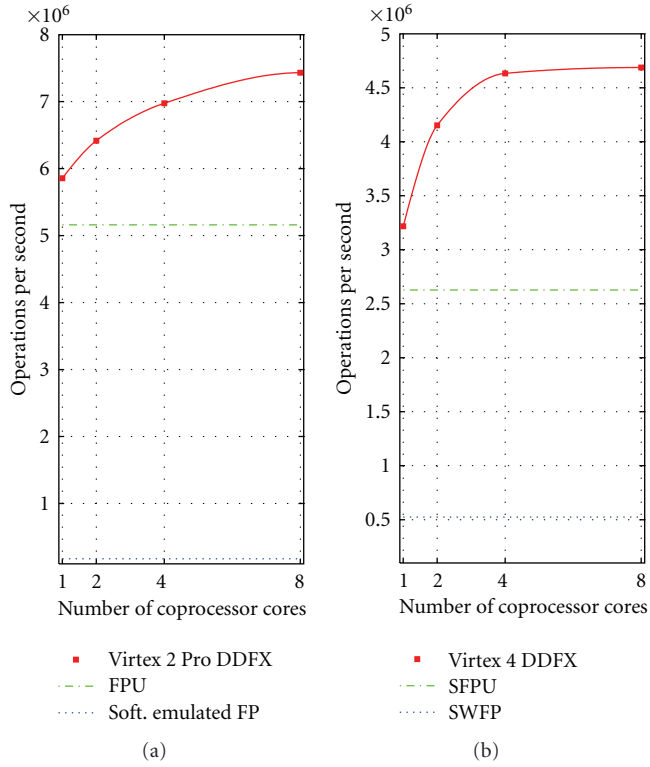

Figure 8: Multi-DDFX core performance for Virtex II Pro MicroBlaze (a) and Virtex 4 PowerPC405 (b). These measurements were taken using running hardware systems described in Section 5.

where t_{op} represents the time required to perform the arithmetic operations while t_{reconf} represents the time it takes to reconfigure the precision or the operation.

6.3.1. Static DDFX Multicore Measurements. First we measured the *static* performance or $t_{reconf} = 0$. In order to assess the setup's scalability, the measurements were taken under different load distribution schemes. First, the whole computation is performed by a single coprocessor. Then the computation is distributed among 2, 4, and 8 coprocessors. A coarse interleaving scheme is used for distributing the data among the coprocessors. Note that, according to the results on resource consumption, one SFPU core is equivalent to 4 DDFX core. Thus, a like-for-like comparison in terms of performance, keeping resources fixed, should be done between an SFPU core and 4 DDFX cores.

Figure 8 shows the performance measurements for all four setups (1, 2, 4, and 8 DDFX cores) and all three operations implemented (addition, subtraction, and multiplication) and compares them to the SFPU and software-emulated FP alternatives. In the case of DDFX, the latency for addition, subtraction, and multiplication is the same. Thus, the performance, and the curves presented in Figure 8, are the same—in each device family—for all three operations. Note that the speedup is not linear with respect to the number of cores. This is due to saturation in the FSL bus as more cores connect to it.

In these graphs, performance for SWFP and SFPU are shown as constants at their best single core's performance. In the case of SFPU, the multiplication is on average ≈5.5% slower than addition. In the case of software-emulated floating point, the difference between addition and multiplication is almost imperceptible for the PowerPC. However, in the MicroBlaze case, the difference is significant. This is mainly due to differences in the compiler (ways to emulate floating point) and the differences between the instructions available to each processor. One can also note that the MicroBlaze implementation outperforms the PowerPC. The reason for this difference lies in the level of integration between the processors and the FSL bus used to connect the coprocessors. The FSL bus was originally designed for the MicroBlaze, and it is tightly integrated with this core. In the case of the PPC, a bridge is needed for adding latency to FSL operations.

DDFX outperforms the SFPU and SWFP alternative in both families. Moreover, since an SFPU unit is equivalent to 4 DDFX cores in terms of resources (i.e., in the case of the adder), one can state that DDFX outperforms SFPU by about 40% in the Virtex II Pro case. For Virtex 4, 4 DDFX cores outperform SFPU by about 80%.

6.3.2. Reconfigurable DDFX Measurements. Next, reconfiguration is added, and the performance of a single core is measured. As mentioned before, all three operations have the same latency. Also, the bitstreams to partially reconfigure both—their precision control section and the operation itself—have the same size. This might seem counter-intuitive at first, since one expects the multiplication to require more logic resources. The bitstream sizes are the same

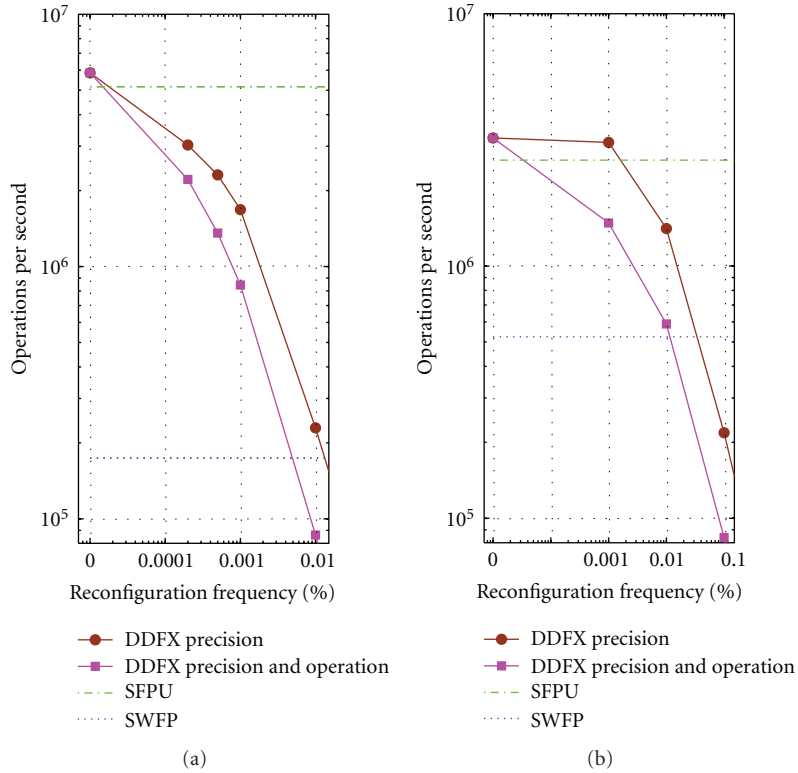

Figure 9: Performance results for a single DDFX core versus reconfiguration frequency for Virtex II Pro MicroBlaze (a) and Virtex 4 PowerPC405 (b). Here, 0% can not be represented in our logarithmic scale, thus it was included as the leftmost point in the graph. These measurements were taken using running hardware systems described in Section 5.

because the partial reconfiguration regions allocated for these sections are similar in area. This is independent of the percentage of resources used in each region. However, note that the architectural changes between the Virtex II Pro and the Virtex 4 families made possible for the areas—thus the bitstreams—to be smaller in the case of Virtex 4. This, as Figure 9 shows, represents a significant difference in the way performance drops for both families, when reconfiguration is included in the measurements. Dynamic reconfiguration can be triggered by the arithmetic cores asserting a signal that will be treated as an interrupt by the microcontroller. In our examples, reconfiguration was hardwired in the testing code to occur after a fix number of operations. This allowed us to emulate different reconfiguration frequencies.

Several observations can be made from the graph. First of all, when both, precision and the operation itself, are reconfigured the performance drops faster than when only the precision is changed. This is a straightforward consequence of the larger bitstream used when both precision and operation are reconfigured. Secondly, one can see a sharper drop in performance for the Virtex II Pro as compared to the Virtex 4. This is a direct consequence of the architectural changes that allowed smaller bitstreams on the Virtex 4. Finally, the graph shows that in the case of Virtex II Pro, the solution is outperformed by the SFPU even with very small reconfiguration frequencies.

In the case of the Virtex 4, changes of precision can be made with a frequency of 0.001% while still outperforming the SPFP alternative. To put this number in context, one could calculate a matrix vector multiplication Ax, where A is a 100×100 matrix and x is a 100×1 vector, reconfigure the precision of the operation once, and still outperform the SFPU alternative. In terms of the number of operations, 0.001% is the *maximum reconfiguration frequency* possible while still outperforming the SPFP alternative. This term is used throughout the paper as an upper bound.

Similarly to the static case, a like-for-like comparison in terms of resource consumption requires us to compare performances between one SFPU core and about 4 DDFX cores. We compared performances results where the load was distributed among 4 DDFX coprocessors and reconfiguration was performed in any of them. The results depicted in Figure 10 show that the *maximum reconfiguration frequency* has been reduced, thus more reconfiguration instances are possible while still outperforming the SFPU alternative.

6.3.3. Reconfiguration Speed. From (7), one way to improve the overall performance is by reducing t_{reconf}. Reducing t_{reconf} is a primary objective of our approach. It was accomplished by reducing the hardware footprint of the changes required to vary precision and operation, and also by improving the overall reconfiguration speed as described in Section 5.1.

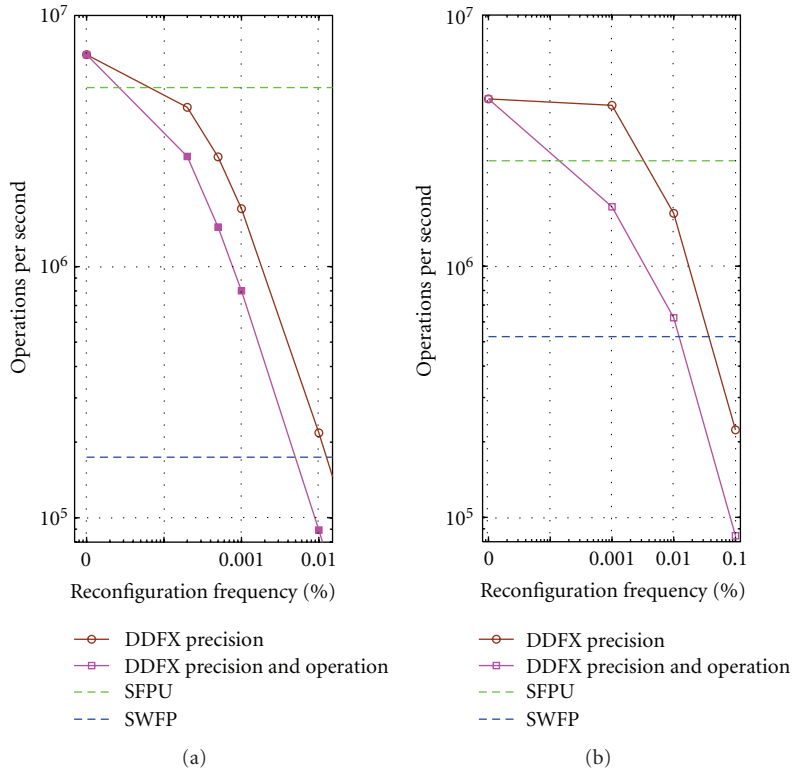

Figure 10: Performance results for 4 DDFX cores versus reconfiguration frequency for Virtex II Pro MicroBlaze (a) and Virtex 4 PowerPC405 (b). Here, 0% cannot be represented in our logarithmic scale, thus it was included as the leftmost point in the graph. These measurements were taken using running hardware systems described in Section 5.

A limitation of the reconfiguration scheme used is the sequential nature of the operations. The processor must stop what it is doing, reconfigure a functional unit, and then continue its work. The performance results presented earlier can be improved by getting the data transfer of reconfiguration memory closer to the ICAP data port bandwidth. Figure 11 models the expected performance of the proposed scheme as we improve the reconfiguration data rate.

The curves in Figure 11 were calculated for the change of precision only. As we get closer to 3.2 Gbits/sec, the size of the problems for which this solution has acceptable performance is reduced drastically. In the case of the Virtex 4, one could change precision $\approx$ one every 5000 operations and still outperform the SFPU alternative. To put this number in context, one could calculate a matrix vector multiplication Ax, where A is a 100×100 matrix, reconfigure the precision of the operation once, and still outperform the SFPU alternative. One could also use video processing applications to put this number in context. If a video is composed of frames of 640×480 pixels, a pixel-based operation over the frame (assuming a single arithmetic operation) equals to 307200 operations. This means that one could be reconfiguring more than once (actually $\approx$ 61 times) per frame, and still outperform the SFPU alternative.

6.4. Power Consumption. We consider three types of power: dynamic power, static power, and reconfiguration power. All results presented in this section were measured using the running hardware systems described in Section 5.

6.4.1. Dynamic Power Consumption. The dynamic power is due to the energy consumed by the device when it is doing useful work. Dynamic power is mainly dependent on the number of transitions in the logic gates' transistors (frequency of operation) and the gate count of the design in question. Some design decisions can help to keep this power in budget (including different number formats). Consider the performance curves shown in Figure 8 and the corresponding power consumption curves presented in Figure 12. The relationship between these curves suggests that our dynamic approach can be exploited by a system to dynamically tradeoff performance for power consumption.

The data in Figure 12 was obtained by physically measuring the current at the FPGA core during the execution of especially crafted test programs. In both graphs, the values in the Y axis are expressed as a percentage of the processor power consumption while in standby.

6.4.2. Static Power Consumption. Static power is due to the current drawn by the device after being powered up and configured but while doing nothing. This power is due to

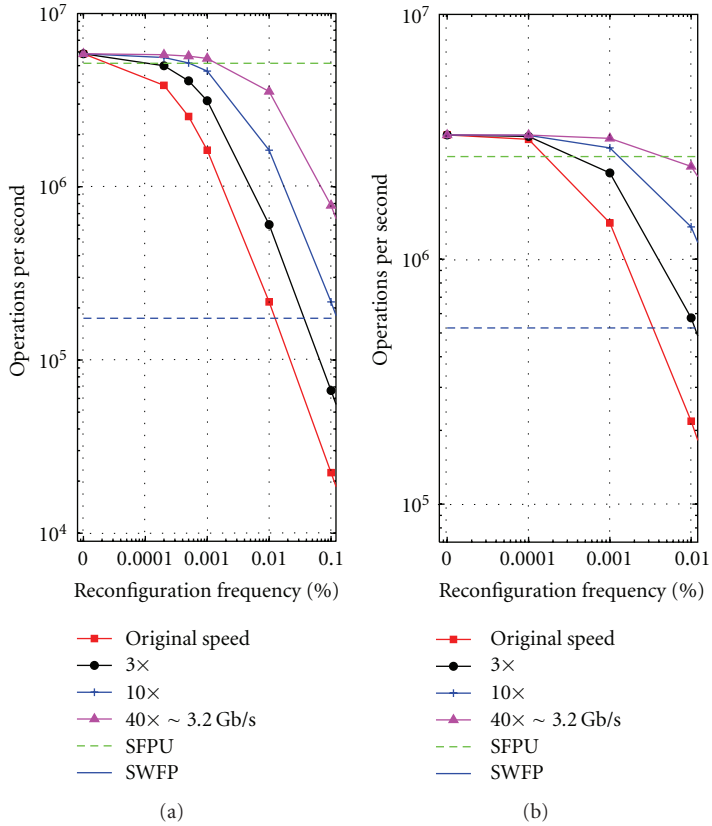

Figure 11: Performance bound curves at the theoretically maximum possible reconfiguration data rate speeds for Virtex II Pro MicroBlaze (a) and Virtex 4 PowerPC405 (b). Here, 0% cannot be represented in our logarithmic scale, thus it was included as the leftmost point in the graph. The original speed measurement (square red) was taken using running hardware systems described in Section 5. The round (black), plus (blue), and triangle (pink) curves were estimated scaling the results from the original speed measurement.

the leak current in the transistors' thin oxide layers. As the number of transistors goes up, and fabrication techniques allow for smaller geometries, this current usually increases. The increase in this current will result in heat dissipation that will in turn result in more leak current. Static power has been and still is a huge problem for the FPGA technology.

We experimented using partial reconfiguration to shut off different sections of a system at different times, without affecting the rest of it. For this purpose, shift registers of different sizes, built upon the SRL16 primitive (basically LUTs) were connected to both SUTs as customized cores. The experiment consisted of shutting off one peripheral at a time and measuring the current at the FPGA core every time. The shut-off of a core is done by reconfiguring the section that the core occupies with a "blank" bitstream—equivalent to having nothing programmed in that section.

The test results showed a linear relationship between the amount of resources used and static power consumption. Figure 13 depicts the test results interpolated with the resources used by common peripherals in an embedded system. As in the case of dynamic power, these results suggest our dynamic approach could be exploited to tradeoff performance for power consumption. The idea is extendable to any reconfigurable system. Considering the amount of time a peripheral could be idle, the savings in static power are considerable.

The overall power consumption is shown in Figure 14. This graph was obtained unifying the results of dynamic and static power consumption. A definite advantage of DDFX over SWFP and SFPU alternatives can be observed in the case of the Virtex 4 while, in the case of the Virtex II Pro, no clear advantage is observable. This discrepancy can be explained by the architectural and technology differences between both families.

6.4.3. Reconfiguration Power Consumption. The power consumed during the reconfiguration process becomes important in the context of runtime partial reconfiguration. For dynamically reconfigurable arithmetic, reconfiguration power is not dissipated only once as in standard reconfigurable systems, but many times. Thus, there is a clear relationship between the reconfiguration frequency and an increase in the mean power consumption of the overall system. Measurements of the reconfiguration power consumption for both the Virtex II Pro and Virtex 4 are presented in Section 6.4.3.

Figures 15 and 16 illustrate the changes in current drawn for the Virtex II Pro case. The curves shown in these

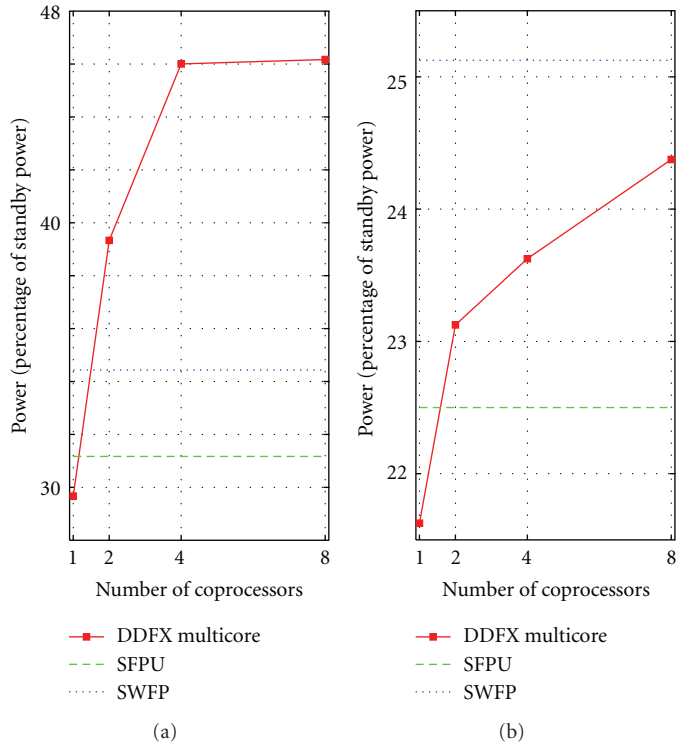

Figure 12: Dynamic power consumption for Virtex II Pro (a) and Virtex 4 (b). These measurements were taken using running hardware systems described in Section 5.

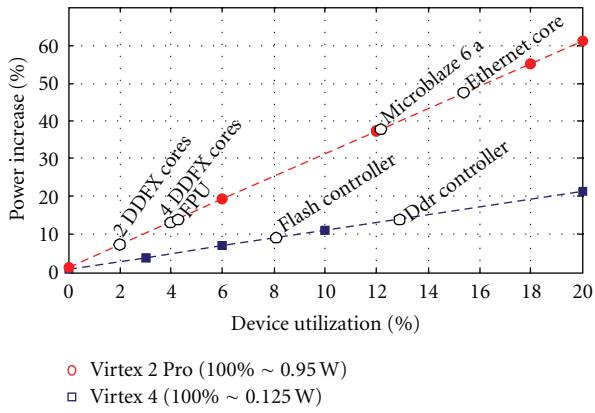

Figure 13: Virtex II Pro and Virtex 4 static power consumption and interpolated power consumption of commonly used peripherals in embedded systems. The red circles and blue squares represent data points obtained through direct measurements following the setup in Section 5.3.

figures correspond to the voltage across the shunt resistor (0.1 Ohms) used to measure the current (see Figure 7).

In Figure 15, the first increment in the voltage curve corresponds to the start of the FPGA's (full) configuration. This process lasts approximately 3 seconds, and it is followed by a second increment corresponding to the FPGA's operating state after configuration. Using Ohms' law, we can estimate that the FPGA draws approximately 15.2 mV/0.1 Ω = 152 mA during its configuration process.

Figure 16 shows the case of a partial reconfiguration instance. The voltage peak lasts about 75 μsec, which correlates with the time reported by the test application. In this case, the peak voltage is about 20 mV, thus the current increased by 200 mA. The reason behind the slight difference between the measurement for a full and a partial reconfiguration is not clear. Unfortunately, there is very little information from the manufacturer as to how exactly this power is used. We will assume, however, that partial reconfiguration in the case of Virtex II Pro requires an extra 200 mA for the duration of the reconfiguration operation. This represents ≈20% of the overall active power consumption. If a system were constantly reconfiguring (which is an unrealistic worst-case scenario), an increase of up to 20% in the mean power consumption can be expected. Reconfiguring precision takes ≈9 msec for the Virtex II Pro's case. Thus, the amount of energy consumed by the precision change is $\approx 2.5 * 0.20 * 9E^{-3}$ sec $= 4.5E^{-3}$ Joules.

Similarly, we found that, in the case of the Virtex 4, the configuration process draws an additional 25 mA. This represents ≈2.5% of the overall active power consumption. If a system were constantly reconfiguring, an increase of up to 2.5% in the mean power consumption can be expected, an increase which is an order of magnitude smaller than in the case of the Virtex II Pro. Reconfiguring precision

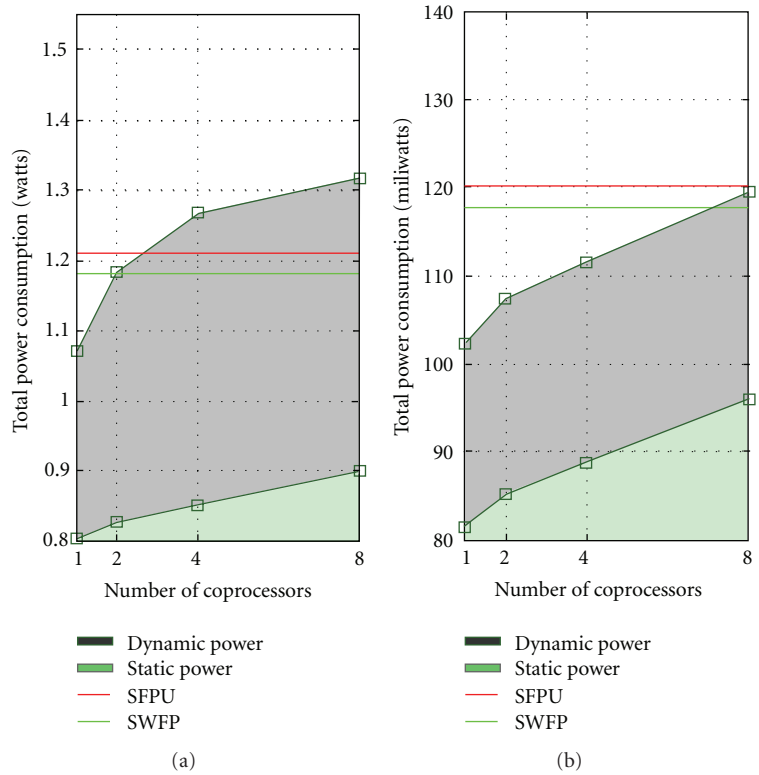

Figure 14: Virtex II Pro (a) and Virtex 4 (b) total power consumption (dynamic + static).

takes ≈4 msec for the Virtex 4's case Thus, the amount of energy consumed by the precision change is $\approx 2.5 * 0.005 * 4E^{-3}$ sec $= 50E^{-6}$ Joules.

7. Large Matrix Inversion Using DDFX

We now consider an application of DDFX in large matrix inversion. The objective is to demonstrate the effectiveness of the approach in moderately large matrices of size 100×100, as discussed in our reconfiguration results. The effectiveness is demonstrated by comparing the DDFX approach to static DFX and double-precision floating-point arithmetic, the numerical representation that is routinely used in such applications. We also want to use this application to demonstrate how a simple scheme for controlling DDFX precision can be used to give effective results in a complex application in numerical linear algebra. Our focus continues to be on iterative systems. In addition to being of interest to DDFX applications, iterative systems also represent a promising framework for developing energy-scalable systems. Since the total energy consumption increases (almost) linearly while the error magnitude gets reduced with the number of iterations, one can device energy scalable solutions by controlling the number of iterations.

Let $Ax = b$ represent a large system of linear equations in the variable vector x. We assume that A is of size $N \times N$, x is of size $N \times 1$, and b is of size $N \times 1$. For full matrices, direct inversion using $A = LU$ requires approximately $2/3N^3$ multiply and add operations [36, page 33]. When n is large, say $n \geq 100$, we are led to consider iterative methods. To setup an iterative approach, we begin by splitting A into $A = S - T$ [37]. Here, for the Jacobi method, we set S to the diagonal entries of A. We then have $(S - T)x = b$. This leads to the iterative system [37]:

$$Sx_{k+1} = Tx_k + b, \tag{8}$$

where k denotes the iteration number. The basic idea is that (8) will be very easy to compute, and that x_k will converge to the true solution. It is then easy to show that the error satisfies $e_{k+1} = S^{-1}Te_k$. The error vector is then given by ([37]):

$$e_k = c_1\lambda_1^k x_1 + \cdots + c_n\lambda_n^k x_N, \tag{9}$$

where $\lambda_1, \ldots, \lambda_N$ denote the eigenvalues of $S^{-1}T$ and $x_1, \ldots, x_N$ denote the eigenvectors $S^{-1}T$. The method will converge as long as the largest eigenvalue satisfies $|\lambda_{\max}| < 1$. With each iteration, the error magnitude gets reduced by at-least $|\lambda_{\max}|$.

For DDFX, we use

$$x_{k+1} = (S^{-1}T)x_k + (S^{-1}b), \tag{10}$$

where $(S^{-1}T)$ and $(S^{-1}b)$ are precomputed. Since S is diagonal, inversion only requires N divisions. For the precomputation overhead, we will assume that divisions are implemented by multiplication by the inverse elements. Note that this precomputation overhead is dominated by the N^2 multiplications and $(N - 1)N$ additions required for $S^{-1}T$.

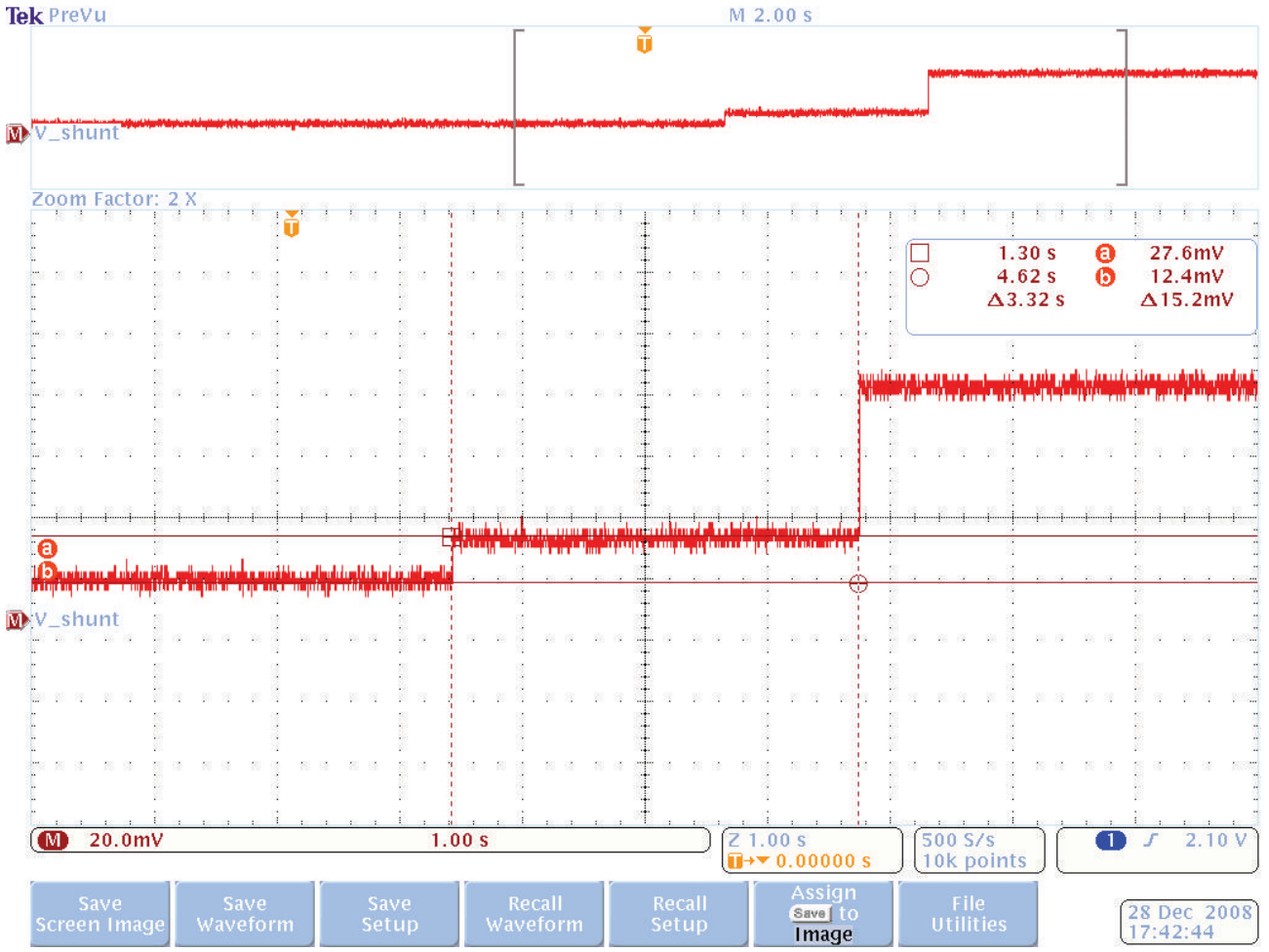

FIGURE 15: Power consumption for Virtex II Pro's full reconfiguration. The Y axis is voltage measured at the shunt resistors during the reconfiguration process according to setup shown in Figure 7. This figure is an oscilloscope screenshot.

For each step of (10), we have $N^2 + N$ multiplications and $(N-1)N+N-1+N = N^2+N-1$ additions. Thus, in order for our iterative scheme to be better than the direct method, we must achieve satisfactory convergence for iterations that are significantly less than N.

Based on our results in Section 6.3.2, for an effective implementation, we must only allow a dynamic reconfiguration after a large number of arithmetic operations. We consider a dynamic reconfiguration after each iteration. From (10), we see that each iteration requires N^2 multiplications and $(N-1)N+N = N^2$ additions. In particular, for $N = 100$, for matrices of size 100×100, we have 10,100 multiplications and 10,099 additions. This exceeds our requirements for the minimum number of operations outlined in Section 6.3.2. Furthermore, it is clear that the dynamic reconfiguration bounds can always be met for sufficiently large matrices.

Similar to our accumulation example in Section 2.3, we expect that the precision requirements for x_k will grow. Ideally, after each iteration is completed, we want to adjust the precision of x_k, so as to maintain the maximum possible precision. On the other hand, the static matrices in (10) are best kept at a constant precision. This approach avoids the costly numerical conversion of a matrix. Here, we note that the type conversion for x_k is limited to N vector elements.

In what follows, we first assume that the total number of bits n is kept fixed. We initialize the iterations by setting p_0 to the maximum number of bits required for storing $S^{-1}T$ and the vector $S^{-1}b$ (see Figure 3). We then set $p_1 = p_0 - 1$ to satisfy our requirement for $p_0 > p_1$. We also initialize $x_0 = 0$, the zero vector. Our approach will thus allocate the low numerical range to $S^{-1}T$ and $S^{-1}b$ (controlled by p_0) while the larger values of x_k will be controlled by p_1. Based on this setup, we want our numerical range to only grow as required by x_k (also see precision variation curve in Figure 1).

We estimate the precision growth in x_k using (10). We recognize the addition of two terms: one coming from the multiplication and a second one coming from adding $(S^{-1}b)$. Since the absolute values of the eigenvalues of $S^{-1}T$ are always less than 1, we do know that the product norm will be less than x_k. This motivates the use of the maximum absolute element of x_k as an upper bound estimate of the maximum absolute value of the product term. Let $\text{Bits}(\max_k|x_k|)$ denote the maximum number of

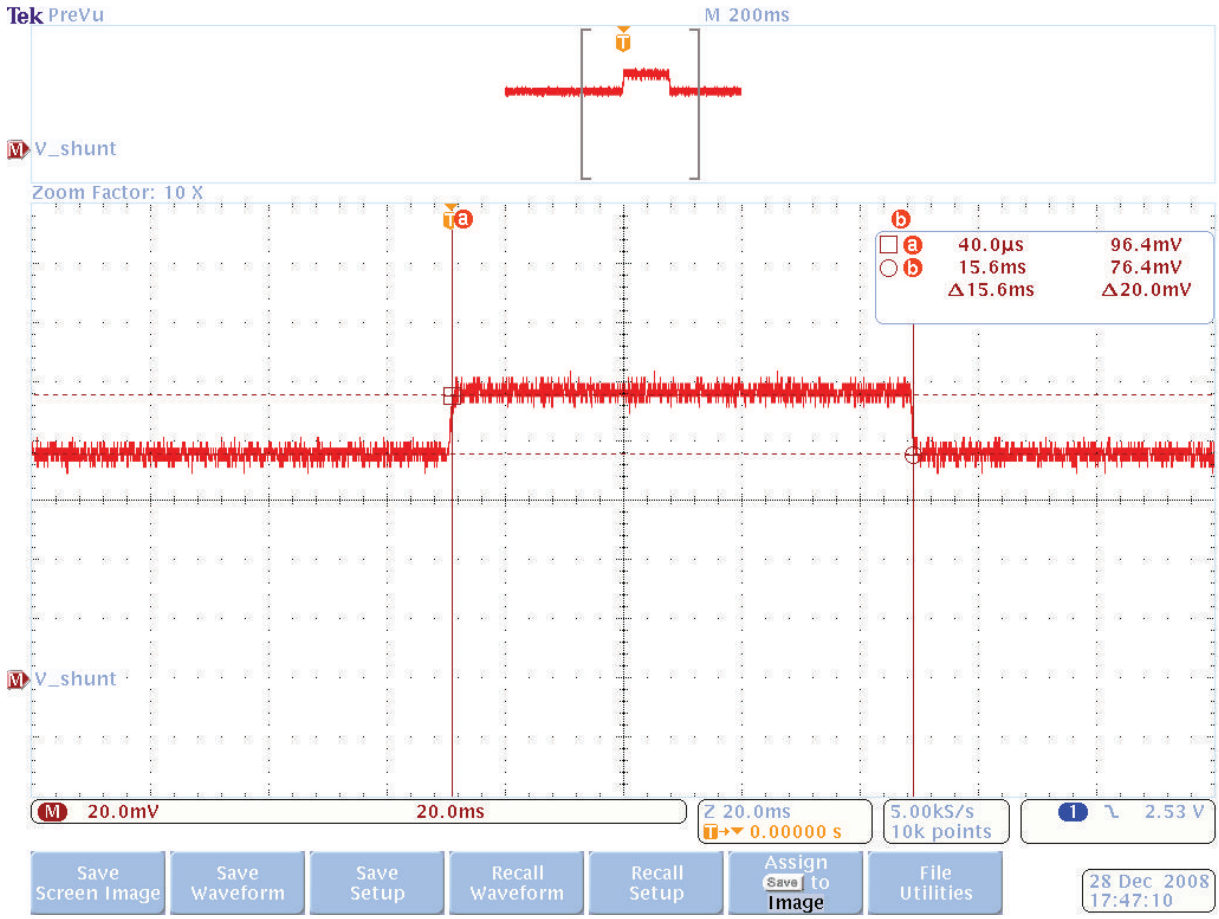

FIGURE 16: Power consumption for Virtex II Pro's partial reconfiguration. The Y axis is voltage measured at the shunt resistors during the reconfiguration process according to setup shown in Figure 7. This figure is an oscilloscope screenshot.

bits required for representing x_k. Similarly, let $\text{Bits}(S^{-1}b)$ represent the precomputed required number of bits for representing the second term. The addition will then be bounded above by $1 + \max(\text{Bits}(x_k), \text{Bits}(S^{-1}b))$. To see this, note that the addition of two p-bit numbers may produce a $(p + 1)$-bit number. Our discussion then leads to the following simple rule for computing the precision for x_k:

```
intBits = Bits (max(abs(x_k)))
p1n = n-2-max(intBits, AddBits)
if (p1n < p1)
    p1 = p1n
end
```

where `AddBits` represents the precomputed value of $\text{Bits}(S^{-1}b)$.

We present a numerical example in Figure 17. Here, A is set to a normalized random correlation matrix of 100×100 elements generated as described in [38] (also implemented in MATLAB with the `gallery` function). The inversion of correlation matrices is routinely used in minimum mean square error estimation (MMSE; see page 302 in [39]). We let b be a unit vector. We initialize DDFX using $n = 32$. Based on the minimum numbers of bits required for $S^{-1}T$ and $S^{-1}b$, we require $p_1 = 29$. We initialize p_0 to $p_1 - 1$. In this example, the maximum eigenvalue was relatively high at 0.93.

The simulation results of Figure 17 demonstrate the success of the approach. The actual solution was estimated using the pseudoinverse (computed using singular value decomposition) in double arithmetic. In our example, we compute Euclidean distances from x_k to the actual solution. In other words, we report on the root mean square (RMS) of the error between x_k and the actual solution. We also report distances in dB using $20\log_{10}(\text{RMS}_{\text{solution}}/\text{RMS}_{\text{error}})$. Similarly, we report the distance between the double-precision floating-point arithmetic and the dynamic dual fixed-point iterate in Figure 17(d). The dynamic precision reconfiguration is shown in Figure 17(e).

Firstly, from Figure 17(b), we can see that the dynamic dual fixed-point and the double-precision floating-point arithmetic iterations converge to the correct solution. The dynamic dual fixed-point is very close to the double-precision floating-point estimate as shown in Figure 17(d). As shown in Figure 17(a), the static dual fixed-point fails to converge. The dynamic dual fixed-point has switched between 4 representations as shown in Figure 17(e). It is

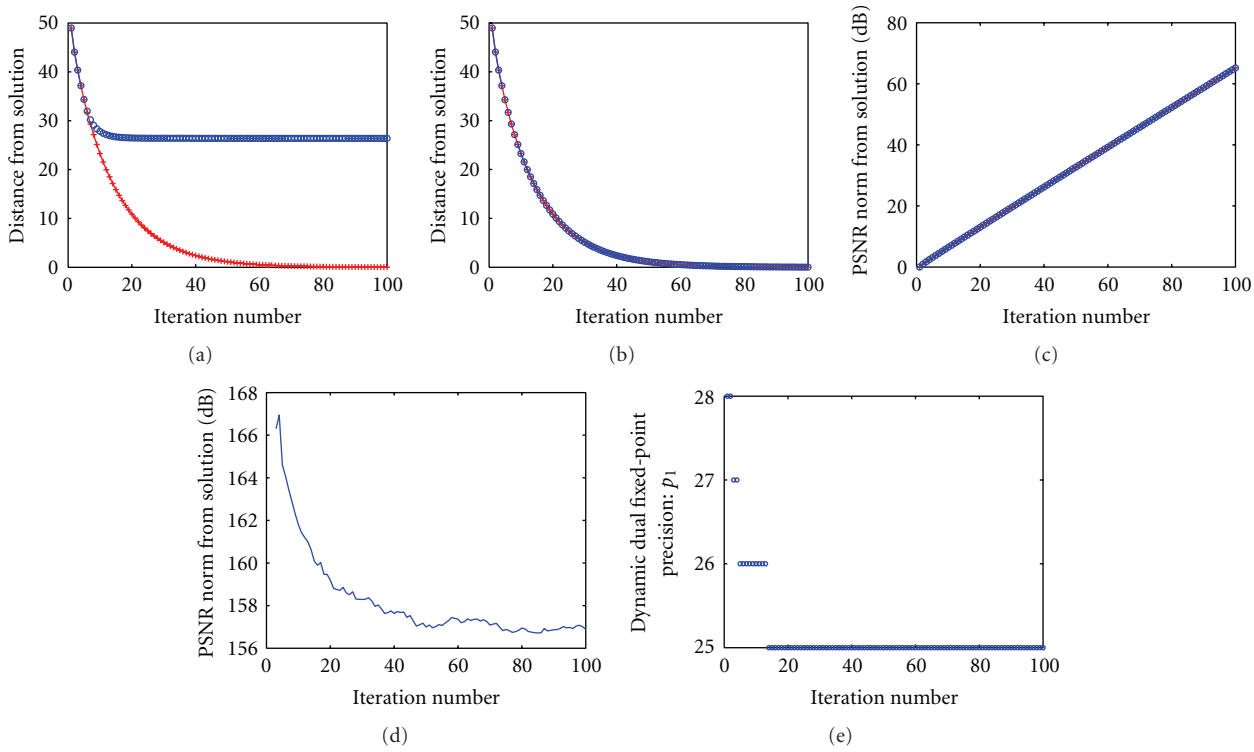

Figure 17: Comparison of DDFX to DFX and double-precision floating-point solutions. (a) Euclidean distance of the DFX (plotted with blue circles) and double-precision floating point (plotted with red "+") from the actual solution as a function of the iteration number. Note the lack of convergence of the standard DFX method. (b) Euclidean distance of DDFX (plotted with blue circles) and double-precision floating point (plotted with red "+") from the actual solution as a function of the iteration number. The DDFX approach converges at the same rate as double-precision floating point. (c) Replot of (b) where Euclidean distance from the solution is measured in dB. The double-precision floating-point and DDFX solutions are indistinguishable in this graph. The blue circles of DDFX overlap with the red "+" of double-precision floating point. (d) PSNR of the distance from the DDFX iterate to the double-precision floating-point iterate. The high PSNR values indicate that the two solutions are virtually identical. (e) Dynamic adjustment of p_1 as a function of the iteration ($n = 32$, $p_0 = 29$). These results were obtained through simulations.

clear from this example that DDFX can achieve convergence to the correct solution using significantly less resources and total energy consumption than double-precision floating-point arithmetic.

8. Conclusion

A dynamic arithmetic architecture using runtime partial reconfiguration was presented and evaluated in terms of numerical precision, dynamic range, logical resources, performance, and power consumption. For the newer Virtex 4 FPGA devices, our testing results show significant advantages of the proposed approach over alternative approaches.

For the same data width and for approximately equivalent precisions, DDFX's dynamic range and logical resources lie between fixed-point and floating point. DDFX exhibits a footprint up to 75% smaller with respect to floating point in the case of addition/subtraction. Keeping logical resources and precision equivalent, DDFX shows a performance advantage in excess of 50% with respect to alternative floating point and software-emulated floating point when no reconfiguration is performed. The maximum reconfiguration frequency, while keeping performance comparable to a floating point unit, is 0.001% (in the case of the Virtex 4). These results were obtained running actual hardware implementations of the proposed architecture. We also presented simulation results of a theoretical system where higher (but possible) reconfiguration speeds are used. These results show that, at higher reconfiguration speeds, one could reconfigure once every 5000 operations while still being as fast as a floating-point unit.

For equivalent logical resources, precision, and performance (equivalent upper bounds), the proposed solution consumes 10% less power (in the case of the Virtex 4) than its alternatives. We also showed an approach that can produce power savings by dynamically managing the number of processing cores, effectively trading off performance by power consumption. Our measurements show that the savings in overall energy consumption are significant. The approach is extensible to more general embedded systems, where a dynamic reconfiguration scheme can be used to shutdown unused cores. We have also quantified the power consumed during reconfiguration, showing that it is not significant. Finally, we provide a simulation example where DDFX was shown to closely approximate results obtained with double-precision floating-point arithmetic. This suggests that DDFX

provides for a promising platform for large numerical analysis problems where floating point is believed to be the only alternative.

Acknowledgment

The research presented in this paper has been funded by the Air Force Research Laboratory under Grant number QA9453-06-C-0211.

References

[1] C. T. Ewe, P. Y.K. Cheung, and G. A. Constantinides, "Dual fixed-point: an efficient alternative to floating-point computation," in *Proceedings of International Conference on Field Programmable Logic*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 200–208, 2004.

[2] N. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, Pa, USA, 2nd edition, 2002.

[3] G. Constantinides, P. Cheung, and W. Luk, *Synthesis and Optimization of DSP Algorithms*, Kluwer Academic Publishers, Norwell, Mass, USA, 2004.

[4] R. Dembo, S. Eisenstat, and T. Steihaug, "Inexact newton methods," *SIAM Journal on Numerical Analysis*, vol. 19, pp. 400–408, 1982.

[5] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance mixed-precision linear solver for FPGAs," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1614–1623, 2008.

[6] G. A. Constantinides and G. J. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters*, vol. 15, no. 2, pp. 137–140, 2002.

[7] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 886–896, 2005.

[8] M. A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 612–615, May 2002.

[9] Altera, "Implementation of CORDIC-Based QRD-RLS Algorithm on Altera Stratix FPGA with Embedded Nios Soft Processor Technology," March 2004.

[10] R. Uribe and T. Cesear, "Implementing matrix inversions in fixed-point hardware," *DSP Magazine*, October 2005.

[11] W. Ling and Y. Savaria, "Variable-precision multiplier for equalizer with adaptive modulation," in *Proceedings of the 47th Midwest Symposium on Circuits and Systems*, vol. 1, pp. 553–556, July 2004.

[12] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum and heuristic synthesis of multiple word-length architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 39–57, 2005.

[13] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432–1442, 2003.

[14] R. Moore, *Interval Analysis*, Prentice Hall, Englewood Cliffs, NJ, USA, 1966.

[15] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: concepts and applications," *Numerical Algorithms*, vol. 37, no. 1–4, pp. 147–158, 2004.

[16] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–1999, 2006.

[17] D. Handelman, "Representing polynomials by positive linear functions on compact convex polyhedra," *Pacific Journal of Mathematics*, vol. 132, no. 1, pp. 35–62, 1988.

[18] D. Boland and G. A. Constantinides, "Automated precision analysis: a polynomial algebraic approach," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 157–164, May 2010.

[19] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1201–1217, 2002.

[20] K. Bondalapati and V. K. Prasanna, "Dynamic precision management for loop computations on reconfigurable architectures," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 249–259, April 1999.

[21] A. R. Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control," in *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 209–216, April 2009.

[22] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*, pp. 642–649, Munich, Germany, 2001.

[23] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[24] B. Radunović and V. Milutinović, "A survey of reconfigurable computing architectures," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL '98)*, vol. 1482 of *Lecture Notes in Computer Science*, pp. 376–385, 1998.

[25] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings on Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[26] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: a survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1-2, pp. 7–27, 2001.

[27] A. Shoa and S. Shirani, "Run-time reconfigurable systems for digital signal processing applications: a survey," *Journal of VLSI Signal Processing*, vol. 39, no. 3, pp. 213–235, 2005.

[28] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99–107, Napa Valley, Calif, USA, April 1995.

[29] A. Dehon, *Reconfigurable architectures for general-purpose computing*, Ph.D. thesis, Massachusetts Institute of Technology Artificial Intelligence Laboratory, October 1996.

[30] J. Resano, D. Mozos, F. Catthoor, and D. Verkest, "A reconfiguration manager for dynamically reconfigurable hardware," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 452–460, 2005.

[31] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS,"

in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 12–17, August 2006.

[32] Xilinx, "Virtex-4 Configuration Guide (UG071)," 2006.

[33] G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins, Baltimore, Md, USA, 1996.

[34] T. Tuan and S. Trimberger, "The power of FPGA architectures: the present and future of low-power FPGA design," *Xcell Journal*, 2007.

[35] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides, "Error modelling of dual fixed-point arithmetic and its application in field programmable logic," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 124–129, August 2005.

[36] G. Strang, *Computational Science and Engineering*, Wellesley-Cambridge Press, Wellesley, Mass, USA, 2007.

[37] G. Strang, *Linear Algebra and Its Applications*, Harcourt Brace Jovanovich, San Diego, Calif, USA, 3rd edition, 1988.

[38] P. I. Davies and N. J. Higham, "Numerically stable generation of correlation matrices and their factors," *BIT Numerical Mathematics*, vol. 40, no. 4, pp. 640–651, 2000.

[39] H. Stark and J. Woods, *Probability, Random Process, and Estimation Theory for Engineers*, Prentice Hall, Englewood Cliffs, NJ, USA, 2nd edition, 1994.