EURASIP Journal on Applied Signal Processing 2005:7, 1093–1109

Prototyping Neuroadaptive Smart Antenna for 3G Wireless Communications

William To
Department of Electrical and Computer Engineering, The University of Auckland, Private Bag 92019, Auckland, New Zealand
Email: wto001@ctimail.com

Zoran Salcic
Department of Electrical and Computer Engineering, The University of Auckland, Private Bag 92019, Auckland, New Zealand
Email: z.salcic@auckland.ac.nz

Sing Kiong Nguang
Department of Electrical and Computer Engineering, The University of Auckland, Private Bag 92019, Auckland, New Zealand
Email: sk.nguang@auckland.ac.nz

Received 13 April 2004; Recommended for Publication by Alexei Gorokhov

This paper describes prototyping of a neuroadaptive smart antenna beamforming algorithm using hardware-software implemented RBF neural network and FPGA system-on-programmable-chip (SoPC) approach. The aim is to implement the adaptive beamforming unit in a combination of hardware and software by estimating its performance against the fixed real-time constraint based on IMT-2000 family of 3G cellular communication standards.

Keywords and phrases: RBF neural network, smart antenna, prototyping, FPGA.

1. INTRODUCTION

The widespread use of cellular communications systems is evident for the past two decades. Third-generation (3G) cellular systems, promising high-speed data transfer for multimedia content and improved voice communications, have already been deployed in Japan by DoCoMo in the Tokyo area experimentally in June 2001 and commercially in October 2001 [1, 2]. Other 3G systems with different protocols under the IMT-2000 banner are currently under extensive developmental testing before their respective commercial deployments throughout the world. In the IMT-2000 family of 3G cellular communications standards, the TD-SCDMA protocol from CATT explicitly requires smart antennas as a *fundamental but not essential* requirement [3], while CDMA2000 and WCDMA implicitly require smart antennas or advanced beamforming techniques to enhance their performance [4].

In radio systems, smart antenna is initially drawn up to combat electronic countermeasures in military communications and radar systems [5]. Its aim is to maximize signal-to-noise ratio between the sender and receiver by adaptive adjustments to output beam patterns with steerable mechanical/electronic antennas arrays. In cellular communications, smart antenna techniques maximize antenna gain and signal-to-noise ratio (SNR) at desired directions, and place nulls at the interference sources. Cellular receiver units move constantly and smart antennas must consider user tracking for genuine adaptive beamforming which requires real-time user tracking and beamforming algorithms. Different researchers have devised algorithms for optimum beamforming for smart antennas. These algorithms are broadly separated into [4, 5, 6]

(i) switched-Beam approximation;
(ii) statistically optimal beamforming;
(iii) direction-of-arrival (DoA) beamforming.

The biggest advantage is that it can achieve *spatial division multiple access* (SDMA) or spatial separation between users and allow them to share the same frequency bands or time slots across cells [4], which increases system capacity through higher spectral resources reuse. Smart antennas extend transmission range with identical transmission power [6]. If effective power is increased by directing maximum antenna gain with optimal antenna pattern, then available range is extended with higher effective tolerable path loss. Directional beamforming also provides two key advantages over fixed antennas: improved building penetration and hole filling in coverage area [4]. Range extension and link quality improvements are provided by managing or even exploiting multipath fading and time dispersion in realistic radio channels.

An associated advantage with increased effective transmission power is a reduction of actual transmission power for identical area/range coverage. During initial rollout, smart antennas save capital costs by directed wide area coverage. The operator can decrease transmission power at individual base stations to cover a smaller area and deploying more base stations to support more simultaneous subscribers when their numbers increase. The precise power control essential in CDMA systems for equal power transmission for arriving signals could be alleviated with smart antennas by isolating uplink signals for different users.

Many digital smart antenna algorithms involve matrices to represent multiple signals. They become computationally intensive to the point that real-time realizations are extremely difficult. A direction for real-time implementations is to simplify and approximate computationally intensive operations with simpler operations. By combining smart antennas and neural networks, new possibilities can be opened for genuine SDMA in cellular communications systems.

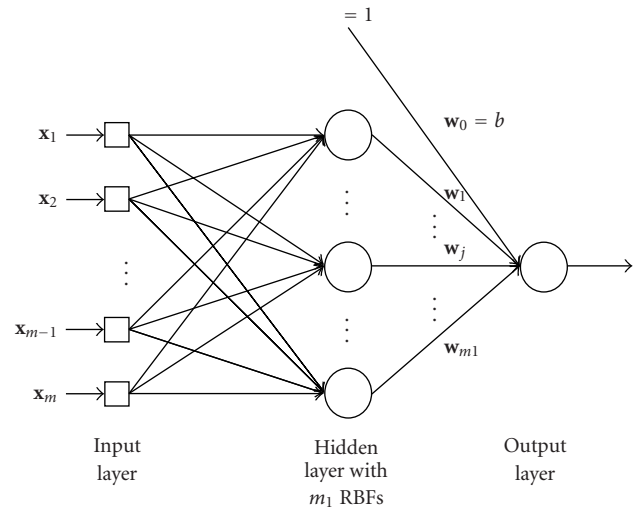


Figure 1: RBF neural network topology.

2. NEUROADAPTIVE BEAMFORMING USING RBFNNs

Artificial neural networks (ANNs) mathematically model the human neuron initially to realize true artificial intelligence (AI). Currently ANNs are mainly used for solving nonlinear problems, complex function approximation problems, or classification/recognition problems [7, 8] that algorithmic computing proved unrealistic to solve. A neural network is a universal function approximator to estimate function output in a specific range by interpolation using hidden-layer weights, which are representations of the input-output relationship inside the expected range. Generalization is defined as interpolation between known sample pairs. If the unseen input vector falls outside of the expected range, it returns an output that extrapolates dependent on output-layer linearity. Neural network is ideal for approximating complex algorithms with inherent parallelism for high performance and strong numerical approximation capabilities with simple arithmetic operations.

Figure 1 shows a generic RBF network with single output from m-length input vector and $m1$ RBF function hidden-layer neurons. The RBF neural network (RBFNN) is a single-layer feed-forward back-propagation neural network [8, 9] based on multidimensional curve fitting or interpolation problem where output values for unseen inputs are estimated in terms of the cluster centers in the training data samples. The major differences between multilayer perceptron neural network (MLPNN) and RBFNN are that RBFNN only has one hidden-layer and uses different hidden-layer activation functions. The MLPNN uses the sigmoid activation function as shown in Figure 2 while RBFNN uses a group of functions called radial basis functions also shown in Figure 2.

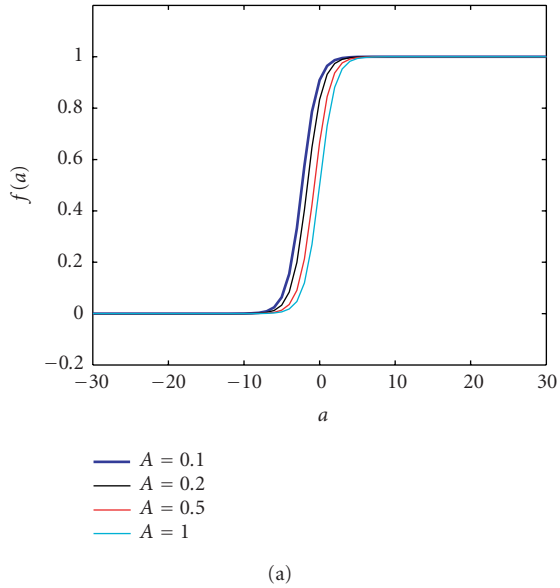
RBF maps input values into a highly nonlinear multidimensional hypersurface for solving function approximation and interpolation problems according to the theorem of separability of patterns proposed by Cover [7]. The input to RBF is a distance measure between input vector and hidden-layer weight vectors. A bias or data-independent variable is added to network output to serve as a form of y-intercept in function interpolation and approximation:

$$F^*(x) = \sum_{i=1}^{m} w_i G\left(\left\|\mathbf{x} - \mathbf{t}_i\right\|\right)_C + b. \tag{1}$$

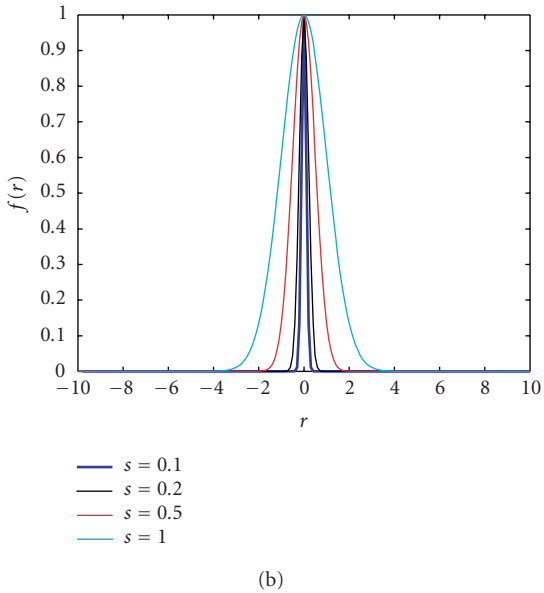
The relationship between input and output in any neural network in terms of weight interconnections, including the RBFNN shown in Figure 1, can be succinctly represented in matrix notation as shown in (1). G is a generalized RBF, $\mathbf{x}$ is input vector, w_i is output weights, and $\|\mathbf{x} - \mathbf{t}_i\|$ is Euclidean distance between input vector and hidden-layer weight vector. The hidden-layer weight vector set $\mathbf{t}_i$ is within the function subspace; therefore generalization will consistently recover a close estimate of the original function if input is within the trained range. The output-layer operation is a linear summation of product between hidden-layer outputs and output-layer weights. A detailed mathematical justification for RBF neural networks could be found in [7].

The interconnected neural network structure is mathematically equivalent to matrix arithmetic with matrix notation for hidden- and output-layer neurons. Depending on network design, each column or row vector represents a hidden-layer neuron and its weights, with the vector length equal to the input vector. During network evaluation of unseen inputs, the input value vector is transformed into a set of distances between the input vector and all stored prototypes.

El Zooghby et al. published detailed descriptions and results for adaptive beamforming algorithm to approximate Wiener beamforming weight evaluation algorithm using RBF neural networks for cellular [10, 11, 12, 13] and satellite communications systems [14]. These are based on approximating the subspace multiple signal classification (MUSIC) DoA estimation algorithm and Wiener filter weight evaluation with respect to DoA estimates with the spatial correlation matrix estimated from input signals.



(a)



(b)

Figure 2: (a) Sigmoid (A-function gradient) and (b) Gaussian activation function (s-function spread or standard deviation).

The standard beamforming weight evaluation algorithm involves complex-number matrix inversion and eigendecomposition [4]. Both operations are computationally intensive with large spatial correlation matrix sizes. The neural network substitution aims to eliminate matrix inversion and eigendecomposition with a numeric subspace approximation to achieve real-time weight updates.

The algorithm consists of training and performance stages. During training, the RBF neural network is trained with input-output pairs obtained from Wiener solution for a range of expected DoA angles:

$$\mathbf{R} = E\left[\mathbf{X}(n)\mathbf{X}(n)\right]^H = \frac{1}{N}\mathbf{X}(n)\mathbf{X}(n). \tag{2}$$

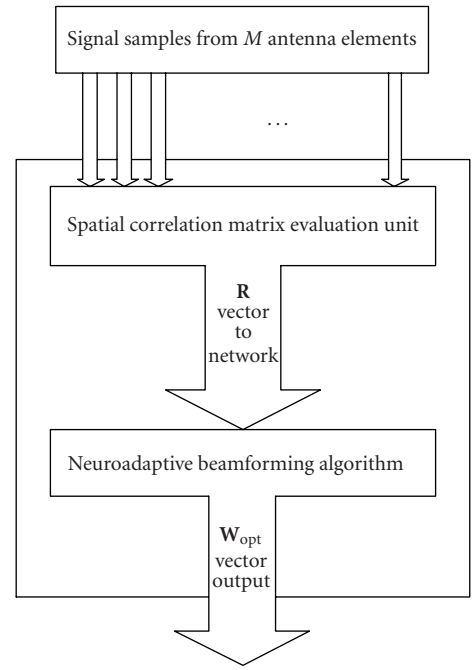


Figure 3: Top-level algorithm decomposition.

The network input is the vector of the upper/lower triangular of the normalized signal spatial correlation matrix estimated from antenna array inputs evaluated by (2). The training output is the Wiener beamforming weights generated using the inverse spatial correlation matrix and output steering direction vector as shown below:

$$\hat{\mathbf{W}}_{\text{opt}} = r\frac{\mathbf{R}^{-1}\mathbf{S}_d}{\mathbf{S}_d{}^H\mathbf{R}^{-1}\mathbf{S}_d}. \tag{3}$$

A fixed number of incoming signal samples are used to estimate $\mathbf{R}$ for $\hat{\mathbf{W}}_{\text{opt}}$ evaluation using the trained neural network. The detailed mathematical derivation of the training algorithm and (3) is shown in [10, 11, 12, 13, 15]. The neural network is expected to estimate $\hat{\mathbf{W}}_{\text{opt}}$ from the trained neural network by presenting unseen spatial correlation vectors to the network input during the performance stage. Only $\mathbf{R}$ is needed to estimate $\hat{\mathbf{W}}_{\text{opt}}$ with a trained network:

$$\hat{\mathbf{W}}_{\text{opt}} = \mathbf{W}_o\left(\varphi\left(d(\mathbf{R}_v, \mathbf{t}_i)b_h\right) + \mathbf{b}_o\right). \tag{4}$$

The neural network solution in (4) transforms (3) into matrix multiplication and is graphically represented in Figure 3.

A real-time criterion is required to determine algorithm real-time performance and estimate performance improvements over the Wiener solution. The antenna output pattern is modified with updated beamforming weight vector and its update time is largely independent of protocol data and chip rates; the protocol frame length determines beamforming update period in TDD mode. TD-SCDMA outlined a 5 milliseconds beamforming update time for smart antenna base stations [3] based on the 10 milliseconds protocol frame length. The frame length for CDMA2000 and WCDMA is

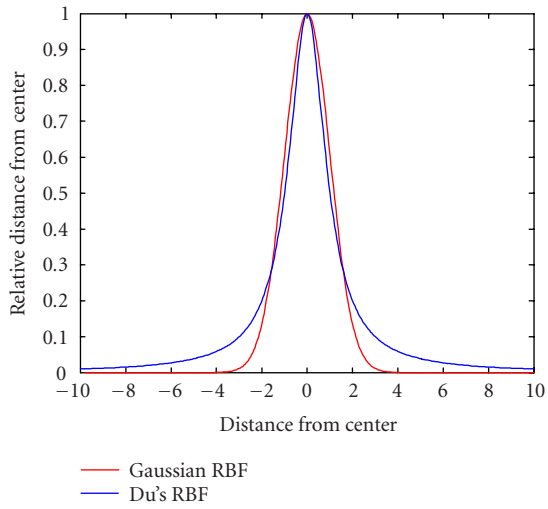


Figure 4: Comparison between Gaussian RBF and reciprocal RBF.

identical to TD-SCDMA, and 5 milliseconds are the real-time criterion for benchmarking performance.

Euclidean distance evaluation requires square, subtract, accumulate, and square-root operations. Square-root operation involves iterative operation in software or hardware using shift-and-subtract, convergence, or CORDIC methods [16]. An simpler alternative without changing network topology is the Manhattan or city block model, where distance between two vectors in a multidimensional space is measured as the sum of total traversed edges that involves only subtract, negate, and add as shown below:

$$d(\mathbf{a},\mathbf{b}) = \sum_{i=1}^{n} |\mathbf{a}_i - \mathbf{b}_i|. \tag{5}$$

The Gaussian RBF can be evaluated in software with lookup tables or iterative approximation with high memory or computation costs. In [17], Du et al. proposed reciprocation as RBF in (6). The RBF function is termed as *reciprocal RBF*, as it is reciprocal to distance between input- and hidden-layer weights. d is the distance from basis centre/hidden-layer weights, and c is similar to standard deviation in Gaussian RBF. Figure 4 shows reciprocal RBF when $c = 1$, and compares it to Gaussian RBF:

$$\varphi(d) = \frac{1}{1 + d^2/c^2}. \tag{6}$$

Both functions are inversely proportional to input distance d, and for positive d, the new RBF is simplified to (7), where $\mathbf{R}_v$ is the input spatial correlation vector, and $\mathbf{t}_i$ is a hidden-weight vector:

$$\varphi(||\mathbf{R}_v - \mathbf{t}_i||) = \frac{1}{1 + ||\mathbf{R}_v - \mathbf{t}_i||}. \tag{7}$$

The arithmetic operations involved in reciprocal RBF are addition, subtraction, and division. This opens up pure software or accelerated software implementations. The new implementation is to substitute reciprocal RBF and Manhattan distance model and generate new output-layer weights and biases using a training algorithm.

From previous discussions on (2) and (4), we can envisage a matrix-centric neural network implementation in Figure 5 with (5) and (7).

The part most affecting real-time performance is the weight evaluation unit that implements (2) and (4). The training algorithm is designed and implemented externally from the neural beamforming weight evaluation unit as real-time training is not yet required for continuous knowledge refinement.

3. PROTOTYPING STRATEGY

We adopted a three-stage prototyping strategy at different abstractions levels to simplify model implementation and to investigate their relative performances. The first step is to implement a simulation model under Matlab for functional analysis and to save development time with a stable development platform with readily available libraries and provide comparison data to further implementation. The second step is to manually translate the simulation model into generic C/C++ working model with an external matrix arithmetic library [18]. The third step is to profile the C/C++ model on the embedded processor platform, analyze for performance bottlenecks, and determine and implement appropriate acceleration techniques to achieve real-time performance.

The Altera APEX FPGA device family enables complex digital system realizations in relatively short time. It is manufactured on 0.18μ process for advanced system-on-programmable-chip (SoPC) realizations and can host multiple Nios embedded processors for high-performance embedded systems applications [19]. It provides extensive support for advanced I/O including PCI and true LVDS suitable for high-bandwidth applications. The device used for investigations is the EP20K200E speed rated at -2X. It contains 376 user I/O pins and 200 000 usable logic gates grouped into 8 320 logic elements.

The Altera Nios is a customizable and expandable embedded soft-core RISC microprocessor targeted to Altera FPGA families. 16- and 32-bit designs are available with 17- or 33-bit maximum memory address for 128 KB or 8 GB of physical memory. The sliding window register bank architecture shows 32 programmer-visible registers from a user-customizable register bank of 128 to 512 total registers to reduce stack operations at subroutines. The instruction set is expandable with 5 opcodes reserved for additional combinational or multicycle sequential logic implemented in parallel with the ALU and controlled by the processor [20, 21]. Integrated C/C++ language development and debugging is provided with the Redhat GNUPro embedded software development toolkit for the Nios platform. The toolkit is based on the open-source GNU compiler collection (GCC) with C/C++ compiler, assembler, software debugger, and profiler for embedded software development using optimized libraries and custom SDK generated with the designer-customized processor core.

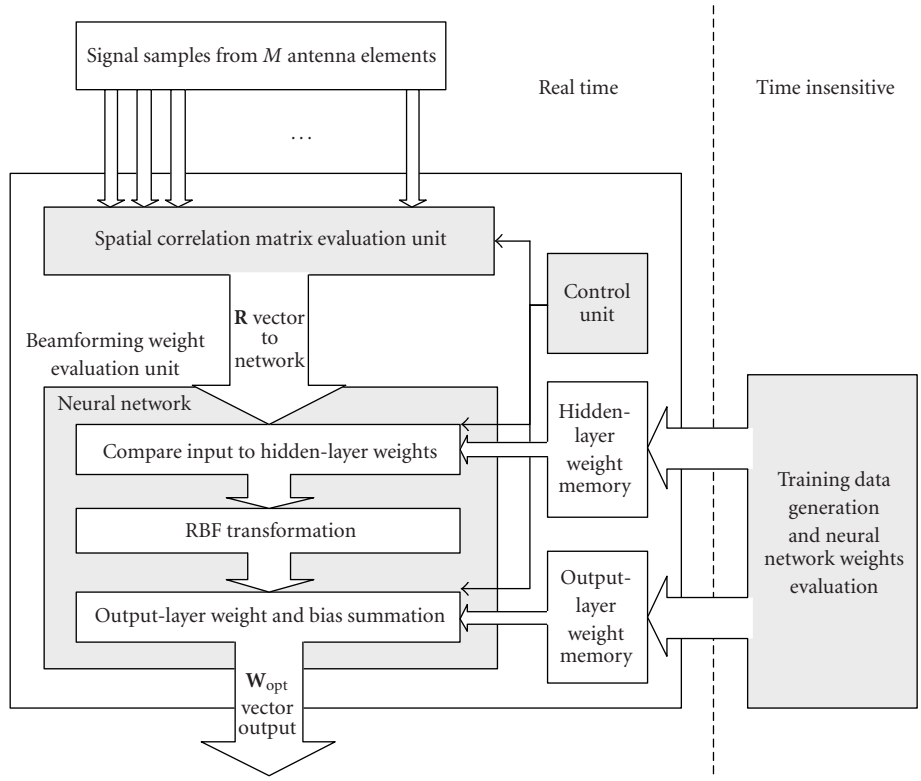


Figure 5: Block diagram for neuroadaptive beamforming algorithm.

A standard platform is followed for hardware and software implementations. Software development is targeted to the Nios synthesized and fitted onto the APEX device. All logic synthesis, functional, and timing analyses were performed with Altera Quatrus II FPGA development tool. The initial target frequency is 33.333 MHz for all designs. If any design is too large to fit onto the device, larger devices in the family with identical speed rating will be used for synthesis and simulation.

4. FUNCTIONAL-LEVEL MODEL WITH MATLAB

The Wiener and neuroadaptive beamforming algorithm are separated into four parts for Matlab implementation:

(1) data sample generation;
(2) input direction-of-arrival (DoA) steering matrix generation;
(3) combine signal and DoA to generate spatial correlation matrix $\mathbf{R}$ with (2);
(4) evaluate Wiener weights $\hat{\mathbf{W}}_{\text{opt}}$ with matrix inversion with (3).

The Matlab simulation model implementation takes advantage of matrix arithmetic for implementing training data generation algorithm. The $(\mathbf{R}, \hat{\mathbf{W}}_{\text{opt}})$ training pairs are used for RBFNN training algorithm to generate network weight/bias matrices. The two processes can be linked together as a single simulation model under Matlab which implements the training data generation algorithm by mapping the algorithm in (2) and (3) to Matlab.

The simulation model integrates parameter input, data generation, network training, and simulation with beamforming weight graphical output to compare exact Wiener output and neural network estimate, integrated on the graphical user interface. It aims to verify the algorithm against claims in [12, 13, 17]; and provides a tool to investigate the algorithm strengths, weaknesses, and limitations. It also benchmarks the future software and hardware-software codesign implementations in performance and accuracy. Figure 6 shows the GUI screenshot and a brief introduction on its usage.

The RBF network is a Matlab data object storing the network structure, weights, and interconnections from newrb or newrbe training functions that train the network in terms of fixed MSE or using zero-error training paradigms. The hidden- and output-layer weights and biases are stored in the network data structure and accessible to the programmer. Weights/biases can be exported to other Matlab routines or other programs for analysis or create custom network models.

Figure 7 shows beamforming performance with generated weight vectors for a 4-element linear antenna array tracking 2 signals at $\theta = 5^\circ$ and $\Delta\theta = 2^\circ$. One thousand signal samples per element and standard Matlab RBF are used. Starting from left, the first $\hat{\mathbf{W}}_{\text{opt}}$ vector corresponds to $(-88^\circ, -83^\circ)$ signal pairs, the second corresponds to

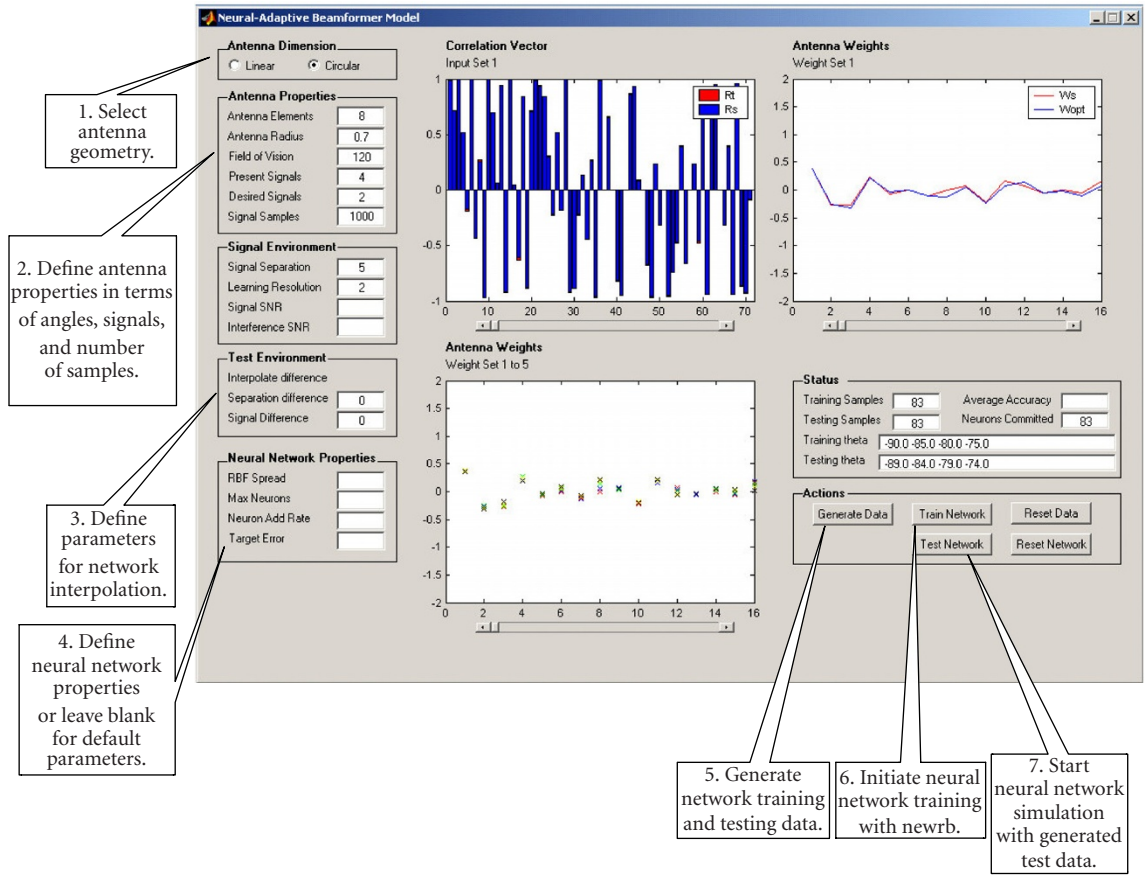


Figure 6: Matlab simulation model running under Matlab R12.1 and Windows 2000.

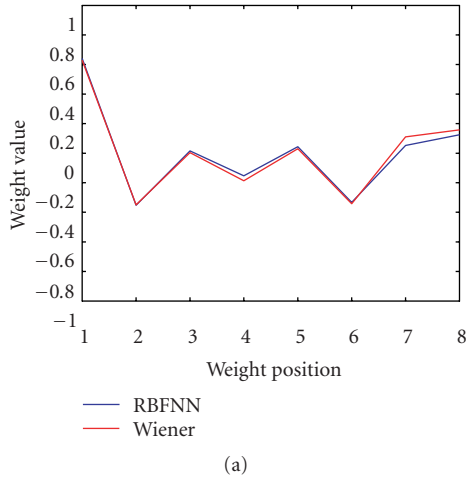
$(-3°, 2°)$ signal pairs, and the third corresponds to $(29°, 34°)$ signal pairs. The neuroadaptive beamforming algorithm can produce close estimates to exact Wiener solution. There are slight problems with signal pairs crossing the horizon and are related to the input signal samples.

To verify and investigate Figure 5, the Manhattan distance model in (5) and reciprocal RBF in (7) are implemented with newrbe training algorithm. The parameters and structures were identical except the algorithm is adapted for Euclidean and Manhattan distance models to test performance and accuracy between them. Additional simulations were performed to investigate the number range in the hidden-layer weights and biases. The plots showed that reciprocal RBF and Manhattan distance model is functionally equivalent to Gaussian RBF and Euclidean distance model. Further investigations into the training algorithm and network parameters can improve approximation accuracy.

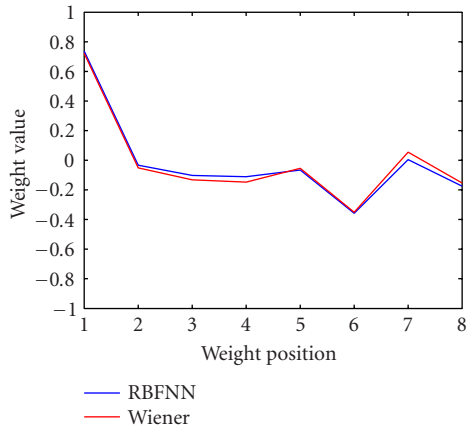
Figure 8 shows a comparison between Gaussian RBF/Euclidean distance model and reciprocal RBF/Manhattan distance model at the θ pair $(-3°, 2°)$ with identical simulation parameters to previous testing.

During simulation model development, some limitations were realized and possible ways to overcome them are discussed. A realistic cellular antenna array must perform beamforming operations in different signal and interference conditions. The criteria for successful generalization include these factors: the antenna tracks users and performs beamforming at arbitrary input elevation and azimuth angles, and it tracks a changing number of user and interference sources at all times. Assuming $\mathbf{S}_d = \mathbf{A}$ for first K output signals solves the problem of output direction finding and simplifies the weight evaluation problem to a nonlinear $(\mathbf{R}, \hat{\mathbf{W}}_{\text{opt}})$ mapping. This simplification leads to two related issues and possible limitations: training data must allow generalization for arbitrary input elevation/azimuth for output beamforming, and must consider changing signal and interference sources.

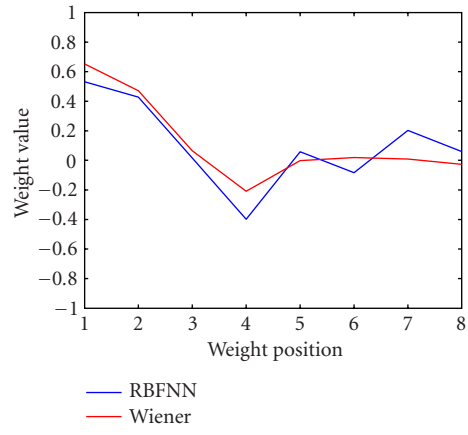
This issue is critical to real-life beamforming since the antenna must estimate $\hat{\mathbf{W}}_{\text{opt}}$ for different incoming signal elevations. The most important training constraint is keeping the one-to-one $((\mathbf{R}, \hat{\mathbf{W}}_{\text{opt}})$ or $(\mathbf{A}, \hat{\mathbf{W}}_{\text{opt}}))$ mapping. The aim is to train the network for arbitrary output steering with training samples. The second issue is also critical for real-life operations due to users continuously moving in and out of the cell. Credible neural network training strategy must consider the combined effects of changing signal elevation angles, changing output signal positions and changing number of information and interference signals. It may be possible to extend the algorithm to include $\mathbf{S}_d$ for more accurate beamforming. The length and elevation position can be learned by the network for different weights.



(a)



(b)

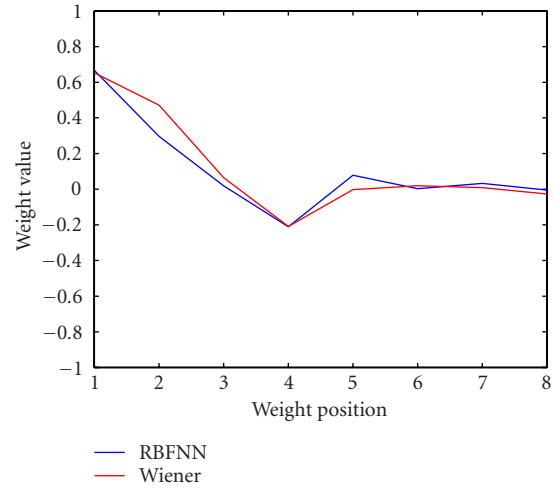


(c)

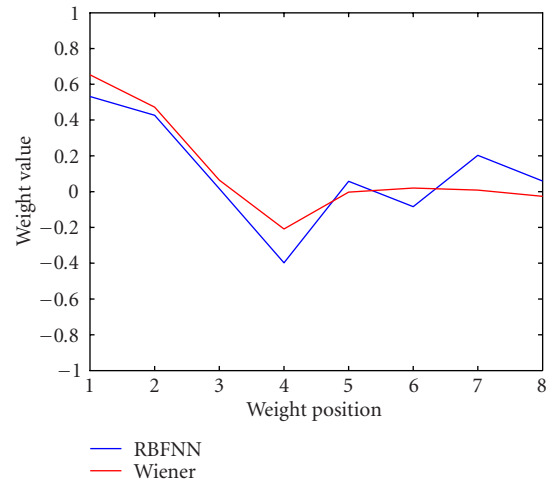
Figure 7: Beamforming weights from Matlab simulation model for 4-element configuration (two signals, $N = 1000$, $\theta = 5$, Ires = 2): (a) $(29, 34)$, (b) $(-88, -83)$, and (c) $(-3, 2)$.

Sources of generalization error for simulation model are grouped into three cases:

(i) number of samples in the signal data matrix;
(ii) neural network spread parameter;
(iii) learning resolution.



(a)



(b)

Figure 8: Beamforming weights from (a) Gaussian RBF/Euclidean distance model and (b) reciprocal RBF/Manhattan distance model (two signals, $N = 1000$, $\theta = 5$, Ires = 2, $(-3, 2)$).

The neuron spread parameter is fixed at 1.0 for simulation cases. Matlab documentation stated if the function width is set too wide, it may cause the network to overfit during generalization. Computation effort increases with neuron numbers (i.e., the hidden-layer weight matrix size) and with output vector length (i.e., the output-layer weight matrix size). It was found that the learning resolution during network training, critical to beamforming accuracy and speed by controlling the number of neurons, should be less than half of the angular separation to minimize neuron usage while retaining required accuracy with unity RBF spread. A finer resolution requires more neurons to fit the approximation with a narrower spread, whereas a coarser resolution requires less neuron to fit the approximation but a wider spread parameter.

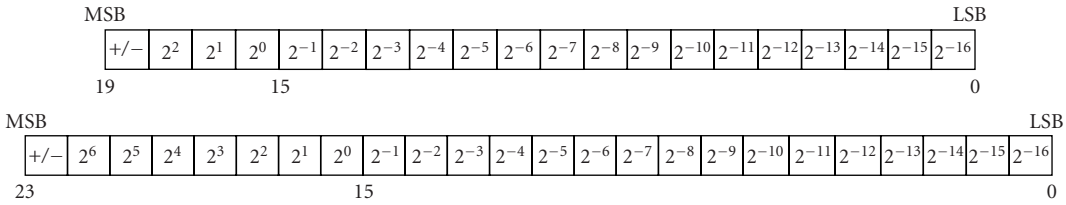


Figure 9: 20-bit and 24-bit fixed-point number format.

5. IMPLEMENTATION ISSUES AND OPTIONS

A hard real-time system produces completely irrelevant results or fails beyond recovery if the real-time constraint (deadline) is violated at a particular time instance [22]. A soft real-time system produces undesirable behavior or increasingly irrelevant results with respect to time but does not fail beyond recovery if the deadline is violated on a low-probability and low-risk basis. Beamforming weight relevancy depends on deadline and drops rapidly with time and is related to user movement. Desired signal for one user can be a strong interference to another user if the deadline is violated on a regular time basis. The beamforming algorithm from $\mathbf{R}$ evaluation to $\hat{\mathbf{W}}_{\text{opt}}$ evaluation is a hard real-time operation from the above criterion. There is no requirement for online network weight update and hidden- and output-layer weights can be generated externally. The network training operation is time-insensitive for offline training, and soft real-time for online training. For future implementations with continuous neural network weight updates, network training may become hard real-time, and must not affect the beamforming operation.

To meet purpose, environment and resources constraints, tasks in embedded systems may be split between hardware and software implementations to satisfy latency, processing performance or throughput requirements, and available processing resources. Generally, time-critical tasks and tasks with low latency, high performance, or throughput would often be implemented in hardware to exploit parallelism and component specialization and optimization. Complex, soft real-time or time-insensitive tasks or tasks with high costs and low occurrence would often be implemented in software to save hardware resources. The above split is dependent on the available resources in terms of information processing units, memory, and peripherals interfacing with physical environment. The neural beamforming operation is a hard real-time operation and must be performed within time constraint; therefore a pure hardware or highly accelerated software implementation is in a better position to fulfill the real-time requirements.

Fixed-point representation interprets a bit string as fraction with imaginary decimal point. The decimal point position does not change after arithmetic operations [23, 24]. The integer place value concept is valid in fixed-point notation with a slight difference. If p/n bits represent integer q/n bits represent fraction, then integer range is 2^P; and precision is 2^{-q}. Different fixed-point number formats, such as the 1.15 [23], 16.16, or 18.14 fixed-point formats are used in software requiring high-performance fractional arithmetic without hardware floating-point arithmetic [24].

Some assumptions were made based on the characteristics for the incoming data to choose an appropriate number format. Input signals and interferences are assumed to follow a steady-state Gaussian/normal distribution model [15]. It is possible to devise a fixed-point format for hardware implementation with normal distribution model $\mu = 0$ and $\sigma = 1$. The probability of incoming signal value falling in $\pm 4\sigma$ is 99.7%. The worst case scenario would be all incoming samples at maximum or minimum. The other criterion is minimum data width to represent required range and precision to minimize storage and bus width. The third criterion is to use minimum additional hardware to implement the number format. From previous assumptions, it is possible to draw up the criteria for designing a suitable number format. The maximum value is ≥ 4 to cover $\leq 99.7\%$ of possible signal samples. The data width, byte order, and sign convention must be compatible with Nios and LPM arithmetic units, and the output fixed-point number format is wide enough to store accumulated product of repeated MAC without arithmetic overflows. Underflows could be neglected.

Figure 9 shows the 20- and 24-bit fixed-point number formats. The MSB is two's complement sign bit. The next 3/7 bit represents an integer value between $\pm 7/\pm 64$. The 16 LSB bits represent a fractional with precision of 2^{-16}. The formats can be sign-extended to 16.16 fixed-point numbers with 16 integer bits to avoid arithmetic overflows in long MAC sequences and compatibility with Nios data path.

Multidimensional data and multiple signals are represented as matrices in signal processing and neural network algorithms. For example, multiple discrete signals each with n samples can be represented as $m \times n$ matrix. Matrix arithmetic is a challenge to hardware and software designers with its complexity. Multiple antenna elements are necessary to obtain a reasonable gain for smart antenna beamforming. For outdoor environments, antenna arrays with 6 to 10 elements were proposed [5], with the 8-element array antenna as the most researched design. The $\mathbf{R}$ evaluation unit uses a 4-element configuration for demonstration, and results will be extrapolated for 6- and 8-element configurations to estimate their performance.

It can be shown that the products between a matrix and its transpose are mirrors at upper and lower triangular. This enables simplification by eliminating lower triangular evaluations to halve multiply-add operations, eliminates upper triangular extraction, and reduces memory requirements to $2M(M + 1)$ elements.

```
for i = 1 : m do
    for j = m : n do
        for k = 1 : p do
            C(i, j) = A(i, k)A(k, j) + C(i, j)
        end for
    end for
end for
```

Algorithm 1: Algorithm for matrix-transpose multiplication.

The total multiplications required in Algorithm 1 are $(m(n+1)/2)p$, and additions are $(m(n+1)/2)(p-1)$. Its complexity is $O(n^2)$. The total inner loop iterations are 10 for the 4-element case, 21 for the 6-element case, and 36 for the 8-element case. The analysis is valid for integer and complex-number matrices.

Complex numbers for I/Q signal representation adds complexity to the problem. Abstract software implementations with dedicated data structures hide implementation details. Abstract complex number arithmetic operations in software are unrealistic with digital hardware implementation. The requirements of processing two related numbers doubles/quadruples required memory and processing steps. It becomes a major concern for designers in terms of processing speed and logic element consumption.

The **R** evaluation unit is mathematically and algorithmically identical to the Matlab model implementation when translated to C/C++ implementation, as shown in Algorithm 2, except the step to generate input steering matrix is not performed, and signal samples are directly used. The differences between Matlab and C/C++ implementation will be centered on number format and level of software functional abstraction.

Matlab has built-in complex number data type and operations, while C/C++ requires users to define the data type with an abstract data class and member functions for data manipulation. The custom data structure contains two 32-bit integers to represent real and imaginary components respectively. Their signs are implicitly stored with the components. The custom data format for 16.16 fixed-point complex number is shown in Algorithm 3.

The integer data type is defined as a 16.16 fraction. The new data type cfraction stores the real and complex components as two fractions. Addition and subtraction algorithms are standard mathematical evaluations, and multiplication algorithm requires a 16-bit shift to LSB to keep the correct number format. The following flowchart shows a simplified scalar **R** evaluation algorithm.

Variables from aa to af are used to store temporary results. Loop unrolling in the while loop shows future algorithm parallelization and saves loop overheads by performing multiple operations in a single iteration.

A C++ RBFNN forward evaluation unit was implemented with (4). A C++ matrix arithmetic library from Ohio State University [18] was adapted to evaluate matrices. Its block diagram and algorithm data flow are shown in Figure 10.

```
t [·] = (0, 0);
while i < SAMPLE do
    aa = mul (a[0][i], a[0][i])
    ab = mul (a[0][i + 1], a[0][i + 1])
    ac = mul (a[0][i + 2], a[0][i + 2])
    ad = mul (a[0][i + 3], a[0][i + 3])
    ae = add (aa, ab)
    af = add (ac, ad)
    t[0] = add (ae, af)
    i = i + 4
end while
```

Algorithm 2: Matrix-transpose multiplication for **R** evaluation unit.

```
typedef short int    fraction;
typedef int          dblfraction;
typedef struct cmplx
{
    fraction real;
    fraction imag;
} cfraction;
```

Algorithm 3: Custom data format for 16.16 fixed-point complex number.

The hidden/output-layer weight/bias matrices in Figure 10 as memory blocks were implemented as global static C arrays and wrapped around to matrix from the library to enable matrix arithmetic. Other matrices and vectors with dynamically assigned values are instantiated to matrix. The implementation is algorithmically close to Matlab RBFNN and (4) is split into 4 major components: Euclidean distance evaluation, RBF transformation, output-layer weight summation and output-layer bias addition.

The software RBFNN implements a 4-element beamforming antenna array. The Euclidean distance between input **R** and each hidden-layer weight vector is the output from hidden-layer neurons and was implemented with standard arithmetic operations and sqrt $(\cdot)$ in standard C mathematic library. The distances are arranged into a column vector with length equal to number of hidden-layer neurons. The RBF is sequentially applied to each element in the hidden-layer output vector to transform Euclidean distances to similarity index. The similarity index values replace Euclidean distances in the vector to save memory. The output vector is estimated by multiplying output-layer weight matrix to hidden layer output vector. The output-layer bias vector is added to the matrix-vector product to estimate the output.

The C++ RBF also accommodates the reciprocal RBF and Manhattan distance model when the original implementation is fully debugged. To provide hidden/output-layer weights/biases to accommodate new distance evaluation and RBF functions, the Matlab training model is modified to generate the output-layer weights and biases for the C++ forward evaluation unit, while the hidden-layer weights/bias remain unchanged. The RBF software routine does not take possible hardware acceleration or algorithm parallelization into account at this stage.

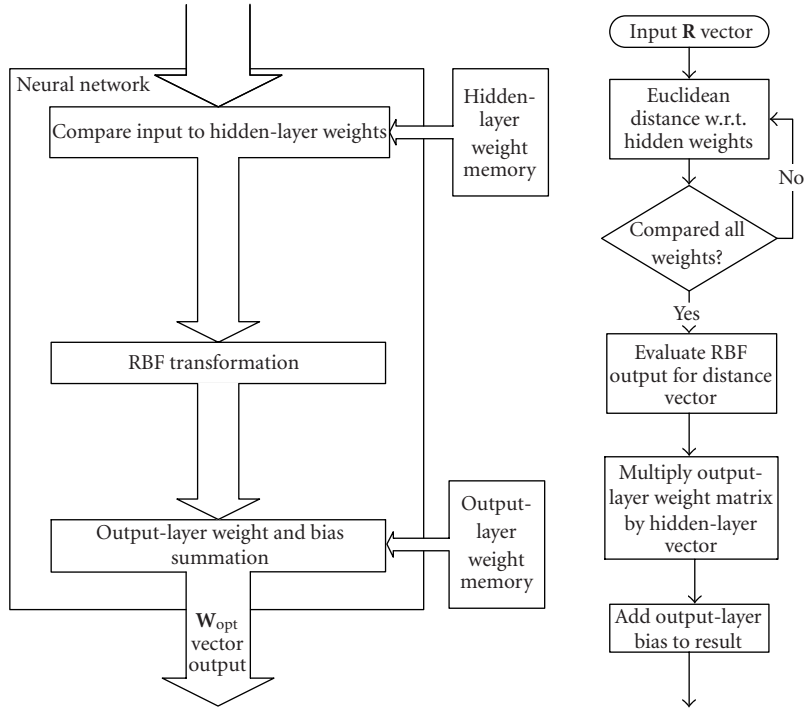


Figure 10: Block and flow diagrams for neural network implementation.

Profiling is performed to obtain a percentage breakdown of time for each major subroutine in the C/C++ solution. Table 1 shows the time breakdown in cycles, milliseconds, and percentage for the major subroutines in the algorithm running at the initial target 33.333 MHz clock frequency. For the 4-element case and assuming that processor performance increases linearly with clock frequency, a Nios processor running at 542.87 MHz can satisfy the 5-milliseconds real-time requirement. Performance extrapolations are made on 6/8-element configurations using the software implementation. The figures were based on dividing the number of clock cycles for each subroutine by the number of iterations in the subroutine for the 4-element scenario, and multiplied by the expected number of operations performed for 6/8-element scenarios. The number of neurons/samples and other parameters are unchanged. The 6-element case requires the processor to run at 624.54 MHz, and the 8-element case requires 718.53 MHz. These estimates do not consider the number of neurons.

The slowest operation is hidden-layer bias multiplication that multiplies a constant fixed-point number by a column vector with 57% of total cycles. It multiplies 88 elements in a column vector by a fixed hidden-layer bias. This is an $O(n)$ operation increasing in terms of neurons. The second slowest operation is distance evaluation to find Manhattan distances between input and neurons. Its complexity increases both in terms of a number of antenna elements (**R** length) and the number of neurons. If the number of neurons is fixed for different antenna configurations, then its complexity is identical to hidden-layer bias multiplication. Matrix multiplication/addition and RBF operations were also major operations. Taking the hidden-layer bias multiplication from the total time, it can be seen that the total time for output weight multiplication/bias addition and RBF operations is 58.25% of the total time used for beamforming weight evaluation.

6. HARDWARE/SOFTWARE OPTIMIZATIONS

With bottlenecks identified in Table 1, it is possible to address these shortcomings with software optimizations, hardware accelerator units, and dedicated hardware implementations. The most significant software bottleneck is memory access for vector/matrix element extraction in distance evaluation operation, and function call variable return for evaluating the hidden-layer multiplication by hidden-layer bias.

The distance evaluation and matrix-constant multiplication algorithms were unchanged, except for hidden layer weights which were accessed with pointers and not matrix row extraction. The hidden-layer output column matrix is evaluated and assigned element by element. Hidden-layer Manhattan distance is multiplied with the fixed bias in the same loop and assigned to the column matrix element. A speedup of 38 was obtained for hidden-layer bias multiplication by changing memory read operations to use pointer arithmetic for direct static memory access. Distance evaluation was faster by 16.5 with pointer operations. When the improved software was used without hardware accelerators, it completes a single beamforming operation in 22.15 milliseconds, or a speedup of 3.46 over the original implementation.

Table 1: Running time breakdown for individual operations in cycles and milliseconds.

Operation	Clock cycles	Time (ms)	Percentage
R evaluation	152 156	4.56	5.61
Distance evaluation	330 817	9.93	12.18
Hidden-layer bias multiplication	1 553 749	46.61	57.24
RBF evaluation	194 941	5.84	7.18
Output weight multiplication	328 284	9.85	12.09
Output bias addition	154 408	4.63	5.69
Total	2 714 355	81.22	100

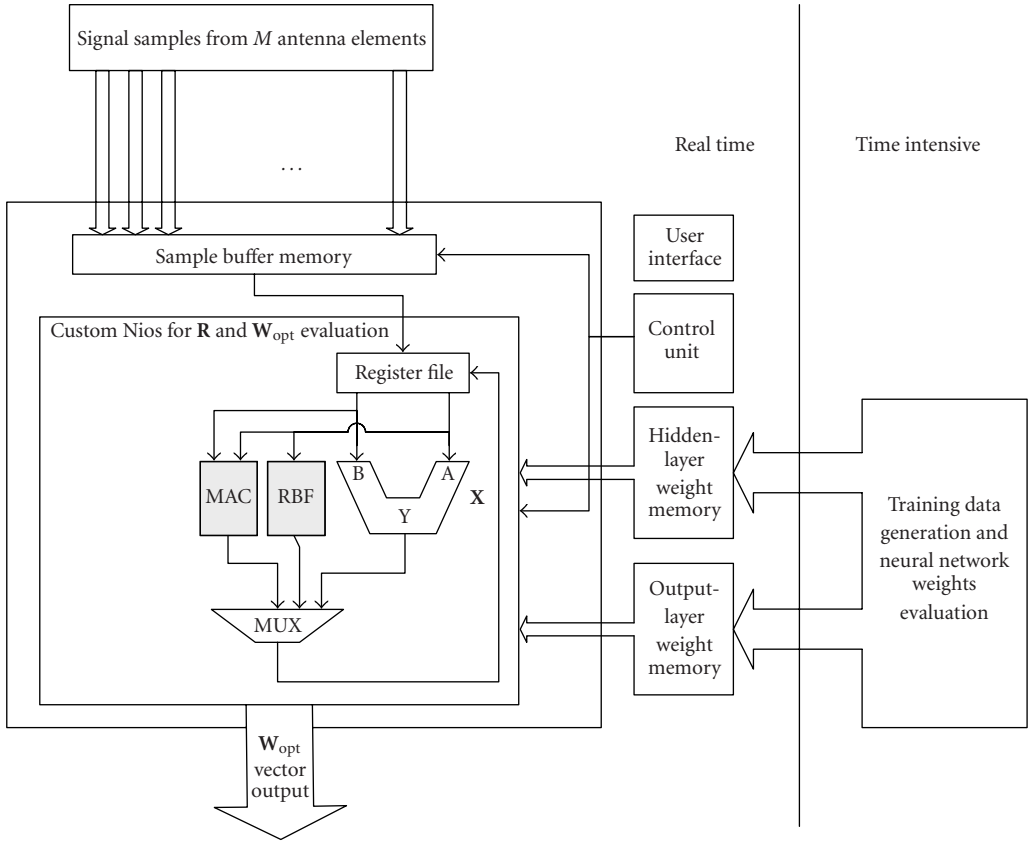


Figure 11: Customized Nios for accelerated neuroadaptive beamforming algorithm.

The problem with Nios is low multiplication, MAC, and reciprocation performance for $\mathbf{R}$ and $\hat{\mathbf{W}}_{\text{opt}}$ estimation. These operations can be accelerated by hardware. A 24-bit fixed-point dual-mode (MAC and multiply) MAC unit fp_mac and RBF evaluation unit fp_rbf were integrated to the Nios data path.

Figure 11 shows the neuroadaptive beamforming algorithm with the custom Nios processor. It is assumed that data samples were stored in a buffer available from register file by load/store instructions, and is reflected in coding that samples were embedded as static program data. The MAC unit maps to complex matrix-transpose multiplication and neuroadaptive beamforming weight estimations with hidden/output weights generated externally from the Nios processor. The RBF unit maps to the RBF distance evaluation operation. The single memory block is conceptually split into sample buffer, hidden- and output-layer weights.

Element-by-element matrix multiplication can be rearranged into scalar multiply-accumulate sequences. The cost of pure software MAC is the required register load/store operations with each multiply-accumulate sequence and in terms of multiplier implementation. A custom MAC unit is assembled with the parameterized lpm_mult and altaccumulate blocks separately. The accumulator input data width equals to multiplier output data width to minimize rounding errors during fixed-point MAC sequences, and rounding is performed at accumulator output. An additional feature for this implementation is that a multiplexer is added for selecting multiplier/accumulator output for each operation without clearing previous MAC results. The MAC control unit

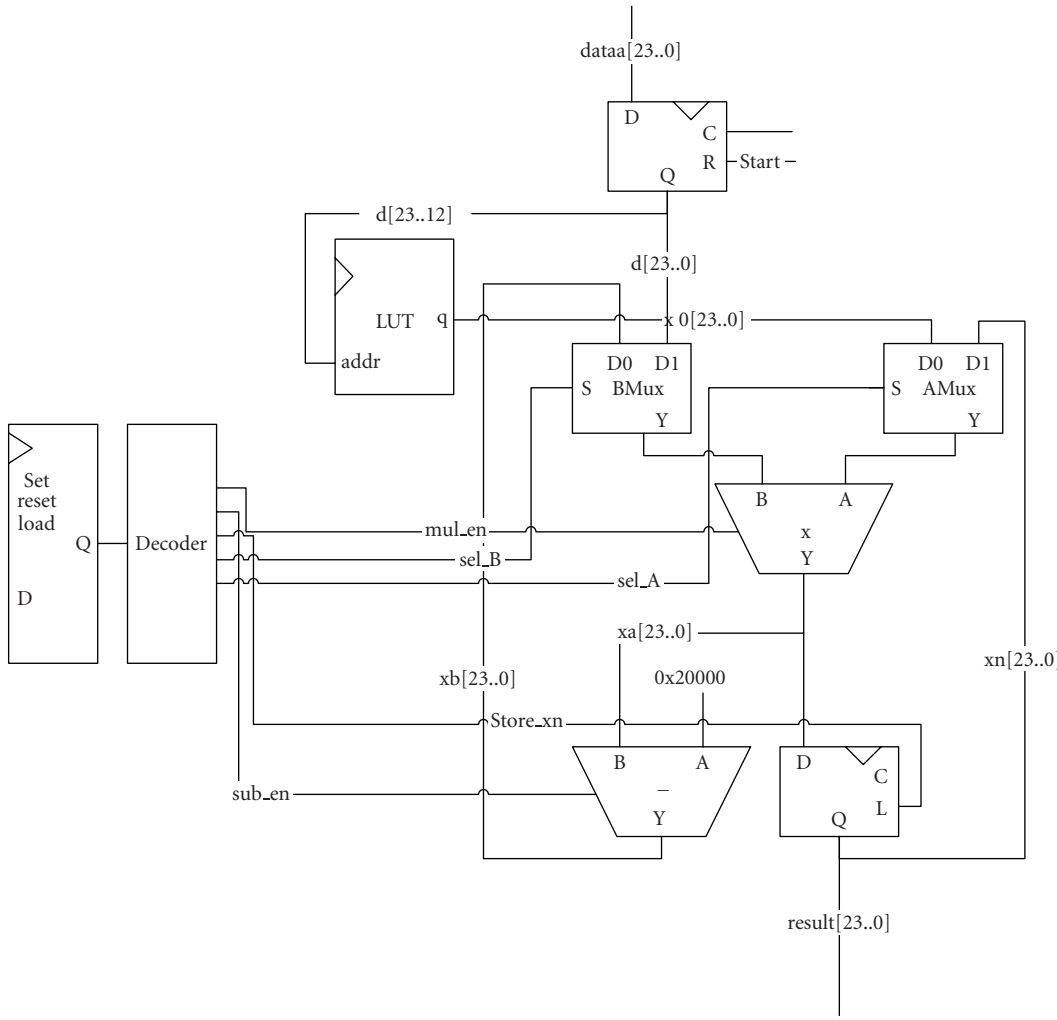


Figure 12: Block diagram for iterative reciprocation.

for both implementations is a 4-state FSM and is activated by start signal from the processor to load input registers and start operation. Add/subtract control is performed with two LSB bits in the 11-bit prefix input to control the accumulator addnsub input. 000_{hex} indicates add, 001_{hex} indicates subtract, and 002_{hex} indicates select multiplier result from custom MAC unit. The prefix value $7FF_{\text{hex}}$ is used for accumulator reset when starting a new MAC sequence.

The RBF evaluation unit evaluates $1/(1+d)$ similarity index during beamforming weights estimation. As Manhattan distance does not produce negative distance, RTL handling for divide-by-0 or arithmetic overflows is not required. The bottleneck for hardware RBF evaluation is divider performance. The Newton-Raphson iterative method finds roots to equations numerically [25].

The expression $x^{(i+1)} = x^{(i)}(2 - x^{(i)}d)$ [16] is the iterative solution for 1/x. To shorten computation time, a lookup table is used to store initial iteration values $x^{(0)}$ to 8-bit precision for the input 8.16 fraction d between $1 < d < 256$ and shortens the total iterations to 2.

Figure 12 shows that a single iteration is separated into three hardware blocks. The data path in Figure 12 maps the 8.4 fixed-point input to $x^{(0)}$ LUT input address bus to map $x(0)$ in 0.16 fixed-point format. The 8.4 input is subtracted by 1 for correct memory address. The LUT is implemented with on-chip ESB (embedded system block) memory as ROM. Lookup values are calculated with a spreadsheet and converted into 0.16 fixed-point representations by multiplying the quotient by 65535 and converted into hexadecimal values. Multiplier sharing between x^a and $x^{(i+1)}$ evaluation is achieved with two multiplexers to select input to multiplier by the control unit for cycles 1 and 3 shown in previous iterative expressions. Multiplexer A (AMux) to multiplier dataa input corresponds to s_init and s_next; and multiplexer B (BMux) to multiplier datab input is connected to s_sub and dataa. The multiplication result s_mul from bit 39 to 16 is connected to both s_next register and subtractor B input. The s_next register is only updated every second multiplication operation. The data path involves feedback at s_sub and s_next outputs. The design is modified to perform RBF

Table 2: Running time in cycles for **R** evaluation with/without hardware MAC unit.

Multiplier implementations	Cycles
16-bit hardware multiplier	152 156
Custom MAC unit	63 194

Table 3: Running time in cycles for algorithm operations with/without hardware accelerator units.

	Software	Accelerated
Hidden-layer bias multiplication	40 897	23 374
RBF evaluation	194 941	14 583
Output weight multiplication	328 284	179 156

evaluation by adding 1 to the integer component when the input value is stored to the register.

Performance gains from hardware acceleration on matrix multiplication and RBF evaluation are measured by modifying software **R** evaluation and neuroadaptive beamforming weight evaluation units to use hardware accelerators. The custom 24-bit MAC is used in matrix-transpose multiplication for **R** evaluation and in matrix-matrix multiplication for neural network evaluation. The tests investigate the speedup from MAC to matrix operations. Table 2 shows performance comparisons for software, with Nios multiplier, and custom MAC unit fp_mac for **R** evaluation.

The MAC unit provided a speedup of 2.4 over the 16-bit hardware multiplier, and in terms of time in milliseconds, the accelerated implementation requires 1.8 milliseconds to complete the 64-sample **R** evaluation. The number of cycles increases in terms of a number of elements and samples. The MAC and RBF accelerators were also used for beamforming weight evaluation to alleviate identified bottlenecks in Table 1. The matrix arithmetic library and the forward evaluation unit were modified to use MAC and RBF accelerators.

Table 3 shows timing results for both implementations. The required beamforming time for the optimized software plus accelerated hardware implementation is 11.74 milliseconds without **R** evaluation and 13.64 milliseconds with **R** evaluation when the processor is running at 33.333 MHz. It can be concluded that simple hardware accelerator units designed and implemented to speedup the operations were useful for parts with $O(n)$ complexity, while further parallelism is required to speed up operations with $O(n^2)$ or higher complexity.

If multiple signal inputs are placed on M data buses synchronous to a clock signal sequentially to parallel arithmetic units below, then matrix-transpose multiplication for **R** evaluation may be completed in $O(n)$ time only in terms of a number of samples in $\mathbf{X}(n)$ by parallelizing the MAC sequences in terms of a number of antenna elements.

The complex-number MAC unit shown in Figure 13 is assembled from two integer MAC units to evaluate two complex numbers and an add/subtract unit to combine the results. If real and complex components are evaluated in parallel without data interdependence, then multiple complex-

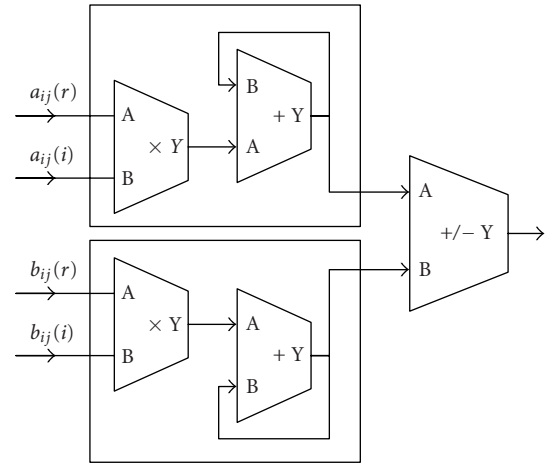


Figure 13: Block diagram for complex MAC unit.

number MAC units could be initiated in parallel to form **R** evaluation unit shown in Figure 14. Since the input I/Q components are represented as complex numbers, multiple complex-number MAC units can be instantiated in parallel to evaluate long MAC sequences required for **R** evaluation. The number of parallel complex MAC units for M elements is $(M(M+1)/2+M(M+1)/2-M)$. For a 4-element design, 16 complex MAC units were required. Outputs of $2M \times 32$ bits are used for the M-element case, therefore 8 20-bit inputs and 8 32-bit outputs for real and complex components are used for I/O in the 4-element case. A simple counter-based control unit is used for controlling evaluation and output stages. After the initial 4-cycle pipeline delay, 64 cycles were used for evaluating input samples. After 64 samples were evaluated, four cycles were used to output the **R** matrix sequentially to the output bus. Each value is valid on the bus for 1 cycle. Averaging or divide-by-64 is performed with left shifts by 6 bits during the output states. Each cycle returns a column vector as stored in the complex-number MAC units. The evaluation unit resets itself after **R** is presented to the data bus.

The estimated time for single **R** evaluation for 64-sample block is 2.8 microseconds for a 40 nanoseconds clock period. The input bus units could be external to the Nios data path and be connected to the ADC units/antenna IF section for direct **R** evaluation.

Using the Altera Stratix FPGA as an alternative platform with higher performance, some timing estimations for the hardware-accelerated solution running at 78.23 MHz were obtained and shown above.

As performance bottlenecks lie in sequential evaluation for **R** and neural network output, it is proposed that the **R** evaluation be moved to the hardware-based solution. The neural network evaluation could be performed in software with further refined hardware accelerators to achieve instruction-level parallelism for vector operations. Improved software algorithms to alleviate software bottlenecks are also required for further speedup.

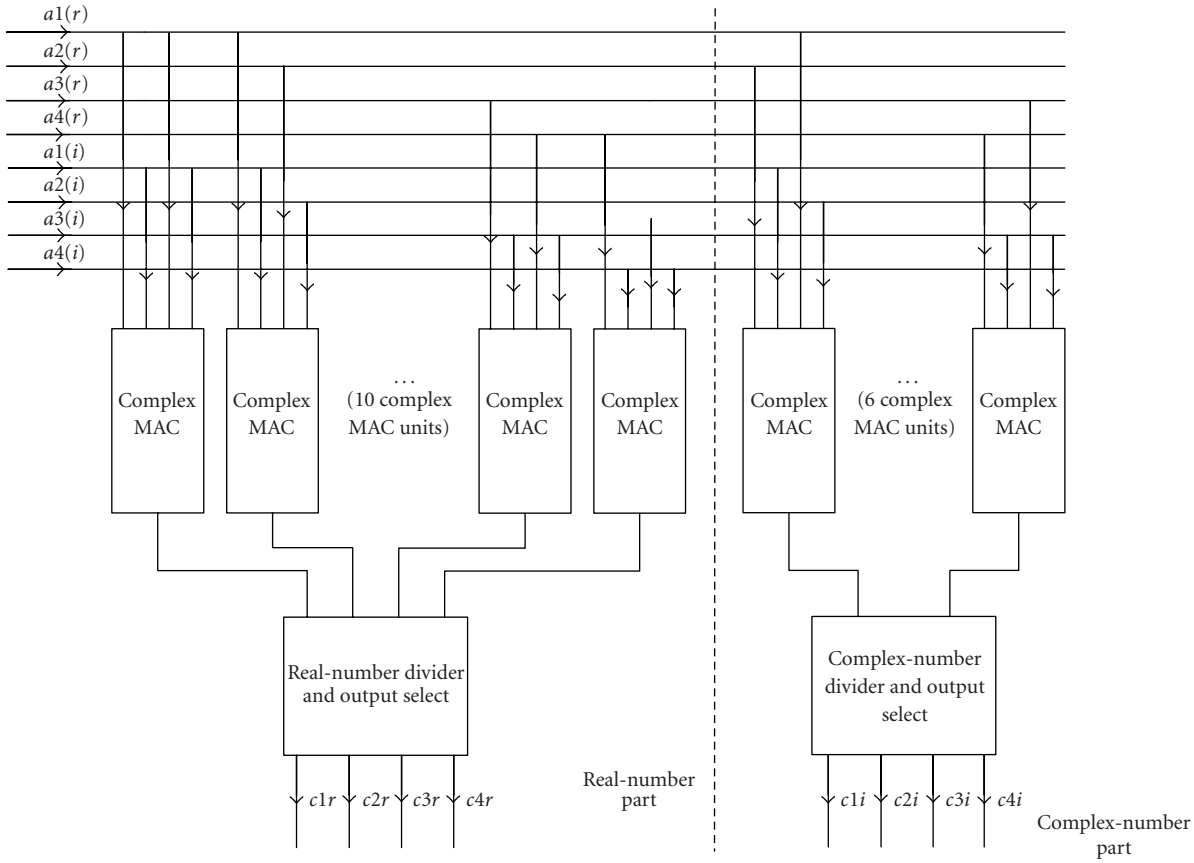


Figure 14: Data path for 4-element sequential **R** evaluation unit with complex-number MAC.

A hardware-software solution exploits hardware acceleration for fast matrix processing required for **R** and neural network output yet retains flexibility and adaptability to change neural network weights/biases in hardware-assisted software solution.

As shown in Figure 15, it is possible to integrate the **R** evaluation unit described in this section into Nios custom instruction framework and control the unit directly from software. The external input buses are connected to the antenna/IF section ADC units to acquire signal samples. The matrix block multiplier unit designed for accelerating matrix arithmetic in neural networks can be integrated to and controlled by ALU in the custom instruction framework. The RBF evaluation unit could also be modified to perform instruction-level parallel (ILP) operations with two operands in registers Ra and Rb as shown in Figure 16. A similar approach may be applied to MAC operations.

A 24-bit block matrix arithmetic unit can be implemented to accelerate matrix arithmetic by performing block matrix operations in hardware in $O(1)$ time with parallel multiply-add units to perform 2×2 matrix multiplication as block matrix arithmetic [26]. The input and output are serialized and mapped to the custom instruction framework shown by the I/O waveform below. Internally, fully parallel operations can be supported with parallel storage registers for input variables to evaluate the result in 2 clock cycles. The output can be serialized again into single data output. 10 clock cycles are required to evaluate a single 2×2 fixed-point matrix multiplication, as shown in Figure 17.

In [26], it can be seen that the number of block operations depends on block size, and can be determined mathematically by Mn/k^2 for a square $k \times k$ subblock. The number of additions to combine completed subblocks is Mn. Assuming that 10 clock cycles are required to evaluate 1 block matrix multiplication, 1 clock cycle is required to evaluate 1 addition, and 4 additions are required to add two 2×2 matrices, then the number of clock cycles required is $(10(Mn/k^2)+Mn)$ for general matrix evaluation.

Performance estimations were made with output weight multiplication in Table 3 when the software algorithm is accelerated by the block matrix multiplier unit. Assuming the output weight vectors are divisible by 2 and an additional zero column vector is appended to the hidden-layer output to create an $n/2$ block vector for multiplication, the clock cycles required to multiply the output weights to the hidden-layer vector are estimated as in Table 4 for an 88-neuron RBFNN.

This proposed matrix-centric block multiplier provides significant performance improvements for matrix multiplication in neural network evaluations, and can be extended to matrix addition and subtraction.

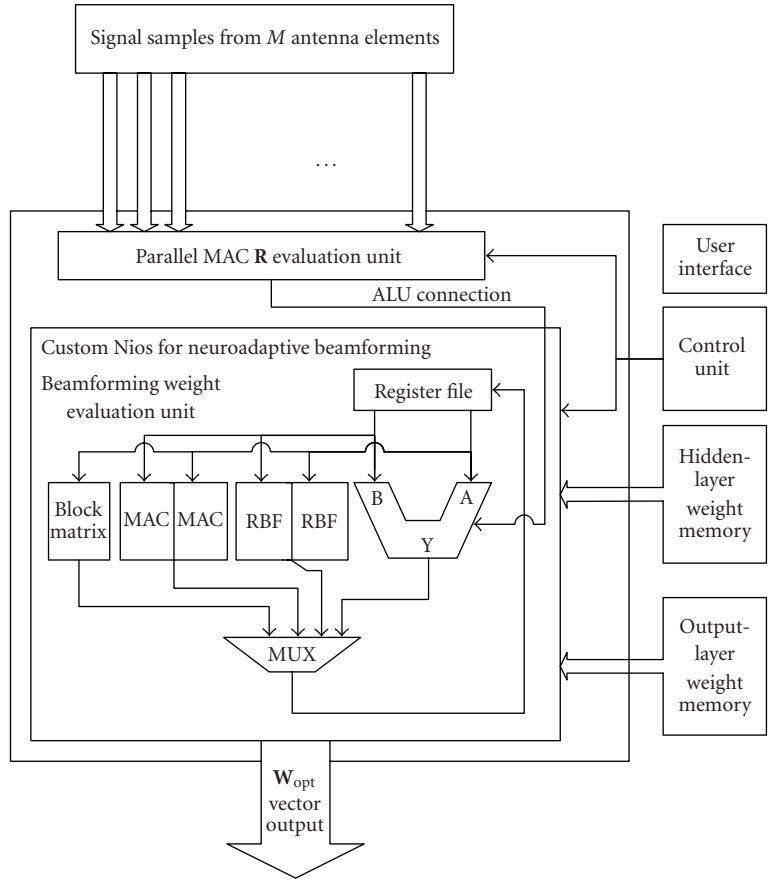


Figure 15: Block diagram for improved hardware-software solution for neuroadaptive beamforming.

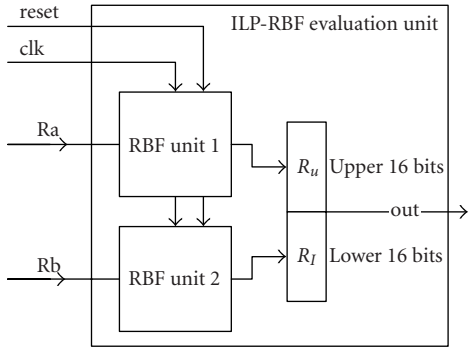


Figure 16: Block diagram for instruction-level parallelism for RBF acceleration.

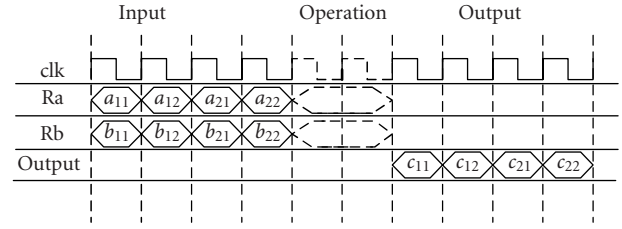


Figure 17: Proposed timing waveform for block matrix arithmetic unit.

7. CONCLUSIONS

In this paper, we explore prototyping options and implementation of a neuroadaptive antenna beamforming algorithm using RBF neural network using FPGA and system-on-programmable-chip (SoPC) approach. The application has many performance bottlenecks for pure software implementation as shown in Table 1. Those bottlenecks can be partially alleviated with improved software algorithms and simple hardware acceleration units to reduce its complexity in an attempt to achieve the 5 milliseconds real-time beamforming constraint outlined in Section 2. Investigations into

Table 4: Performance estimations in cycles for 2×2 block matrix multiplier.

Elements	Block multiplication	Addition	Total
4 elements	1760	704	2464
6 elements	2640	1056	3696
8 elements	3520	1408	4928

Table 5: Time estimates for 4-, 6-, and 8-element antenna configurations.

Operation	4-element		6-element		8-element	
	Cycles	Time (ms)	Cycles	Time (ms)	Cycles	Time (ms)
R evaluation	63 194	0.81	132 707	1.70	277 498	3.55
Distance evaluation	19 933	0.25	19 933	0.59	19 933	0.59
Hidden bias multiplication	23 374	0.52	23 374	0.69	23 374	0.69
RBF evaluation	14 583	0.19	14 583	0.43	14 583	0.43
Output weight mul.	179 156	2.29	268 734	3.44	358 312	4.58
Output bias addition	154 408	1.97	231 612	2.96	308 816	3.95
Total	454 648	6.03	825 310	9.81	1 360 828	13.79

pure hardware implementations of **R** evaluation unit with parallel MAC solutions showed that pure hardware solutions can provide high performance suitable for real-time implementation by exploiting parallelism and optimal hardware design. However, their current resource usage is very high.

Performance estimations presented in Table 5 showed that by a combination of improved device technology, simple hardware accelerators, pure hardware **R** evaluation unit, and improved software algorithms, it is possible to achieve real-time beamforming for the 4-element scenario. However, further parallelism using the possible methods outlined in this section is required for further performance gains to achieve real-time beamforming for 6/8-element or larger configurations. Throughout investigations on antenna elements/antenna IF section, interfacing should be performed to determine the final input data width, range, and precision, so the design can be scaled and optimized to fit realistic operation scenarios.

REFERENCES

[1] M. Kinoshita, "DoCoMo's vision on mobile commerce," in *Proc. the 2002 Symposium on Applications and the Internet (SAINT '02)*, pp. 39–40, Nara, Japan, January–February 2002.

[2] P. Brislen, *World's First 3G Network in Trial*: IDG Communications, 2001, http://idg.net.nz/news.nsf/UNID/CC256CED0016AD1ECC256A5D0010A487?

[3] CWTS, "Physical layer—General description, TS C101 V3.0.0," 1999.

[4] J. C. Liberti Jr. and T. S. Rappaport, *Smart Antennas for Wireless Communications: IS-95 and Third Generation CDMA Applications*, Prentice-Hall, 1999.

[5] P. H. Lehne and M. Pettersen, "An overview of smart antenna technology for mobile communications systems," *IEEE Communications Surveys*, vol. 2, pp. 2–13, 1999.

[6] C. B. Dietrich Jr., W. L. Stutzman, B. K. Kim, and K. Dietze, "Smart Antennas in Wireless Communications: Base-Station Diversity and Handset Beamforming," *IEEE Antennas Propagat. Mag.*, vol. 42, pp. 142–151, 2000.

[7] S. Haykin, *Neural Networks: A Comphensive Foundation*, Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1999.

[8] C. Christodoulou and M. Georgiopoulos, *Applications of Neural Networks in Electromagnetics*, Artech House, 2001.

[9] M. J. D. Powell, "Radial basis functions for multivariable interpolation: a review," in *IMA Conference on Algorithms for the Approximation of Functions and Data*, RMCS, Shrivenham, 1985.

[10] A. H. El Zooghby, C. G. Christodoulou, and M. Georgiopoulos, "Antenna array signal processing with neural networks for direction of arrival estimation," in *1997 IEEE Antennas and Propagation Society International Symposium*, vol. 4, pp. 2274–2277, Montreal, Que., Canada, July 1997.

[11] A. H. El Zooghby, C. G. Christodoulou, and M. Georgiopoulos, "Performance of radial-basis function networks for direction of arrival estimation with antenna arrays," *IEEE Trans. Antennas Propagat.*, vol. 45, no. 11, pp. 1611–1617, 1997.

[12] A. H. El Zooghby, C. G. Christodoulou, and M. Georgiopoulos, "Adaptive interference cancellation in circular arrays with radial basis function neural networks," in *1998 IEEE Antennas and Propagation Society International Symposium*, vol. 1, pp. 203–206, Atlanta, Ga, USA, June 1998.

[13] A. H. El Zooghby, C. G. Christodoulou, and M. Georgiopoulos, "Radial basis function neural network algorithm for adaptive beamforming in cellular communication systems," in *1998 IEEE-APS Conference on Antennas and Propagation for Wireless Communications*, pp. 53–56, Waltham, Mass, USA, November 1998.

[14] A. H. El Zooghby, S. El Khamy, and C. Christodoulou, "Adaptive antenna arrays for mobile satellite communications," in *Proc. IEEE Southeastcon '96. 'Bringing Together Education, Science and Technology'*, pp. 324–327, Tampa, Fla, USA, April 1996.

[15] S. Haykin, *Adaptive Filter Theory*, Prentice-Hall, Englewood Cliffs, NJ, USA, 4th edition, 2002.

[16] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Design*, Oxford University Press, Oxford, New York, USA, 2000.

[17] K.-L. Du, K. K. M. Cheng, and M. N. S. Swamy, "A fast neural beamformer for antenna arrays," in *IEEE International Conference on Communications, 2002 (ICC '02)*, pp. 139–144, New York, NY, USA, April–May 2002.

[18] V. Gazi, *RBF Neural Network Source Code, Intelligent Control Laboratory (EE 758)*, 2002, http://www.eleceng.ohio-state.edu/ gaziv/ee758/rbf.tar.gz.

[19] Altera Corporation, *APEX 20K Programmable Logic Device Family Data Sheet*, 2001.

[20] Altera Corporation, *Nios Embedded Processor 32-Bit Programmer's Reference Manual*, 2003.

[21] Altera Corporation, *Custom Instructions for the Nios Embedded Processor*, 2002.

[22] J. W. S. Liu, *Real-Time Systems*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2000.

[23] Analog Devices, *ADSP-2100 Family User Manual*, 1995.

[24] B. Pendleton, *Doing it Fast, Fixed Point Arithmetic Techniques and Fast 3D Transforms*, 2002, http://www.gameprogrammer.com/4-fixed.html.

[25] M. Attenborough, *Engineering Mathematics Exposed*, McGraw-Hill, New York, NY, USA, 1994.
[26] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, Md, USA, 3rd edition, 1996.

William To was a postgraduate student in the Department of Electrical and Computer Engineering at the University of Auckland. He received his B.E. and M.E. degrees in 2001 and 2003, respectively, in computer systems engineering. His research interests include hardware/software tradeoffs in embedded systems implementation, field programmable gate arrays, and hardware implementation of complex algorithms.

Zoran Salcic is a Professor and holds the Chair in computer systems engineering in the Department of Electrical and Computer Engineering, the University of Auckland. His research interests are in complex digital systems architectures and design, prototyping and customization of digital systems, and new architectures and formal models for embedded systems for combined data- and control-driven applications. He has published more than 150 papers in journals and refereed conferences, seven books, and numerous technical reports. He led or took part in several dozens of research projects and complex digital and computer-based systems designs in over 30 years. He established and leads Embedded Systems Research Group at the University of Auckland. He received B.E., M.E., and Ph.D. degrees in electrical engineering from University of Sarajevo in 1972, 1974, 1976, respectively .

Sing Kiong Nguang graduated (with first-class honours) from the Department of Electrical and Computer Engineering, the University of Newcastle, Australia, in 1992, and received the Ph.D. degree from the same university in 1995. He is currently holding a Senior Lectureship in the Department of Electrical and Computer Engineering, the University of Auckland, New Zealand. He has over 60 journal papers and over 40 conference papers/presentations on nonlinear control design, nonlinear H-infinity control systems, nonlinear time-delay systems, nonlinear sampled-data systems, biomedical systems modelling, fuzzy modelling and control, biological systems modelling and control, and food and bioproduct processing.