Analysis of the Computational Requirements of a Pulse-Doppler Radar Signal Processor

R. Broich*† and H. Grobler*
*Department of Electrical, Electronic and Computer Engineering, University of Pretoria
†DPSS Radar and Electronic Warfare, CSIR, South Africa
Emails: RBroich@csir.co.za, Hans.Grobler@up.ac.za

***Abstract*— In an attempt to find an optimal processing architecture for radar signal processing applications, the different algorithms that are typically used in a pulse-Doppler radar signal processor are investigated. Radar algorithms are broken down into mathematical operations and the relative processing requirements of each operation is determined. Implementation alternatives for the operations with the highest relative processing requirements are briefly discussed for an FPGA based soft-core architecture.**

I. Introduction

In pulse-Doppler radar systems, the amount of useful information that can be extracted from the received echo's directly depends on the computational performance the radar signal processor (RSP) can deliver. Traditionally the design methodology follows a bottom-up approach, in which existing processing technologies are pieced together to form the RSP. Radar algorithms are often simplified and modified to fit the available processing architectures [1]. These simplifications are often degrading to algorithmic performance and thus to the entire radar system.

In this paper the different computational operations that are used in pulse-Doppler radar signal processing are explored, in order to find an optimal radar signal processing architecture. The theoretical radar algorithms are broken down into signal processing constructs and analysed from a computational and data-flow perspective. The relative percentage usage for each processing construct is discussed and a list of dominant operations extracted. Potential mappings to an FPGA based soft-core architecture are then investigated for these dominant operations.

II. Radar Signal Processor Flow of Operations

To determine the processing requirements of each radar algorithm, the simplified RSP in Fig. 1 will be used as a test-case. Although this is just one of many implementation variations, it represents a variety of mathematical operations that are typically found in RSPs and many of the operations also apply to the higher level data processing. This section discusses each of the RSP algorithms from a computational perspective. Rather than analysing the complexity or growth rate of each algorithm (based on big O-notation for example) a data-flow approach similar to that described by Kienhuis [2] was used to extract approximate quantitative results. Quantitative analysis is a well known technique for defining general

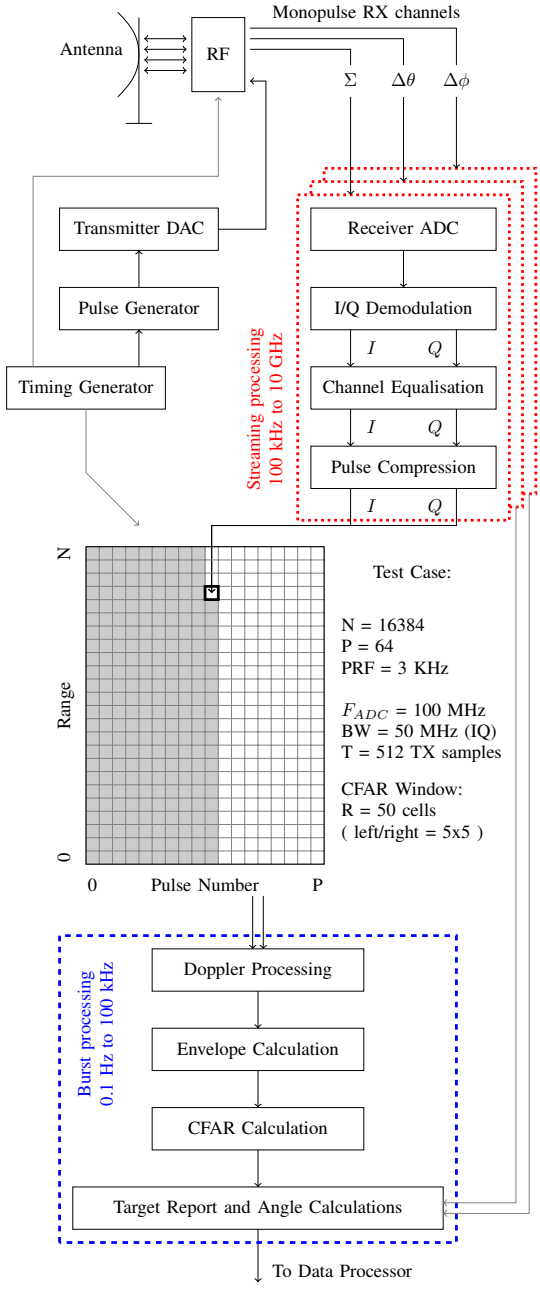

Fig. 1. Radar signal processor (RSP) flow of operations

purpose computer architectures [3]. An abstract machine, in which only memory reads, writes, additions and multiplications are considered to be significant operations, is chosen for the model of computation. For each algorithm, a pseudo-code listing is used to find an expression for the required number of additions/subtractions, multiplications/divisions, as well as memory reads and writes. Based on the parameters of the test case, the computational requirements (in millions of operations per burst of data) are listed for comparison purposes.

A. I/Q Demodulation

The IQ demodulation stage is most commonly digitally implemented as either a Hilbert transform or an in-phase and quadrature mixing operation. The mixing operations are simply digital multiplications by sine and cosine as shown below (where P is the number of pulses per burst, and N is the number of range bins per pulse).

```
loop i=0 to P-1
 loop j=0 to 2*N-1
   input_re[i][j] = input[i][j]*sin(2*pi*fIF*j/fadc)
   input_im[i][j] = input[i][j]*cos(2*pi*fIF*j/fadc)
```

Thus P×2×N×2 multiplications and memory writes as well as P×2×N×3 memory read operations are required provided that a lookup table is used for the trigonometric functions. Digital low pass filters (typically real FIR filters) are used to eliminate the negative frequency as well as DC components of both I and Q after the mixing operation. The filter output is then decimated by two to discard redundant data. The filter requirements (length L) are therefore P×N×L×2 multiplications, additions, and memory reads as well as P×N×L coefficient memory reads.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L-1
    iq_dem[i][j].RE += input_re[i][j*2-l]*lpf_coeff[l]
    iq_dem[i][j].IM += input_im[i][j*2-l]*lpf_coeff[l]
```

When the ADC sampling frequency is 4 times as high as the IF frequency, the above mixing operation simplifies to multiplications by 1,0,-1,0 and 0,1,0,-1 for the I and Q channels respectively, reducing the computational requirements to P×2×N memory reads and writes as well as P×N negations.

```
loop i=0 to P-1
 loop j=0 to N-1
   if (N=even)
     input_re[i][j] = input[i][j*2]
     input_im[i][j] = input[i][j*2+1]
   else
     input_re[i][j] = -input[i][j*2]
     input_im[i][j] = -input[i][j*2+1]
```

In this case the filtering stage can also be simplified, since only every second filter coefficient (even coefficients for I, odd coefficients for Q) is needed [4]. The computational requirements are thus reduced to P×N×L multiplications, additions, and memory reads, as well as P×N×L coefficient memory reads as shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L/2-1
    iq_dem[i][j].RE += input_re[i][j-l*2]*lpf_coef[l*2]
    iq_dem[i][j].IM += input_im[i][j-l*2-1]*lpf_coef[l*2+1]
```

Alternatively, a Hilbert transform can be used to extract the Q-channel from the real sampled data [5]. Unlike the mixing method however, the Hilbert transform method does not shift the frequency band to baseband. The FIR implementation of a Hilbert filter is essentially a wide band-pass filter (BPF) which has a constant 90-degree phase shift.

```
loop i=0 to P-1
 loop j=0 to N-1
   iq_dem[i][j].RE = input[i][j*2]
   loop l=0 to L-1
     iq_dem[i][j].IM += input[i][2*j-l]*hil_coeff[l]
```

Thus P×N×(L+1) memory reads as well as P×N×L coefficient reads, multiplications and additions are required.

The performance requirements for I/Q demodulation strongly depend on the required resolution and acceptable filter stop-band attenuation. Since most high frequency ADCs have between 10 and 14-bit resolutions, a 16-bit filter of length 32 (order 31) is usually more than adequate. The computational requirements (all figures are in millions) for the test-case are summarized in Table I.

TABLE I
I/Q DEMODULATION COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Multi	Add / Subtr	Mem Reads	Mem Writes
Standard Mixing	71	67	107	6
Simplified Mixing	34	35	69	3
Hilbert filter	34	34	68	2

B. Channel Equalisation

In practice the transfer characteristics of RF front-end components as well as ADC converters are not consistent across the entire frequency band. Additionally the gains between the different antenna channels are not matched. To compensate for such imbalances in amplitude, both the I and the Q channels are multiplied by a fractional scaling factor as shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
   amp_comp[i][j].RE = iq_dem[i][j].RE*amp_coeff
   amp_comp[i][j].IM = iq_dem[i][j].IM*amp_coeff
```

Frequency compensation typically involves a complex-valued digital filter such as the FIR filter below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L-1
    freq_comp[i][j] += amp_comp[i][j-l] * freq_coeff[l]
```

The complex multiplication in the above filter can be accomplished with 4 real multiplications and an addition as well as a subtraction. Since the input, coefficients and output is complex, 2 memory accesses need to be performed for each variable, resulting in P×N×L×4 memory reads, multiplications and additions. The computational requirements are summarised in Table II.

C. Pulse Compression

The process of pulse compression involves correlating the transmitted pulse against the received signal. This matched filter can be realised as a digital correlation or a fast convolution.

TABLE II

CHANNEL EQUALISATION COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Multi	Add / Subtr	Float / Div	Mem Reads	Mem Writes
Amplitude	0	0	2	2	2
Frequency	134	134	0	134	2

The most straightforward implementation, as derived from the correlation integral, is shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop t=0 to T-1
    pc[i][j] += freq_comp[t+j] * tx_pulse*[t]
```

Similar to the FIR filter for the frequency compensation, P×N×T×4 memory reads, multiplications and additions are required. However, since the transmit pulse length is usually longer than the above used filter lengths, the processing requirements are much more demanding.

The fast convolution method on the other hand, converts the input sequence to the frequency domain with an FFT, performs a complex multiplication by the frequency spectrum of the time-reversed complex conjugated transmitted pulse, and converts the result back to the time domain using the inverse FFT.

```
loop i=0 to P-1
  in_freq[i] = FFT(freq_comp[i])
  loop j=0 to N-1
    fil_freq[i][j] = in_freq[i][j] * FFT(tx_pulse*[-t])[j]
  pc[i] = IFFT( fil_freq[i] )
```

An FFT of length N has $\log_2 N$ stages, each consisting of N/2 butterflies. The decimation in frequency FFT butterfly can be described algorithmically as:

```
A'[m] = A[m] + B[n]*W[p]
B'[m] = A[m] - B[n]*W[p]
```

Each butterfly thus requires one complex multiplication, addition and subtraction. Thus the total computational requirements for an FFT are P×N/2×$\log_2$N×4 real multiplications and memory writes as well as P×N/2×$\log_2$N×6 real additions and memory reads. Bit inverted addressing is required for either memory reads or writes, depending on the selected DIT or DIF algorithm type. The multiplication stage requires P×N×4 multiplications and memory reads, as well as P×N×2 additions and memory writes, provided the inverted and conjugated spectrum of the transmitted pulse is precomputed. The IFFT processing requirements are similar to those of the FFT, except for an added stage that divides all output values by N and swaps real and imaginary parts of input and output [5].

TABLE III

PULSE COMPRESSION COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Multi	Add / Subtr	Float / Div	Mem Reads	Mem Writes
Digital Correlation	2 147	2 147	0	2 147	2
Fast Correlation	63	90	2	92	61

D. Doppler Processing

Doppler processing can be divided into two major classes: pulse Doppler processing and moving target indication (MTI). When only target detection is of concern, the MTI filter is usually adequate. MTI filters are often low order; even a first or second order FIR high-pass filter (such as 2 or 3 pulse MTI cancellers) can be used to filter out the stationary clutter. Higher order filter types are typically not used as they only provide modest performance improvements over the pulse cancellers [6].

```
loop i=0 to N-1
 loop j=0 to P-1
   mti[i][j] = pc[j][i] - temp
   temp = pc[j][i]
```

The processing requirements for the two pulse canceller are thus N×P×2 memory reads, subtractions and memory writes, while the 3-pulse canceller requires an additional N×P×2 multiplications and additions.

```
loop i=0 to N-1
 loop j=0 to P-1
   mti[i][j] = pc[j][i] - 2*temp1 + temp2
   temp2 = temp1
   temp1 = pc[j][i]
```

Pulse Doppler processing is computationally more demanding, but improves SNR performance and provides target velocity information. In the pulse-Doppler processing, spectral analysis is performed over the "slow-time" pulse to pulse samples representing a discrete range bin. Thus N independent FFTs over all P samples are required, each of which is preceded by a windowing function as shown below.

```
loop i=0 to N-1
 loop j=0 to P-1
   win[i][j].RE = pc[j][i].RE * w[j]
   win[i][j].IM = pc[j][i].IM * w[j]

loop i=0 to N-1
  dop[i] = FFT(win[i])
```

The processing requirements for the FFT stage are thus N×P/2×$\log_2$P×4 real multiplications and memory writes as well as N×P/2×$\log_2$P×6 real additions and memory reads. As before, bit-inverted addressing on either the input or output as well as twiddle-factor coefficients are required. The window function requires N×P×2 multiplications and memory writes as well as N×P×3 memory reads.

TABLE IV

DOPPLER PROCESSING COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Multi	Add / Subtr	Float / Div	Mem Reads	Mem Writes
2-pulse Canceller	0	2	0	2	2
3-pulse Canceller	2	4	0	2	2
Pulse Doppler	15	19	0	22	15

E. Envelope Calculation

The envelope calculation extracts the magnitude information from the complex signal. The linear magnitude takes the square root of the sum of the squares of real and imaginary parts, requiring N×P×2 multiplications and memory reads as well as N×P additions, square roots, and memory writes as

shown below.

```
loop j=0 to N-1
 loop i=0 to P-1
   env[j][i] = SQRT(dop[j][i].RE^2 + dop[j][i].IM^2)
```

Depending on the processing architecture, the square root operation may have a significant impact on performance. To simplify these computational requirements, the squared magnitude operation does not make use of the square root. Another variation takes the (linear) magnitude and scales the output logarithmically. In such a case the square root operation simplifies to a multiplication by 2 after the log operation. The computational requirements are summarized in Table V.

TABLE V
ENVELOPE CALCULATION COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Multi	Add / Subtr	Float / Div	Mem Reads	Mem Writes
Linear Magnitude	2	1	1	2	1
Square Magnitude	2	1	0	2	1
Log Magnitude	3	1	1	2	1

F. Constant False Alarm Rate

The CFAR detection process greatly improves the simple threshold detection performance by estimating the current interference level rather than just assuming a constant level. The simplest of the CFAR classes is the cell averaging (CA-) CFAR, which averages the returns from numerous reference cells (the CFAR window: R = 2×WIN_R×WIN_D cells) around the cell under test, and then uses this average in the detection process.

```
loop j=0 to N-1
 loop i=0 to P-1
   sum_left = 0, sum_right = 0
   loop l=0 to WIN_R-1
    loop k=-WIN_D/2 to WIN_D/2
      sum_left += env[j+GUARD+l][i+k]
   loop l=0 to WIN_R-1
    loop k=-WIN_D/2 to WIN_D/2
      sum_right += env[j-GUARD-l][i+k]
   if (env[j][i] > alpha * (sum_right+sum_left))
     target.RANGE = j
     target.VELOCITY = i
```

Thus N×P×(R+1) additions, N×P×(R+1) memory reads, as well as N×P fractional multiplications, and comparisons are required. The number of memory writes depends on the expected number of targets. Sliding window techniques, where previously summed columns are stored in registers and reused in the next iteration, further reduce the number of memory reads and additions to N×R + N×(P-1)×WIN_R×2 + N×P and N×R + N×(P-1)×(WIN_R×2+2) respectively.

To minimize target masking effects, SOCA-CFAR uses the smaller of the two (left and right) windows. Similarly GOCA-CFAR uses the greater of the two windows to suppress false alarms at clutter edges. Computationally these two CFAR classes are similar to the CA-CFAR with the exception of one less addition and one extra comparison for each N×P loop. The sliding window approach to SOCA and GOCA requires 2 more additions (as well as 1 extra comparison) per cell in order to keep the two window sums separate. Another variation of CFAR determines the mean of both the left and right CFAR windows, and depending on their difference, makes a logical decision whether CA, GOCA, SOCA is to be used.

Heterogeneous environments may bias the threshold estimate. The censored (CS-) CFAR discards the M largest (both largest and smallest in the case of trimmed mean (TM-) CFAR) samples in the window, making it much more suitable for such environments than the CA-CFAR. The interference power is then estimated from the remaining cells as in the cell averaging case. For the typically small values of M (between 1 and 3), a selection algorithm as shown below is well suited.

```
loop m=0 to M-1
   min = 0, max = 0
   loop d=1 to R-1
     if x[min] > x[d] min=d
     if x[max] < x[d] max=d
   delete x[min]
   delete x[max]
```

This selection algorithm thus requires M×(R-1)×2 comparisons, M×R memory reads and M×2 memory invalidations per cell in the range-Doppler map.

Similar to CS and TM-CFAR, order statistic (OS-) CFAR also requires the samples in the window to be numerically sorted. Rather than discarding samples however, OS-CFAR selects the K-th sample from the sorted list as the interference statistic. This value is then multiplied by a fractional and used in the comparison against the current cell under test. Although a selection algorithm could be used, well established sorting algorithms such as merge-sort can sort R samples with $R \times \log_2 R$ comparisons, memory writes and memory reads. Since the reference window has to be sorted for each cell in the range-Doppler map, the sorting stage is computationally demanding.

Adaptive CFAR algorithms iteratively split the window (including the test cell) at all possible points and then calculate the mean of the left and right windows. The most likely splitting point Mt that maximises the log-likelihood function based on the calculated mean values is then chosen. The final step performs the standard CA-CFAR algorithm on the data in which the test cell is located. In such a case, a sum area table (SAT) of all reference cells can be formed as shown below.

```
sum = 0
loop r=0 to R-1
   sum = sum + x[r]
   sum_area[r] = sum;
```

The sum of all cells in the windows to the left of the transition point Mt is now simply sum_area[Mt], while the sum of the right window is sum_area[R] - sum_area[Mt]. The log-likelihood function then requires 3 additions, 2 divisions, multiplications and logarithm calculations, as well as 1 memory read and write per transition point as shown below.

```
loop mt=1 to R-1
   L[mt] = (R-mt) * ln((sum_area[R]-sum_area[mt])/(R-mt)) -
           mt * ln( sum_area[mt] / mt )
```

An iteration over the (R-1) possible transition points is required to select Mt such that L[Mt] is maximised. If Mt is smaller than R/2, the left window (otherwise the right window)

is selected as the CFAR statistic. The processing requirements are summarized in Table VI.

TABLE VI
CFAR COMPUTATIONAL REQUIREMENTS

Radar Algorithm	Mul	Add / Subtr	Float / Div	Mem Reads	Mem Writes
CA-CFAR	0	55	1	53	0
CA-CFAR (SW)	0	14	1	12	0
SOCA/GOCA	0	55	1	53	0
SOCA/GOCA (SW)	0	17	1	12	0
CS-CFAR (M=2)	0	255	1	154	4
OS-CFAR	0	297	1	298	296
Adaptive CFAR	103	260	156	157	104

It is clear that the CA algorithms require the least amount of processing of the CFAR classes, especially when implemented with the sliding window optimisation. Although OS-CFAR typically performs better in the presence of interferers [7], the processing requirements are more than 33 times as high. The requirements in the previous tables are each in millions of instructions required for burst processing (one burst is 22 ms in the test-case). Fig. 2 compares the number of operations required per second in a streaming RSP for the different CFAR implementations.

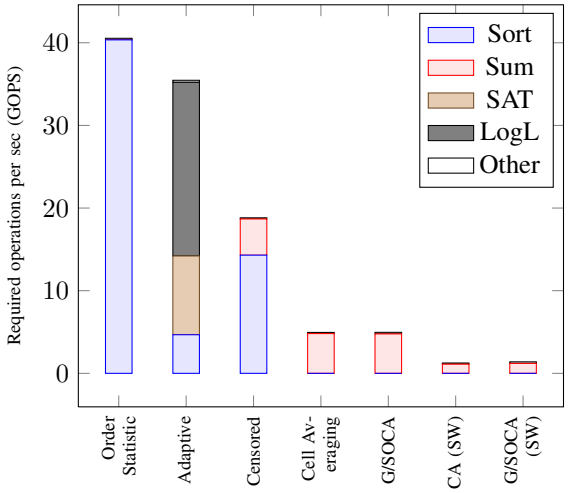

Fig. 2. CFAR processing requirements

III. Computational Breakdown into Mathematical Operations

Based on the previously discussed algorithmic requirements, the applicable mathematical and signal processing operations are established in this section. The relative importance and normalised percentage usage of these reoccurring operations are also investigated.

Fig. 2 already gives a hint at how the different CFAR algorithms are broken down into mathematical operations; the most obvious being numerical sorting as well as block/vector summation. Although functions such as the SAT and the log-likelihood function (LogL) are used mainly in the adaptive CFAR implementation, they could be used for other algorithm classes.

The remaining RSP algorithms follow a fairly natural breakdown into common mathematical / signal processing operations as shown in Fig. 3. The number to the right of

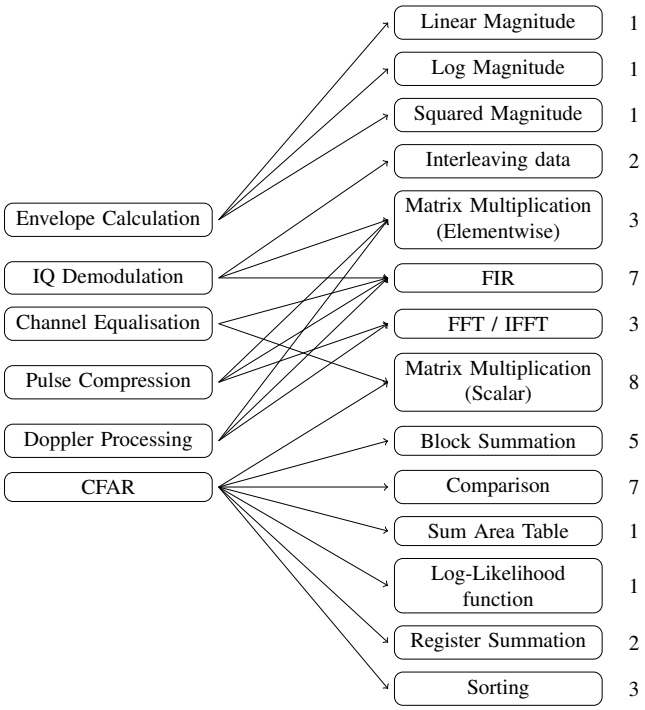

Fig. 3. Radar signal processor algorithmic breakdown

each of the mathematical operations indicates the number of algorithms that make use of that particular operation. For example, the FIR filter structure is used in all three of the discussed IQ demodulation algorithms, in two Doppler algorithms, in channel equalisation and in one of the pulse compression algorithms.

The most important of these mathematical operations need to be selected for optimisation, without giving too much importance to one specific implementation alternative. Logically only mathematical operations making up a fair amount of processing time should be optimised. To find the normalised percentage usage, the different implementation options for each algorithm are thus weighted according to their computational requirements. Algorithms with low computational requirements should be favoured over those with high requirements. For example, pulse compression can be implemented with the fast correlation or as a FIR filter structure. The FIR filter structure has a substantially larger computational requirement in terms of operations per second, but may be better suited for streaming hardware implementations, and thus cannot be neglected. Within radar algorithm groups, each implementation is assigned a percentage of total computational requirements for that group. This number is then subtracted from 100% and normalised such that the added weights of each group add up to unity. Table VII shows the normalised processing requirements for each mathematical operation across the different implementations that were discussed.

TABLE VII

RSP NORMALISED PERCENTAGE USAGE

Mathematical Operation	**%**
FIR	56.63
FFT / IFFT	22.50
Sorting	9.57
Block Sum	3.53
Log-Likelihood	3.36
Sum Area Table	1.53
Matrix Multiplication (Elementwise)	1.51
Matrix Multiplication (Scalar)	0.49
Interleaving data	0.18
Log Magnitude	0.17
Comparison	0.17
Linear Magnitude	0.16
Squared Magnitude	0.15
Register Sum	0.07

It comes as no surprise that the FIR filter and FFT operations have the highest usage. Although only used in 3 of the 7 discussed CFAR algorithms, the computational requirement of the sorting operation indicates it's importance for optimisation purposes. Figure 4 graphically confirms the above results for 5 different RSP implementation alternatives. All five alternatives make use of amplitude and frequency compensation, the 3 pulse canceller, pulse-Doppler processing and the linear magnitude envelope calculation. The large 'other' block in option 3 primarily consists of the log-likelihood function. It is interesting to note that the summation operation does not require much resources compared to the FIR filter, FFT and sorting operations.

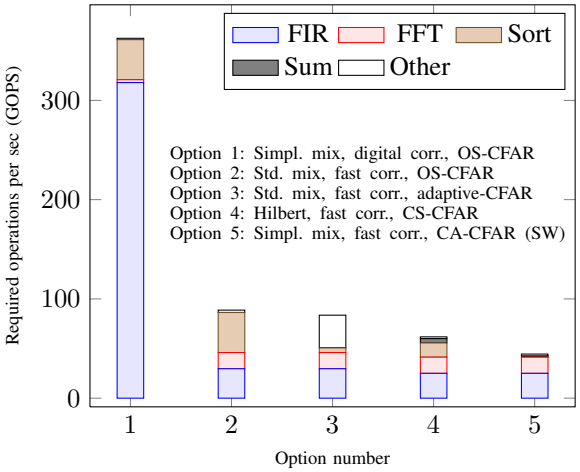

Fig. 4. Complete RSP computational requirements

IV. IMPLEMENTATION OPTIONS FOR THESE OPERATIONS

From an implementation perspective, the direct form FIR filter structure maps well to a hardware implementation (such as the FPGAs logic blocks), capable of producing an output sample every clock cycle. Purely sequential implementations (such as DSPs or general purpose CPUs), on the other hand, require numerous iterations over the array for a single output sample. By merging the architectures of both the sequential processor and the hardware implementation, flexibility and performance can be balanced in such a custom soft-core processor. Coefficients could be preloaded into a memory block, and input samples can be shifted into the hardware based FIR filter structure directly from the custom processor. This structure can then be used for numerous mathematical operations such as FIR filters, block summation (when the coefficients are preloaded to 1), the sum area table (with 0 initial conditions), CA-CFAR windows (guard cells can be excluded by setting their coefficients to 0), pulse cancellers, digital correlation, and digital convolution.

The other dominant operation, the FFT, dictates requirements for both small length FFTs of 128 or less for pulse-Doppler processing, as well as extremely large sizes across the entire range line for pulse compression or fast filtering. FFT co-processors however typically only support a limited range of lengths. For purely sequential implementations, DSP manufacturers often include radix-2 butterfly instructions to speed up the implementation. Since decimation in time or frequency algorithms permit FFTs of length N to be computed with two FFTs of length N/2, a hybrid soft-core implementation consisting of several smaller memory mapped FFT-coprocessors could be software coordinated to achieve larger lengthed FFTs. Optimisations such as these can improve the performance as well as simplify the programming model from a hardware descriptive language (HDL) design to a library based software environment, permitting quick evaluation of new radar algorithms during field-tests.

V. CONCLUSION

This paper presents an overview of the computational resources required for a pulse-Doppler radar signal processor. Radar algorithms are broken down into signal processing constructs, and ranked according to their importance and computational requirements. Although FIR filters and FFTs are by far the most important operations, sorting and block summation constructs make up a substantial share of the total processing requirements. Based on the results of this study and in particular having identified the operations that dominate, one can now define an architecture that maximises performance for the given algorithms whilst providing flexibility.

REFERENCES

[1] D. R. Martinez, R. A. Bond, and M. M. Vai, *High Performance Embedded Computing Handbook*. CRC Press, 2008.

[2] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors, IEEE Int. Conf.*, July 1997, pp. 338 –349.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2002.

[4] K. Teitelbaum, "A flexible processor for a digital adaptive array radar," *Aerosp. and Electron. Syst, IEEE*, vol. 6, no. 5, pp. 18–22, May 1991.

[5] R. G. Lyons, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Mar. 2004.

[6] M. A. Richards, *Principles of Modern Radar: Basic Principles*, 1st ed. North Carolina: SciTech Publishing, 2010.

[7] ——, *Fundamentals of Radar Signal Processing*, 1st ed. New York: McGraw-Hill, 2005.