ANALYZING THE ADVANTAGES OF RUN-TIME RECONFIGURATION IN RADAR SIGNAL PROCESSING

Dennis Johnsson, Anders Åhlander, and Bertil Svensson
Centre for Research on Embedded Systems, Halmstad University
PO Box 823, SE-30118 HALMSTAD
Sweden
Dennis.Johnsson@ide.hh.se, Anders.Ahlander@ide.hh.se, and Bertil.Svensson@ide.hh.se

ABSTRACT
Configurable architectures have emerged as one of the most powerful programmable signal processing platforms commercially available, obtaining their performance through the use of spatial parallelism. By changing the functionality of these devices during run-time, flexible mapping of signal processing applications can be made. The run-time flexibility puts requirements on the reconfiguration time that depend both on the application and on the mapping strategy. In this paper we analyze one such application, Space Time Adaptive Processing for radar signal processing, and show three different mappings and their requirements. The allowed time for run-time reconfiguration in these three cases varies from 1 ms down to 1 µs. Each has its own advantages, such as data reuse and optimization of computational kernels. Architectures with reconfiguration times in the order of 10 µs provide the flexibility needed for mapping the example in an efficient way, allowing for on-chip data reuse between the different processing stages.

KEY WORDS
Reconfigurable architecture, radar signal processing, configuration time.

1. Introduction

Some high-speed digital signal processing applications require real-time performance well beyond what can be achieved with commercial microprocessors. Configurable parallel processors bear promise to combine programmability with the required performance.

The signal processing in future-generation Active Electronically Scanned Array Antenna (AESA) based radar systems is one such challenging problem. One of the main advantages of the AESA approach is the capacity to apply Space-Time Processing, STP, to the received signal data. The multiple sub-antenna apertures distributed over the antenna array make it possible to extract not only temporal, but also spatial information, e.g. the direction of incoming signals. By estimating the arrival directions and other properties of unwanted signals in the sidelobes, the signal processing can suppress these by applying complex-valued weights to the signals from the antenna channels. This type of radar signal processing is generally referred to as Space-Time Adaptive Processing (STAP) [1][2][3].

The signal processor in an AESA based radar may be required to process data from hundreds of antenna elements, which means that the required performance for the key operations can reach hundreds of GFLOPS. Moreover, especially in airborne applications, there are additional demands for small space and low power consumption. There are also new, emerging applications for STAP, such as performance enhancement in GPS positioning [4] and wireless communication [5]. Energy consumption and cost are important issues also in these new (high volume) applications.

The signal processor should be capable of efficient scaling in different dimensions, such as the number of input channels and the types of algorithms. There is a need for new parallel computing structures and application development environments to cope with the demands of these systems. The choice of computing structure comprises everything from multi-node system level issues down to the internal function of the individual computer nodes. Moreover, in many cases the radar platform should be able to perform several different tasks, which leads to requirements on flexible solutions. These demands often call for a programmable solution where fixed function ASICs, despite superior performance, do not meet the requirements.

Configurable solutions can reach performance that is one to two orders of magnitude higher than what conventional microprocessors can achieve [6]. Furthermore, if run-time reconfiguration is an option there is flexibility in when and where the computations take place. In this paper we explore how this flexibility can be used in a STAP algorithm. We do not propose a specific reconfigurable architecture, but derive the requirements on reconfiguration time for implementing the algorithm. We also show that choosing to reconfigure more frequently opens up for more reuse of data on chip, which in turn reduces the I/O bandwidth demand.

The organization of this paper is as follows: We start by introducing the algorithm used in this study, showing
what the computations are and how the data set that these computations operate on look like. We also show the performance requirements. In the next section we give a short description of different reconfigurable architectures and state the assumptions for this study. Thereafter we show three different cases of how reconfiguration could be used in order to compute the STAP algorithm. We end the paper with the results, a discussion and conclusions.

2. Radar Example

In its basic operation the STAP algorithm uses the received data to estimate the clutter and jammer interferences and, from this estimation, computes a set of weights that is applied to the data. As the estimation is based on the data itself, the algorithm is categorized as adaptive. In most radar algorithms there are a number of computations that are common to many algorithms. Examples of such computations are convolution and/or fast convolutions in preprocessing stages, fast Fourier transformations (FFTs) and matrix operations such as matrix multiplication, matrix-vector multiplication, and QR-decompositions. We refer to these as functions that should be performed on a dataset. These functions are then assumed to run as tasks on compute nodes.

This article uses the RT_STAP benchmark application [7], developed by MITRE Corporation for the US Air Force, as a basis for analysis of benefits and costs of different runtime reconfiguration strategies. The benchmark specifies three cases with an increasing level of computational complexity. The hardest one is a 22-channel, 3-order Doppler-factored STAP, requiring approximately 40 GFLOPS. The indata per channel is 64 pulses and 480 rangebins per pulse, collected over a period of 32.25 ms; this constitutes a coherent processing interval (CPI).

The RT_STAP calculations, illustrated in Fig. 1, mainly consist of the steps below.

1. **Preprocessing.** This stage comprises video-to-I/Q conversion, array calibration and pulse compression. Real integer data from A/D converters (“video”) is converted to complex floating point format and collected into a so called CPI cube with the dimensions $L \times P \times N$ (channels x pulses x range samples). The complex numbers represent the in-phase and quadrature-phase parts of the received signal. For each pulse and channel, fast convolutions apply the combined operation of array calibration and pulse compression along the $N$-dimension (over the range samples). The $N$ samples per pulse are decimated, by a factor $D$, to $N_D$ samples per pulse.

2. **Doppler processing.** For each range bin and channel, an FFT of length $K$ is applied in the $P$-direction to extract velocity information, turning the pulses into Doppler channels. The CPI cube now takes the size $L \times K \times N_D$. Here $K = P$. The outdata cube from this stage is used as indata in both Step 3 and Step 4 (therefore it has to be buffered until its use in Step 4).

3. **Weights computation.** A set of weights is computed for each Doppler channel and for a number of non-overlapping intervals of rangebins for each velocity bin. Each interval is $N_R$ rangebins long. Here $N_R = N_D / 2$, i.e., two intervals per Doppler channel. This calculation step, which is the most demanding one, consists of the totally dominating QR decomposition followed by a weight extraction computation.

4. **Weights application.** The last step in this algorithm is the application of the weights computed in Step 3. The weights are applied on the data produced in Step 2. This is done with matrix-by-vector multiplications.

![Fig. 1. CPI data cube processing in a 22-channel STAP example. The different functions that operate on the data in a CPI are indicated, as well as the performance requirements](image)

3. Run-time Reconfigurable Architectures

Configurable architectures are increasingly used for signal processing applications. Sometimes they act as ASIC replacements in systems accelerating a specific function, sometimes as a stand-alone system for the complete application. The most common technique is field programmable gate arrays, FPGAs. These have very fine-grained configurable elements, which gives great flexibility for implementing arbitrary logic functions. However, this fine granularity makes implementation of arithmetic operations rather costly, both in space and time. Therefore, many FPGA manufacturers, such as Xilinx, Altera, and Lattice, implement common arithmetic functions directly in hardware, making these much more efficient than if implemented using configurable elements.

Run-time reconfiguration of FPGAs has been explored in e.g. [8][9]. However, the long reconfiguration times of today’s commercial FPGAs as well as the large configuration sizes, make commercial FPGAs more suited for infrequent reconfiguration, as when initiated by human interaction.

Even if FPGA is mainstream technology and has proven successful in many applications, there are cases where they are not efficient for reasons of cost, power consumption, or flexibility. Therefore, there are many proposals for alternative reconfigurable architectures which are targeted for word-length operations as in signal processing applications. These often include capabilities to reconfigure at run time. There exist several academic proposals for run-time reconfigurable architectures, e.g. [10].
In this paper we use a coarse classification of configurable architectures. At one end of the spectrum there are the FPGA architectures with very fine-grained configurable objects. In the middle we have reconfigurable arrays of word-length operations (typically ALUs) with limited control, as in architectures from PACT Technologies [11], IPFlex [12], and MathStar [13]. At the other end we have arrays of program-driven processors with reconfigurable interconnection, as in processors from PicoChip [14] or as in the MIT RAW processor [15]. For the purpose of this paper we focus on the middle of the spectrum where we assume a global controller that handles reconfiguration of an array consisting of ALU-sized objects. These architectures are interesting since the signal processing applications we consider mostly consist of a small set of signal processing functions operating on streaming data with fixed word-lengths. This means that we believe that the control requirements of these algorithms are such that there is no need for program-driven processors in the array.

4. Mapping Strategies on Run-time Reconfigurable Arrays

In a radar signal processing chain there are a number of different functions operating on the data in one CPI, one after the other. The goal is to have efficient implementations of these on a reconfigurable architecture and then operate these in a sequence by timesharing the reconfigurable hardware. Sometimes the algorithm needs different types of computations on different parts of the dataset. In these cases a general type of processing element that can handle the different cases can be built, but this can lead to inefficient use of the hardware. By using a reconfigurable architecture with the possibility to quickly swap configuration it is possible to design different optimized processing kernels and switch between these. It can also be easier to design several distinct processing kernels, and schedule these kernels, than to design a general processing kernel for the whole computation.

Since, in the STAP case, the number of tasks is limited, there is an opportunity to use a run-time reconfigurable architecture in this way with good efficiency. We calculate the requirements on the reconfiguration times assuming a reconfigurable architecture that is time-shared between the functions.

The algorithm processes data blocks organized in a three-dimensional volume. The processing takes place in different directions of this volume, using vectors or matrices taken from the volume, as indicated in Fig. 1. By assigning a configuration to each basic function of the algorithm we can derive different schedules that either use as few configurations as possible or try to reuse data while it is on chip.

We show three cases based on different configurations operating on the data set. In the first case configurations operate on the full data volume collected during a CPI. The second case splits the data set in a CPI into smaller parts, enabling successive computations to be performed on smaller portions of data. This can improve performance through reusing on-chip data between the computations. These two schedules, called case $a$ and case $b$, are described below under the heading “task level scheduling”. In the section thereafter, QRD mapping, we take the data partitioning one step further and call that case $c$. We look at the most demanding computation, the QR decomposition, to see if there is any gain in using run-time reconfiguration within the implementation of the QRD task.

4.1 Task level scheduling

In the illustrations below we use C1, C2 etc. to denote different configurations. Fig. 2 shows the first schedule of computations, case $a$, where tasks operate on the full data matrix before reconfiguration is done for the next task. The figure also shows the size of the data set that enters and exits each configuration. This case has the fewest reconfigurations but does not attempt to reuse on-chip data. The data computed in the Doppler processing stage is used by both the weight computation and the weight application stage.

![Fig. 2. Case a: Configurations C1 to C5 operate on the full data volume in a sequence](image)

Fig. 3 shows another schedule of the same computation, where each configuration operates on a smaller part of the data set (case $b$). The difference between case $a$ and case $b$ is that configurations 3 to 5 operate, in sequence, on a smaller portion of the data. In order to process the complete data set this sequence has to be repeated. There are in total 128 weight matrices that should be computed and applied to the indata. This results in more tasks and requires more reconfigurations but has the potential to reuse on-chip data. In Fig. 3 the size of the smallest data set needed for each task is shown. The figures show that it is reasonable to assume that the data needed for the most demanding computations in the weight computation and weight application stage could stay on chip. The in- and output data together with intermediate results is less than 324 KB (not including any temporary storage within configurations C3, C4, or C5). Keeping data on chip has the double advantage of both reducing power consumption and increasing processing performance. The reduced power consumption comes from the reduced number of off-chip memory accesses. Performance can be increased since the available on-chip memory bandwidth can be several orders of magnitude
higher than the off-chip memory bandwidth. This can increase the parallelism that can be efficiently used in the computations. As a consequence of operating on a smaller data set the latency of the results is also decreased.

**Fig. 3. Case b: Operating on smaller parts of the data set in order to reuse on-chip data**

### 4.2 QRD Mapping

In the previous cases we have assumed that the configurations correspond to the individual functions that constitute the whole STAP algorithm. However, there might be cases when it would be beneficial to partition these individual algorithms. This is what we study in case c. The partitioning can be done in different dimensions in time and space: It can be based on splitting the data into blocks, or on the fact that the algorithm has different computations on different parts of the data set, or on the fact that the computation consists of different distinct phases. In this case we look at the QRD and propose a partitioning of a known systolic solution. This partitioning of the data is motivated by the structure of the systolic QRD solution, where there are two different types of systolic cells. By building the systolic processor on a run-time reconfigurable array the size and function of the systolic processor can change during the execution of the algorithm.

By using a cut and pile strategy of the algorithm/array we can derive a suitable sub-problem that can be directly mapped on the target architecture. The triangular structure of the QRD-array contains two different types of computations which can be implemented in different configurations. All configurations do not need to contain the same number of cells from the fully parallel systolic array. In Fig. 4 we illustrate the triangular structure of the computations and the two different configurations that we use in case c.

The main motivation for using run-time reconfiguration in this case is that the hardware is better utilized if the different systolic-array cells are implemented in separate configurations. In the QRD based on Givens rotations the round cells in Fig. 4 on the diagonal of the array contain division and costly square root computations. The square cells in Fig. 4 just implement multiply and add operations. Even if there are modified versions of the QRD without the square root, the difference in latency and area of the round and square cells can motivate to separate them in order to achieve as high utilization as possible. The number and size of buffer memories and result storage depends on the size of the problem, the size of the configuration, and also on the schedule of configurations.

**Fig. 4. The triangular structure of a QRD computation is shown to the left. t1 to t7 illustrate a possible schedule of the two configurations shown to the right**

### 5. Results

In order to have a flexible mapping of the STAP computations we assume that each step or basic function in Fig 1. is at least one configuration. This means that, in order to process the data in a CPI, there are at least 5 configurations, as shown in Fig. 2. With the assumed CPI time there are about 32 CPIs per second (32.25 ms period). Furthermore, we need to put a cost to the reconfiguration. In this paper we choose to only look at the time cost for reconfiguration, and we define it as a percentage of the overall time needed for the computations. As a quite aggressive assumption we state that the configuration cost for the STAP processing is allowed to be 10 percent of the total processing time. Fig. 5 illustrates the three cases described above. In Table 1 we list, for each case: the number of times each configuration is used (per CPI), the total number of reconfigurations per CPI and per second, the I/O bandwidth needed both per CPI and per second, the number of unique configurations, and the requirements on the reconfiguration times.

In case a, each CPI uses 5 configurations and there are 32 CPIs per second. Thus there are 160 reconfigurations per second. If these are allowed to use 10% of the total time it gives a maximum of about 0.6 ms per reconfiguration. Input to each configuration is an entire data cube produced by the previous configuration. In the case of the weight application the input is a data cube from the Doppler processing and a set of weights from the weight extraction stage. If all in- and output data sizes are summarized for the computations in a CPI there are in total 50.4 MB of data. This results in a data bandwidth requirement of 1.6 GB/s.

In case b, the first two stages of CPI processing use 2 configurations, the three last stages use 3 · 128 = 384
configurations. This results in 12352 reconfigurations per second, allowing for about 8 µs in reconfiguration time. If intermediate results between the three last configurations are kept on chip the I/O demands decrease from 50.4 to 32.4 MB, requiring a bandwidth of 1 GB/s.

In case 'c', the QRD computation is divided into two different configurations. The number of reconfigurations in the QRD is dependent on the size of the kernel, i.e., the parallelism the kernel uses, and the size of the resulting matrix. We assume a 4 x 4 kernel computing 4 rows of the resulting matrix. The total number of rows and columns in the result matrix is \( 3 \cdot L = 66 \). If each kernel computes results for 4 rows, and there are two kernels per row for all but the 4 last rows, there are in total \( (3L)/4 \cdot 2 - 1 = 33 \) reconfigurations for a QRD. The rest of the computations are as in case 'b'. For each individual matrix there are 33 + 2 configurations, and there are 128 such matrices per CPI. The processing of one CPI is then done by \( 2 + 128 \cdot (33+2) = 4482 \) configurations. For the 32 CPIs processed each second there are in total 143424 configurations. The allowed reconfiguration time is then 700 ns.

![Diagram of the three cases used in this study. Each case represents finer partitioning of the data than the previous one and, as a consequence, higher demands on short reconfiguration times.](image)

**Fig. 5.** The three cases used in this study. Each case represents finer partitioning of the data than the previous one and, as a consequence, higher demands on short reconfiguration times.

<table>
<thead>
<tr>
<th>Table 1. Run-time reconfiguration requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#C1/CPI</code></td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

6. **Discussion**

Based on the numbers in Table 1 we can conclude when it is suitable to use a run-time reconfigurable architecture and what the advantages and disadvantages are when choosing one case over another.

Case 'a' is intended to serve as the starting point for using a run-time reconfigurable architecture in the STAP. It has the least demands for fast reconfigurations, but has the longest latency. Each algorithm works on the whole data set in a CPI, which makes it hard to take advantage of data locality between the different functions. Thus the bandwidth demand is higher.

Going from case 'a' to case 'b' has the purpose of enabling data reuse between functions, enabled by data locality, but it also has the advantage of offering lower latency. The cost for this is a demand for faster reconfiguration times. Compared to case 'a', case 'b' has a better potential for speeding up the whole computation of weight calculation and weight application, since executing these, one after the other, on the smallest possible dataset required for the computations can make it possible to have the dataset on chip. This is important if the bandwidth difference between off-chip and on-chip memory access is large.

Going from case 'b' to case 'c' serves as an example of how to partition the data further within an algorithm which comprises different types of operations on different parts of the data set. It can offer further potential for better hardware utilization, by only having the required functions on chip at any given time instance. Going from case 'b' to case 'c' does not reduce the latency, and it requires very fast reconfiguration times. The benefit is the potential for improving the utilization of the configurable elements since each configuration is tuned to the specific computation.

The reconfiguration times for the different cases range from around 1 ms in case 'a' down to less than 1 µs in case 'c'. All of the outlined cases require the use of less than ten different configurations, which suggest that there is not a demand for storing a large number of configurations. As with processors for signal processing, in which it can be hard to take advantage of data caching while instruction caching is beneficial, data transfers in a configurable architecture should be managed by the configuration (program) while the configuration (instruction) sequencing could benefit from using cache techniques.

7. **Conclusion**

We found that run-time reconfiguration can be used in different ways, each with its own merits and demands on reconfiguration times. The demand on reconfiguration times varies from less than a microsecond to a millisecond. Today, there exist architectures which can satisfy the requirements on reconfiguration times in all the outlined cases. However, the fine-grained architecture of
the most common technology today, FPGA, is not suitable as a configurable platform for the described run-time reconfiguration scenarios.

For our STAP case the most interesting mapping would be case b, where reconfiguration is used between the basic algorithms and where intermediate data can reside on chip. For the STAP case an architecture should be able to be reconfigured within 10µs and be able to store input and intermediate data, amounting to less than 324kB. These requirements are not unrealistic, as shown by architectures from several companies, such as the XPP from PACT [11] or the DAPDNA from IPFlex [12].

The analysis indicates that coarse-grained reconfigurable architectures should be used because of their short reconfiguration times, but they are also favorable because of their smaller configuration sizes. Having smaller configuration sizes makes it possible to make use of a configuration cache with a few configurations stored on chip without consuming too much of the on-chip memory resources. Our study shows that less than ten configurations are enough for mapping the STAP in three different ways.

References


