International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'04, pp. 177–183, Las Vegas, Nevada, June 2004.

Two-level Reconfigurable Architecture for High-Performance Signal Processing

Dennis Johnsson, Jerker Bengtsson, and Bertil Svensson
Centre for Research on Embedded Systems, Halmstad University,Halmstad, Sweden
Dennis.Johnsson@ide.hh.se, Jerker.Bengtsson@ide.hh.se, and Bertil.Svensson@ide.hh.se

Abstract

High speed signal processing is often performed as a pipeline of functions on streams or blocks of data. In order to obtain both flexibility and performance, parallel, reconfigurable array structures are suitable for such processing. The array topology can be used both on the micro and macro-levels, i.e. both when mapping a function on a fine-grained array structure and when mapping a set of functions on different nodes in a coarse-grained array. We outline an architecture on the macro-level as well as explore the use of an existing, commercial, word level reconfigurable architecture on the micro-level. We implement an FFT algorithm in order to determine how much of the available resources are needed for controlling the computations. Having no program memory and instruction sequencing available, a large fraction, 70%, of the used resources is used for controlling the computations, but this is still more efficient than having statically dedicated resources for control. Data can stream through the array at maximum I/O rate, while computing FFTs. The paper also shows how pipelining of the FFT algorithm over a two-level reconfigurable array of arrays can be done in various ways, depending on the application demands.

Keywords: *Signal processing, reconfigurable array, dataflow*

1. Introduction

Many advanced signal processing applications put hard requirements on the embedded computer architecture. These requirements manifest themselves as high compute requirements, limited power consumption, small space, low cost, etc. This means that there has to be efficient signal processing architectures. The use of dedicated hardware is common in these applications, but more flexible solutions are sought for. Rather than going for the acceleration of one specific calculation one should identify domains of applications and design an architecture that supports this entire domain. This enables building domain specific architectures rather than application specific architectures. Further, by the use of reconfigurable hardware it is possible to accelerate larger parts of the algorithms than with a fixed computer architecture.

In this paper we outline an architecture targeted for multi-dimensional signal processing. It is characterized by its two abstraction levels: the macro-level and the micro-level. In order to make efficient processing possible in a dynamic way, reconfiguration of the hardware is allowed on both levels. On the macro-level, the architecture can take advantage of the available function and high-level pipeline parallelism in the applications. On the micro-level, the massive, fine-grained data parallelism allows efficient parallel and pipelined computations of algorithms such as FFTs, FIR filters, and matrix multiplications.

The micro-level is characterized by regular, highly parallel dataflow and repeated calculations. Thus, in structures for this level, it should be possible to allocate most of the hardware resources to the parallel data path and only minor parts to the control of the computations. Since this division of tasks is application dependent, it is interesting to study the use of architectures where the same hardware resources can be used both for data processing and for the control flow. Therefore, in this paper, we study the implementation of an FFT on a reconfigurable array of arithmetic logic units (ALUs) that can be used for either of these tasks. The goal is, of course, to use as much as possible of the available resources for the data path.

The rest of the paper is organized as follows: First we briefly describe the characteristics and demands of multi-dimensional signal processing and argue that a two-level architecture is an efficient way to achieve the required flexibility and computational power. We then discuss various micro-level structures and in particular the more coarse-grained, word-level reconfigurable array architectures. We describe an implementation of an FFT on one such, commercially available array — the XPP. We find that, in our implementation, a larger portion of the array is actually used for the flow control than for the actual butterfly operations in the FFT. Still, it can be argued that the array is actually more area-efficient than a more conventional solution. The work of the control part roughly corresponds to the task of the program code and control logic in a microprocessor or digital signal processor. Whether the ratio between the two parts could be changed by al-

terations to the architecture or by more efficient mapping is still an open question.

We conclude the paper by illustrating how signal processing applications can be mapped onto the architecture on the macro-level, including how an FFT computation can be mapped onto a pipeline of processor arrays in order to exactly match the requirements of the application.

2. Structures for high-performance embedded signal processing

In high-performance embedded signal processing it is crucial that the architecture exploits the available parallelism in an efficient way. Parallelism exists on different levels — from the macro-level function parallelism down to the micro-level, fine-grained data parallelism. The applications considered here use multi-dimensional signal processing on streaming data. A typical example is phased array radar signal processing. In these applications, functions such as transformations and linear operations are applied on the input data volume. The functions are performed in succession, implying that a natural approach for parallelization of the application is to arrange a pipeline of function blocks. Since the applications considered here are multi-dimensional it also means that, for each dimension, the same sub-function is applied in parallel to all elements in that dimension. Of course, this makes it possible to execute the same function on different dimensions in parallel. The type of functions could be matrix by vector multiplications, FIR filters, FFTs, etc. These functions themselves have inherent data parallelism. Figure 1 shows an example of a data volume in a radar application. The transformations are made on vectors along different dimensions of the volume and after each other. The size and shape of the volume can vary depending on the task performed by the radar. Moreover, radar tasks — and hence data volume shapes — may change during operation.

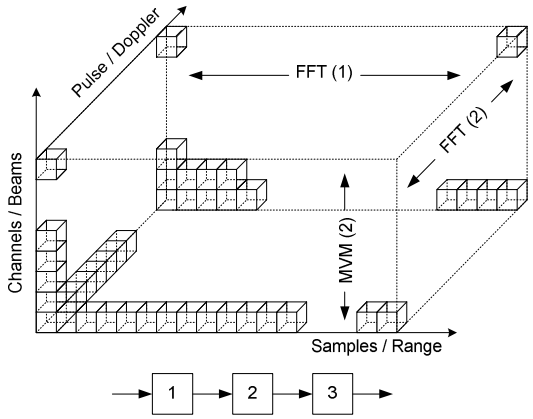

Figure 1. A multi-dimensional signal processing example. Functions are performed in different directions and in succession.

The applications also have real-time demands, both in terms of high throughput and in terms of latency. As a consequence of the changes in radar tasks these requirements also change. Different tasks have different requirements on throughput and latency.

An efficient architecture for embedded high speed signal processing is characterized by quantitative measures such as high compute density, both regarding space (operations/cm^3) and power (operations/W). Other, more fuzzy, metrics could be flexibility or how much general-purpose the architecture is. The architecture should offer more flexibility than those built with ASICs but it does not require the generality of a system consisting of one or more general purpose processors. The question is then what the required flexibility is for a certain domain of applications. The architecture needs to manage parallel data paths. The control of these data paths could be made in different ways. The most flexible — but least efficient — way is to have one control program for each data path, as in a microprocessor. Using one program for a group of data paths, as in a SIMD processor, or a more static mapping of the control, as found in configurable architectures, offers significant efficiency gains. These tradeoffs can be investigated by actual implementation studies on different platforms. Another vital aspect, not touched upon here, is how to develop applications for an architecture, i.e., what tools there are to support the application developer.

Demanding application domains, such as radar signal processing, need highly parallel architectures. Advanced radar systems may require computing 10^{12} arithmetic operations per second, with constraints on both space and power usage [1]. These architectures should be scaleable and support the computation of applications with highly different characteristics, e.g. different sizes and shapes of the data volume, and different demands on throughput and latency. In order to efficiently map the applications we envision mapping of functions onto a parallel architecture where, for each function, a node or group of nodes are responsible for executing the function. The nodes should then implement the calculations of the functions in an efficient way. The architecture should provide for static mapping of the functions and the dataflow, but also provide means to quickly, during runtime, reconfigure the mapping and communication pattern to adapt to a different scenario, e.g., changing the data set shape and modifying the signal processing functions.

Embedded high-performance signal processing applications often require complete systems consisting of multiple interconnected chips and even boards. It is therefore important that the architecture allows scaling. In order to ease the integration the same structure should be applied repeatedly. We believe a suitable architecture is a two-level configurable architecture. The first level, the macro-level, is the target for mapping of functions. The dataflow between nodes is configured on this higher level, and functions are assigned to one or a group of nodes depending on the requirements. System configuration on the macro-level should support run-time re-allocation of nodes to functions.

An efficient architecture could be built using a mesh of nodes. A mesh architecture is scalable to larger structures and is also favorable from an implementation point of view. Functions could be mapped to the mesh in different ways depending on the requirements of the application. The nodes in the mesh should then, on the micro-level, execute the functions as efficiently as possible, meaning that the low-level data parallelism within the functions should be exploited in the most efficient way. Figure 2 illustrates this two-level approach.

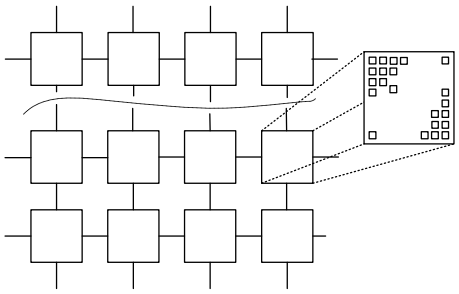

Figure 2. A two-level reconfigurable architecture.

3. Micro-level structures

The primary issue regarding the design of the micro-level architecture is to find a proper structure that offers a sufficient degree of flexibility to exploit the data parallelism found within the different functions. Microprocessors and DSPs offer full flexibility but do not have the performance obtainable with ASICs or configurable architectures [2]. ASICs offer the best performance regarding compute density and power consumption and are often used for high-end signal processing when DSPs fail to meet the requirements. However, ASICs are usually highly adapted for a specific problem and therefore not very flexible. Another technology that has evolved and, in recent years, matured to become a competitive alternative for application specific solutions is the field-programmable gate array (FPGA). FPGAs have been shown to be performance effective; however, fine grained FPGA circuits tend to require large portions of logic to implement common operations like multiplication on word length data. Major FPGA manufacturers such as Xilinx and Altera try to reduce this drawback by embedding dedicated multiplication units within the fine grained logic. Conclusively, our desire would be to find a programmable micro-level architecture that correlates well with algorithm requirements within the domain of signal processing, and which is flexible enough to exploit data parallelism within functions but also allows swift reconfigurations between functions within an application.

When choosing architecture for the micro-level there are a number of aspects that have to be addressed. What are the functional requirements on the nodes? Is it possible to take advantage of the limited range of algorithms that need to be implemented? Is it possible to do without a general microprocessor and, if so, how will the control of the algorithms be implemented? Coarse grained reconfigurable architectures offer the possibility of implementing control flow through run-time reconfiguration, or use static configurations of the control flow when possible. If a reconfigurable architecture is used there are questions of how efficient the resource usage is, how the reconfiguration is done and how fast it is.

Further, the I/O bandwidth is often an issue with all signal processing architectures. The micro-level should have a balanced architecture with I/O bandwidth matching the compute performance. The memory requirement also has to be considered. What is the proper balance between on-chip memory and computing resources?

Word-level reconfigurable architectures represent one class of candidates for implementing high-performance multidimensional signal processing. The functional, control, I/O and memory issues of these architectures need to be studied in relation to the application domain.

3.1. Coarse-grained reconfigurable architectures

Besides programmable solutions like FPGA, which are considered fine-grained architectures, there are many other configurable alternatives today. Many companies and academic research groups have developed coarser grained, programmable processor concepts. The processing elements in these span a quite wide spectrum from simple ALUs [3][4] to pipelined full scale processors [5][6][7]. Two instances of these different coarse-grained paradigms are the XPP processor from PACT XPP Technologies and the RAW processor developed at MIT.

The RAW processor is in principle a chip multiprocessor, constituting an array of 16 full scale MIPS-like RISC processors called tiles. Characteristic for RAW is the high I/O bandwidth distributed around the chip edges and the dynamic and static networks that tightly interconnect the processors, directly accessible through read and write operations in the register file of the processors. RAW has been implemented in a 0.15 μm process running at a worst case clock frequency up to 225 MHz. The area is reported to be 256mm^2 [7]. The RAW processor with its 16 tiles has a maximum peak performance of 3.6 GOPS. Large portions of a RAW tile constitute instruction fetch and decode units, a floating point unit and units for packet oriented interconnection routing.

The XPP, on the other hand, constitutes a 2D array of word oriented, single instruction-depth processing elements. The processing elements in the XPP are fixed-point ALU units without instruction or data memory. This means that instructions are statically mapped to processing elements and all control functions must be implemented using these. The instruction sequencing as done in a microprocessor must be performed by mapping instructions statically to the XPP array and/or using dynamic recon-

figuration of the array. Unlike RAW, which have multiple, full-fledged packet routed interconnection networks, XPP ALUs are connected via statically switched networks.

The XPP core has been implemented in a 0.13 μm process running at 64 MHz and the area requirement was reported to be 32 mm^2. This roughly corresponds to a peak performance of 4.1 GOPS for the 64 ALU XPP array [8]. We are not aiming at comparing the two specific architectures. Rather, we see them as examples of two architecture classes: one chip multiprocessor with dedicated control and program sequencing circuitry (RAW), the other one an ALU array (XPP) with resources only for pure dataflow processing. In the latter case, some of the processing resources need to be used for control, and we are interested to see how large portion is needed.

It is probably easier to come close to the peak performance in the chip multiprocessor. On the other hand, the peak performance is much higher in the ALU array. Therefore we are interested in the processing efficiency of the XPP array and use RAW as a reference. To make the comparison we need to normalize the two architectures, which we do by estimating the area of the XPP array in the same process as RAW. When normalizing the area of the XPP to the 0.15 μm process used for the RAW implementation, one XPP array would be about 42mm^2, which means that an XPP array occupies about 17 % (42/256) of the RAW area. Thus, 6 XPP arrays could be fitted on the same area as one RAW array. When calculating peak performance we assume the same clock frequencies as reported for XPP and RAW, this corresponds to a peak processing performance of 24.5GOPS (6 arrays at 4 GOPS) for XPP compared to 3.6 GOPS for RAW, using the same area. Using about 15 percent of the peak performance for computations in an application would still mean that the ALU array is competitive compared to the more traditional chip multiprocessor architecture. (However, it should be noted that XPP64-A is a 24-bit architecture whilst RAW is a 32-bit architecture). Theoretically this means that up to 85% of the ALU resources, can be used to implement control functions for more complex algorithms, still the ALU array could be more effective than a chip multiprocessor, when comparing performance/area.

An obvious tradeoff is that, by making the processing elements simpler there is also a requirement that more ALU resources are used to implement the program control flow. Since the XPP processing elements have no instruction memory, the array needs to be reconfigured if algorithms are too large to fit on the array. This means that the designer first tries to statically map as large portion of the application, data and control flow, as possible. If the complete algorithm does not fit on the array it must be temporally divided and the array is reconfigured during runtime when executing the algorithm. Even if ALU arrays offer a lot of parallelism, there is a great challenge to efficiently make use of it.

4. Implementation study

We have chosen to study the XPP architecture as a candidate for micro-level reconfigurability in our outlined architecture as shown in Figure 2. In the implementation study we address the control of the computations implemented on this coarse-grained reconfigurable architecture. On this type of reconfigurable array architecture both computations and control functions have to be implemented using reconfigurable elements and communication paths. Therefore, it is interesting to see the relation between how much of the resources are used for computations and how much are used for control.

4.1. The XPP Architecture

The “extreme processing platform”, XPP, architecture from PACT XPP Technologies, represents a word-level reconfigurable architecture. A recent implementation of this architecture, the XPP64-A, operates on 24-bit data.

The architecture consists of a number of parallel processing array elements labeled PAEs. There are in turn two types of PAEs, arithmetic and memory, labeled ALU-PAE and RAM-PAE respectively. The PAEs are arranged in an array with both ALU-PAEs and RAM-PAEs, as illustrated in Figure 3. The XPP64-A has an 8x10 array, where the first and the last columns are RAM-PAEs. Four I/O element objects are connected to each of the four corners of the array. The I/O can be configured to read simple streaming input as well as to access an external memory.

The array elements can be configured to execute their operations when triggered by an event signal indicating that new data is available at the input ports. A new output can be produced every clock cycle and the result constitutes a data output and an event signal indicating that data is ready on the output port.

The ALU-PAE comprises a data path, with two inputs and two outputs, and two vertical routing resources. The vertical routing resources can also perform some arithmetic and control operations. One of the two is used for forward routing and the other for backward routing. The forward routing resource, labeled FREG, is, besides for routing, also used for control operations such as merging or swapping two data streams. The backward routing resource, BREG, can be used both for routing and for some simple arithmetic operation between the two inputs. There are also additional routing resources for event signals which can be used to control PAE execution. The RAM-PAE is exactly the same as the ALU-PAE except that the data path is exchanged by a static RAM. The RAM-PAE can be configured to act either as a dual-ported memory or as a FIFO.

The design flow for the XPP technology constitutes using either a vectorizing C-compiler or direct programming

in the native mapping language (NML), which is the XPP-ISA assembly language.

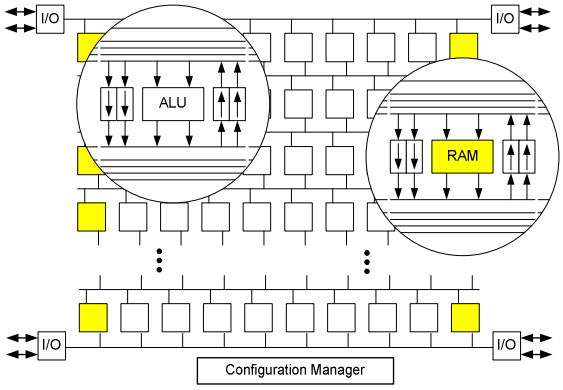

Figure 3. The XPP architecture. The highlighted areas show an ALU-PAE and a RAM-PAE including the vertical routing resources.

4.2. Implementation of the FFT algorithm

FFT is one important class of the algorithms often used in signal processing applications. Some of the most common FFT algorithms are the well known Radix FFT algorithms where the algorithm complexity in computing the Fourier transformation can be reduced from N^2 to $N \log N$ [8]. By inspecting the characteristic Radix-2 pattern, which is illustrated in Figure 4, it is easily seen that the algorithm can be divided into $\log_R N$ consecutive stages, where N is the number of samples to be processed and R is the radix of the FFT. The data in each stage is being processed and rearranged according to the characteristic radix pattern, often referred to as bit reversed order, while it is propagated through the algorithm. The basic computation performed at each stage is called a butterfly. As can be seen in Figure 4, the complex sample b is multiplied with a complex phase shift constant W_N. The output pair (A,B) is then formed by adding and subtracting the complex sample a with bW_N. In total, N/R butterflies have to be computed in each FFT stage. The FFT was chosen for the implementation study since the radix pattern requires fairly complex control functionality.

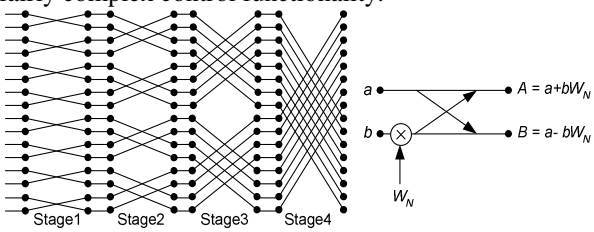

Figure 4. A 16-point Radix-2 Decimation In Time FFT communication pattern is shown to the left and a butterfly computation to the right.

We have used available development tools for the XPP reconfigurable architecture to implement and simulate a pipelined Radix-2 FFT. Figure 5 shows the functional schematic of an FFT module that can be configured to compute one or several butterfly stages. In this case, we use a double buffering technique between two consecutive stages in the FFT so that an entire sample sequence can be read and written in parallel without conflicts in the address space.

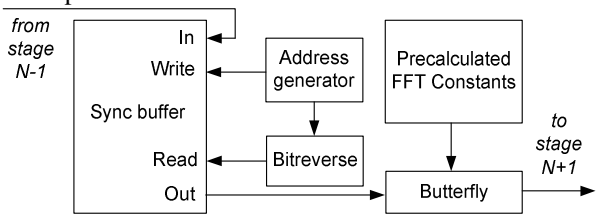

Figure 5. Double buffered FFT stage.

Radix-2 address calculation is performed when reading the data from the buffer and streaming it to the butterfly. In order to prevent the input and output streams from accessing the same address range concurrently, the write and read address streams have to be synchronized before the address counting is restarted after each butterfly stage. The phase shift constants that are multiplied with the input in the butterfly computation are precalculated and stored internally using RAM-PAEs. After the data has been streamed and processed through the butterfly it is being passed on to the consecutive stage. The RAM-PAEs can be configured to form one or several larger RAM banks dependent on the required size. We are using two separate I/O channels to stream data into the array, since we are using separate words for the real and the imaginary parts of the complex valued input. Therefore two double buffers are used to store the real and imaginary data and two memory banks to store the real and imaginary parts of the FFT constants. Figure 6 shows a block schematic of the implemented double buffer.

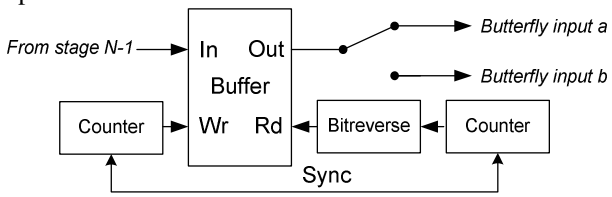

Figure 6. Synchronized I/O buffers between two consecutive FFT stages.

The buffer address space is divided into a lower and an upper half and we use two separate counters to generate addresses for reading and writing to the buffer. The counter to the left in Figure 6 generates write addresses in consecutive order for the input stream. The counter to the right produces the read address which then is bit reversed to generate the characteristic Radix-2 address pattern. A counter can be implemented by combining several FREG and BREG instructions in a single PAE. The Radix-2 bit reversal is implemented using a combination of PAEs performing shifts and additions. When both counters have reached the end of each sequence, a synchronization signal is generated and fed back to the counters to make them reset and start over.

4.3. Radix-2 Butterfly

Figure 7 shows the data operation that is performed in the butterfly. Two samples are needed to compute the output from each of the *N*/2 butterflies in each stage. As could be seen in Figure 6, the sequential stream of samples are alternatingly forwarded to the *a* and *b* inputs in the butterfly through a PAE demux instruction. Since the XPP is a fixed point arithmetic architecture, the results from the arithmetic operations have to be scaled to assure the output is kept within the used integer range. After the data has propagated through the butterfly, the resulting *A* and *B* samples are merged into a serial stream and then fed through the I/O channels to the next stage. This is implemented using PAE merge instructions.

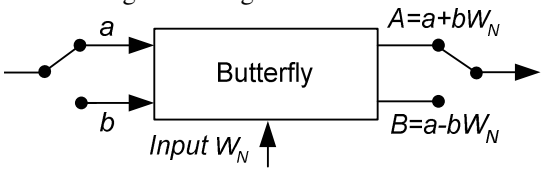

Figure 7. Data stream through the Radix-2 butterfly.

The implemented FFT configuration is capable of processing a continuous stream of data with the speed of one complex-valued word length sample per clock cycle. All programming has been done using the NML language. All PAE configurations have been placed manually. This was made in order to achieve a functionally correct and balanced dataflow through the array.

Table 1 shows how large portion of the available reconfigurable resources that have been used for the FFT implementation. The available number of configurable objects of each kind on the array is marked within the parentheses next to the name of each object. This is shown to the left in the table. For each class, the number of objects that have been configured for the FFT implementation is listed in the middle. The rightmost column shows the ratio, in percent, between used objects of each kind through the available objects of that kind.

Table 1. Array resource usage for the Radix-2 FFT.

Object	Used	Total share
ALUs (64)	12	19%
FREGs (80)	31	39%
BREGs (80)	44	55%
IOs (8)	4	50%

As can be concluded from the figures in the table, we use less than half of the available array resources, except for the BREGs. However, a portion of these BREGs are used for vertical routing of data on the array. We have not optimized the usage of routing resources which would likely be possible to do with more careful and strategic placement of the configured PAEs on the array.

A considerable part of all the arithmetic operations performed on the array are used for controlling the dataflow on the array. In Table 2 we have listed how many of the configured ALU operations in total that are used for controlling the dataflow and how many are used in the butterfly, where the signal output is calculated. About 74 percent of the configured ALU resources are used for dataflow control and 26 percent are used for computing the butterfly outputs. A great portion of the resources used for the dataflow control is a consequence of using separate address counters and control functions for each of the I/O streams. The distribution of the figures may, at first look, be surprising. However, one should bear in mind that, in ordinary program code, many algorithms normally require portions of the code to do address calculations, incrementing and testing loop bound counters etc.

Table 2. Distribution of the total Arithmetic operations.

Functionality	ALU ops	Total share
Dataflow control ops	34	74%
Butterfly ops	12	26%

Out of the 12 ALU instructions used for the butterfly configuration, two are used for fixed point scaling. The 34 ALU instructions implementing the dataflow control functionality are used for address counters, bit reverse and data path switching operations across the array. What proved to be difficult was the implementation of the dataflow control structures while maintaining a balanced dataflow within the entire algorithm. Even though the XPP has automatic dataflow synchronization, it required time consuming, low-level engineering to balance the dataflow through all parts of the FFT and keep up a high throughput rate. Algorithms like the FFT that contain address calculations, loop control and memory synchronization tend to require much of the resources to be spent on implementing the dataflow control in the algorithms. It might be possible to optimize the FFT implementation so that fewer configurable objects would be needed. However, solutions using long pipeline data paths tend to be complex to implement and balance on the XPP array. Therefore, in order to reduce complexity with synchronization and dataflow balance, we used separate flow control for each of the two I/O streams which resulted in the presented area tradeoff.

Still, referring back to the coarse area comparison between an array of ALUs and a chip multi processor, one could argue that, for the computation studied, the ALU array makes more efficient use of the silicon area. Even if the majority of the resources were used for control purposes, the performance achieved in our implementation is well beyond the peak performance of the more coarse-grained multi processor width dedicated control resources.

4.4. Macro-level mapping

The implemented FFT module could be used as a building block for FFTs of variable size. The performance could be adapted for different requirements, by pipelining

FFT modules using several nodes, which is illustrated in Figure 8. The highest configuration level — the macro level — constitutes a higher abstraction level for efficient mapping of pipeline parallelism within an algorithm, like the implemented FFT, or between different kinds of functions in a signal processing chain, like in the earlier described radar signal processing application. Figure 9 illustrates the general situation. Different functions in the graph should be allocated to either a single node or a group of nodes depending on resource and performance requirements. The mapping onto the macro level should try to exploit pipeline parallelism as well as parallelism stemming from multiple inputs.

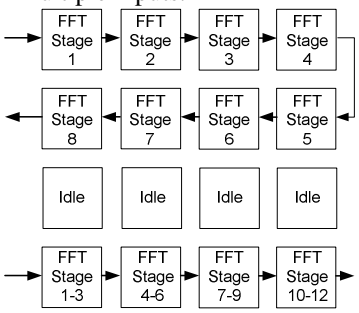

Figure 8. Pipelined FFT computations mapped onto the macro-level.

To support dynamic mapping, the macro level architecture needs to support reconfigurable interconnection between the nodes. Further studies of how to best implement this, as well as how to control the configuration flow at the macro level and the control of dataflow between nodes, will be needed. These studies must be made using more and realistic applications to capture the characteristics of the application domain.

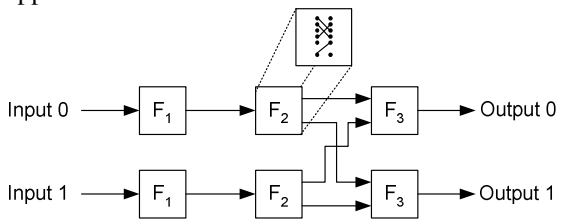

Figure 9. Functions that should map onto the macro-level structure.

5. Discussion and Conclusions

We have outlined a two-level configurable architecture, intended for embedded high-performance signal processing. The coarse-grained macro-level consists of nodes interconnected in a mesh. For the micro-level we believe reconfigurable hardware could be used for efficient implementation of the signal processing functions.

We described an implementation mapping an FFT algorithm onto a coarse-grained reconfigurable architecture. With this implementation study we quantified the resource usage in the FFT algorithm. The main focus of the study was to find out how the architectural resources were allocated to different parts of the algorithm, in particular the amount of resources needed to feed the main computational core with data. This would roughly correspond to the program code and control logic in a microprocessor.

The implementation study shows that a considerable part of the resources are used for implementing the control structures in the FFT implementation. By implementing these structures in parallel the data can be kept streaming through the nodes continuously at maximum I/O speed. The needed allocation of resources for control is not surprising since, in this architecture, there are no dedicated functional units for control purpose. Many algorithms in signal processing require less control than the FFT. It is possible to map these with good efficiency. Other algorithms will need reconfiguration of the array during runtime. This implies efficient reuse of hardware, but may have negative impact on performance.

The FFT implementation has also been used to show different ways of mapping a pipelined algorithm onto several nodes of configurable arrays.

Further studies must be made to analyze the requirements on the interface to the micro-level nodes. On the macro-level the requirements on the communication and configuration must be analyzed for different, realistic applications.

6. Acknowledgments

This research has been financed by a grant from the Knowledge Foundation. We are also grateful to PACT XPP Technologies for making their development tools available to us.

7. References

[1] W. Liu and V. Prasanna, “Utilizing the Power of High-Performance Computing”, *IEEE Sig. Proc. Mag.*, Sept. 1998, pp. 85-100.

[2] A. DeHon, The Density Advantage of Configurable Computing, *IEEE Computer*, Vol. 33, No. 4, April 2000, pp 41-49.

[3] Elixent, “Changing the electronic landscape”, http://www.elixent.com.

[4] V. Baumgarte, F. May, M. Vorbach, and M. Weinhardt, "PACT XPP – A Self-Reconfigurable Data Processing Architecture," *Int’l. Conf. on Engineering of Reconfigurable Systems and Algorithms ERSA 2001*, Proc., CSREA Press, 2001.

[5] R. Baines and D. Pulley, “A Total Cost Approach to Evaluating Different Reconfigurable Architectures for Baseband Processing in Wireless Receivers”, *IEEE Communications Magazine*, vol. 41,no. 1, Jan. 2003, pp. 105-113.

[6] Paul Masters, “A look into QuickSilver’s ACM architecture”, *EETimes*,
http://www.eetimes.com/story/OEG20020911S0070

[7] M. B. Taylor, et. al., ”The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs”, *IEEE Micro*, Mar/Apr 2002, pp. 25-35.

[8] J. G. Proakis, D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd ed., Prentice-Hall International, UK (1996)