A HW/SW Design Methodology for Embedded SIMD Vector Signal Processors

J.P. Robelly, G. Cichon, H.Seidel, and G. Fettweis

***Abstract*— SIMD processors have made their way from supercomputers architectures through embedded real-time signal processing. This trend has been driven by signal processing applications with heavy number-crunching requirements like for example base-band processing on mobile devices.**

Depending on the data dependencies of algorithms and implementation constraints like real-time, power consumption and die size, the necessary SIMD parallelism can be put into a piece of silicon for a certain application. This poses two challenges: On the one hand, the DSP core design has to be streamlined in such a way that changes on the architecture can be prototyped very fast. On the other hand, the algorithm design and its development have to be done independent of the level of SIMD parallelism available on the DSP in order to enable software reusability.

In this paper we report our HW/SW methodology in order to design DSP cores and algorithms that exploit SIMD parallelism. On the hardware development side and taking as a starting point a novel hardware architectural template called STA[1], we explain how with our approach we automatically generate simulation and hardware models of DSP cores with a scalable level of SIMD parallelism. On the software development side and based on an algebraic model that captures the SIMD computational model, we explain how algorithms can be designed independent of the available SIMD parallelism. We also report how this algebraic model can be easily expressed in Matlab syntax. This enables the automatic code generation from Matlab programs for our family of DSP cores.

***Index Terms*— SIMD, DSP, Design Methodology, Automatic Code Generation.**

I. Introduction

SIMD vector signal processors offer the potential to increase the data transfer rates between memory and computational resources, since data vectors residing on memory are accessed and processed in parallel fashion. This enables the achievement of speed up gains in the implementation of DSP algorithms into these processor architectures. In the light of the advancement of VLSI technology, we have experienced in the last time how the SIMD computational model, whose origin goes back to the old times of supercomputers, has made its way through the implementation of embedded real-time signal processing. Supercomputing has become feasible for embedded applications.

This renewed attention on SIMD processors has been driven by low-power and low-size applications with ever increasing algorithm complexity, where a programmable device is favored over the fixed wired solution. This is the case of wireless communications, where programmable devices not only allow for the flexible implementation of constantly changing standards but enable the realization of the software defined radio paradigm. Moreover, it turns out that the time-invariant nature of the wireless communications channel leads to a frame based signal processing. Channel variations are tracked and a snapshot channel estimation is used for the detection process. The detection process is then carried out over a frame that is considered to have been transmitted through a channel which is assumed to remain constant over the buffer length. This frame based nature of signal processing for mobile applications is a natural fit to the memory based processing approach of the SIMD computational model [1].

The tight area and power constraints of mobile devices have been also a driving force for the SIMD computational model: A piece of silicon can be filled up with processing elements like adders and multipliers in order to deliver the required computational power and the control overhead remains constant regardless of the level of SIMD parallelism available on the processor. This contrasts with other approaches in which reconfigurability or programmability is guaranteed at expenses of a considerable overhead.

However, the amount of SIMD parallelism that has to be implemented in a processor does not only depends on power and area constraints, but it also depends on the characteristics of the algorithms. Data dependencies of an algorithm can cause that the achievable speed up factor is upper bounded.

This has as a consequence that power, area and real-time constraints together with the data dependencies characteristics of the algorithms have to be taken into account in order to determine the required level of SIMD parallelism. This imposes new challenges to the HW/SW design methodology. On the one hand, the hardware development process and the DSP architecture have to support the implementation of processor cores with a scalable level of SIMD parallelism. On the other hand, the algorithms have to be designed and implemented in such a way that they can be easily scaled to different levels of SIMD parallelism in order to enable software reusability.

In this paper we present our methodology for dealing with these challenges. The paper has two parts: one concerned with hardware design and one with software design. In Section II, the STA microarchitecture and its framework for the generation of DSP cores will be addressed. The compiler friendliness of the STA architecture is the starting point of section III. In this section we present an algebraic approach that offers the

Manuscript received June 1, 2004; This work was supported by the German Science Foundation (Deutsche Forschungsgemeinschft DFG) under grant SFB358.

J.P. Robelly, G.Cichon, H. Seidel and G. Fettweis are with the Vodafone chair for mobile communications systems at the Technische Universitaet Dresden, D-01062 Dresden, Germany (e-mails: robelly,cichon,seidel,fettweis@ifn.et.tu-dresden.de).

[1]Synchronous Transfer Architecture

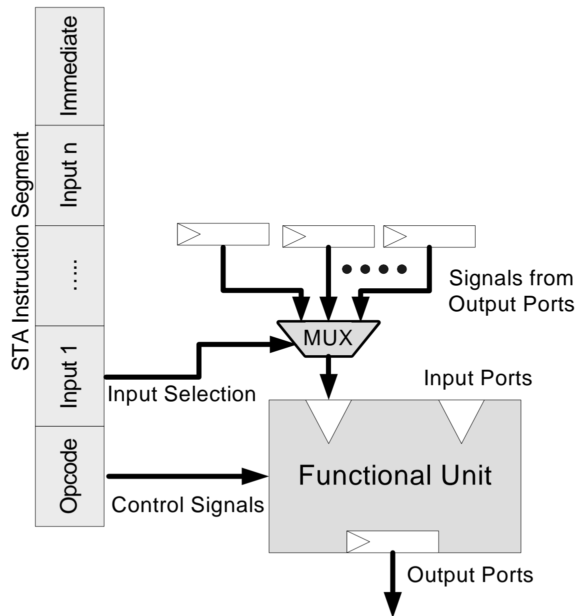

Fig. 1. STA Architectural Template

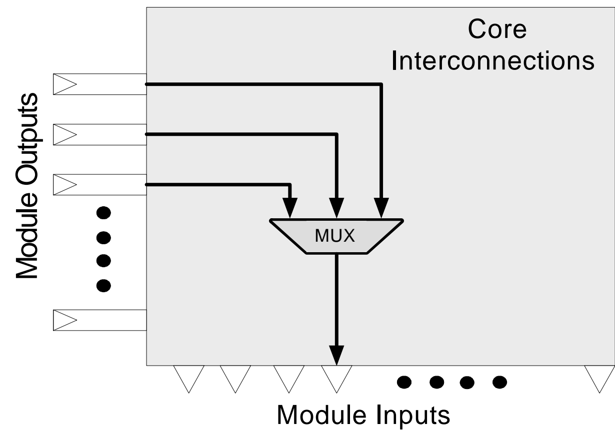

Fig. 2. STA Interconnection Network

required abstraction for designing algorithms independent of the level of SIMD parallelism available. In this section we also explain how this algebraic framework has been used in order to enable the automatic generation from programs written in Matlab syntax for the family of our processor cores. Finally, in section IV our conclusions.

II. HW Design: Synchronous Transfer Architecture STA

A. STA Architectural Template

Our HW design concept is based on basic blocks of the form showed in figure 1. Such a basic block has an arbitrary number of input and outputs ports. It is important to remark that in our concept only the output ports of basic blocks are registers. The input and output ports can deal with a certain data type, e.g. bool, 16-bit integer, vectors of 16-bit integer, etc. Each basic block implements some functionality. In our HW concept, a system is build up upon these basic blocks. Thus, ports of the same data type are connected with each other through an interconnection network formed by multiplexers. This is sketched in figure 2. Both the functionality of basic blocks and the input multiplexers are explicitly controlled by processor instructions. At each cycle the instruction configures the multiplexing network and the functionality of the basic blocks. In this context, our multiplexing network resembles the bypass network frequently used in superscalar microprocessors [2]. Thus, the whole system forms a synchronous network, which at each clock cycle consumes and produces some data. The produced data will in turn be consumed by other basic blocks in the next cycle. Due to the synchronous transference of data between basic blocks we have named the architecture STA [3]. In our concept, basic blocks can be highly optimized data paths or memory blocks. Memory blocks can be either registers files or memories. There is a register file for each data type available in the processor.

The STA architecture supports data and instruction level parallelism. In fact, SIMD data parallelism is supported by letting the input, output ports and the data paths to deal with vectors data types. Instruction level parallelism is supported, since at each cycle a wide instruction controls each basic block and the multiplexing network in *VLIW* fashion. This poses two problems. On the one hand, for large STA systems the multiplexing interconnection network becomes a critical part of the design. On the other hand, a wide instruction memory is needed. The complexity of the interconnection network can be alleviated by reducing the number of connections between ports. An obvious strategy for this is to determine those connections which allow for reusing program variables that present a high data locality. This is a viable approach, since applications are known at the design time of the processor and thus, a customization of the multiplexing interconnection network can be carried out in order to fulfill die size and power consumption requirements. For those connections which are not frequently reused, a connection with the register file suffices. To alleviate instruction memory footprint we are applying code compression techniques similar to [4].

B. Formal Description of STA Cores

In order to formalize and manipulate the necessary information that captures an STA design, we have developed a dynamically-typed object-oriented class library called *RNA* [5]. By means of RNA we can create a machine description file that contains information regarding data types, input and output ports of processor basic blocks, connections between ports and behavioral of each basic block. For example a 16-bits and a 40-bits integer data type can be declared in RNA as follows:

```
int_t:(type-int true 16 name="int_t")
accu_t:(type-int true 40 name="accu_t"),
```

where `int_t` and `accu_t` are labels for the data types 16-bits integer and 40-bits integer respectively. Such labels can be instantiated at other stages in the machine description file, for example a data type vector of 8 elements, where each element is a 16-bit integer can be declared in RNA as follows:

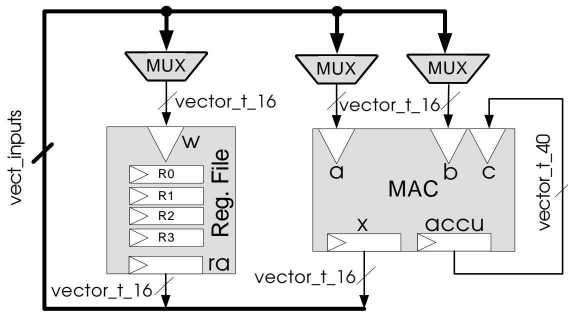

Fig. 3. Simple STA Core: MAC and a Register File Unit

```
vct_16:(type-vector int_t
(dimension-known s:8) name="vct_16")
```

Likewise, we can declare a data type vector of 8 elements where each element is a 40-bit integer:

```
vct_40:(type-vector accu_t
(dimension-known s:8) name="vct_40")
```

It is important to point out that in these declarations the number of vector elements can be parameterized and thus, we can describe STA cores with different levels of SIMD parallelism. We use the labels related to data types in order to describe the input and output ports of basic blocks. For example, in order to declare a *MAC* unit in RNA we write

```
mac:(md-fu "mac" in_ports out_ports),
```

where `in_ports` declares three input ports `a,b,c`

```
in_ports:[
(md-input "a" vct_16 vect_inputs)
(md-input "b" vct_16 vect_inputs)
(md-input "c" vct_40 [
(md-connection mac "accu")])]
```

and `out_ports` declares two output ports `x,accu`

```
out_ports:[(md-output "x" vct_16)
(md-output "accu" vct_40)]
```

It is important to note that in the declaration of `in_ports` we find the description of which output ports of the same or of other basic blocks are connected to the input ports. For instance, the 40-bit output port of the unit `mac` is connected to the 40-bit input port of the same unit. In order to describe more complex connections we can make use of labels again. For example, the label `vect_inputs` of the declaration `in_ports` can be defined as follows:

```
vect_inputs:[
(md-connection vect_rf "ra")
(md-connection mac  "x")]
```

To complete our example we write the declaration of a register file unit with four registers of type `vct_16` as

```
rf:(md-rf "rf" 4 vct_16 wr_port rd_port),
```

where

```
rd_port:[(md-read "ra")]
```

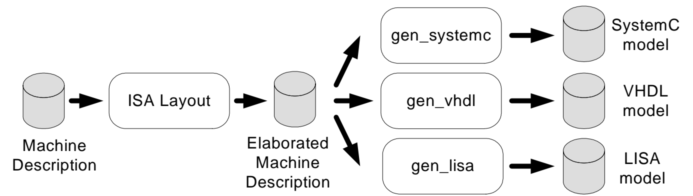

Fig. 4. STA Core Generator

and

```
wr_port:[(md-write "w" vect_inputs)].
```

This example of RNA describes a simple STA system with the organization presented in figure 3. In the same way as we have described this simple example, we can describe more complex systems comprising memories, address generators, program counters, ICU (Data Interconnection Network), etc. However, by means of RNA we can not only describe data types, basic blocks and connections between ports, but we are also able to describe the behavior of processor instructions. For example, the declaration

```
(md-op "mac_op" (md-opcode mac 0x1)
(md-tmpl "+" [(md-tmpl "*" [
(md-match "a" 0 vct_16)
(md-match "b" 0 vct_16)])
(md-match "c" 0 vct_40)
][
(md-match "accu" 1 vect_40)
])
behaviour="$accu=$a*$b+$c;"
)
```

is the instruction template for a vector MAC operation. The instruction has three vector inputs. Two inputs are vectors with 16-bit elements and one input is a vector with 40-bit elements. The input operands are fed to the ports `a,b,c` at the time 0 and the result `accu` is produced one cycle later. In the case of pipelined data paths, this parameter can be replaced by the latency of the unit. The behavior section describes in a C-like syntax how the result has to be computed.

C. Interim Discussion

Contrary to other processor description approaches like [6],[7][8], the purpose of RNA is not to become a new generic *ADL* (Architecture Description Language). Our starting point is the STA architectural template and the origin of RNA was motivated by the necessity of describing STA systems in an structural rather than at a behavioral way. In fact, the intention of RNA is to capture the necessary information of our scalable STA cores in order to enable the automatic manipulation of the machine description file.

D. Automatic Core Generation

The description of an STA concept in a machine description enables the automatic manipulation of this information in order to generate different hardware and simulation models

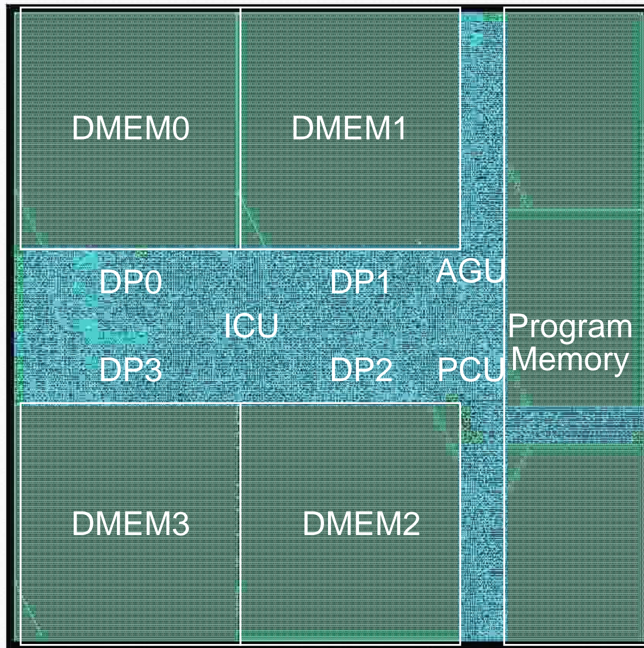

Fig. 5. STA Processor Layout

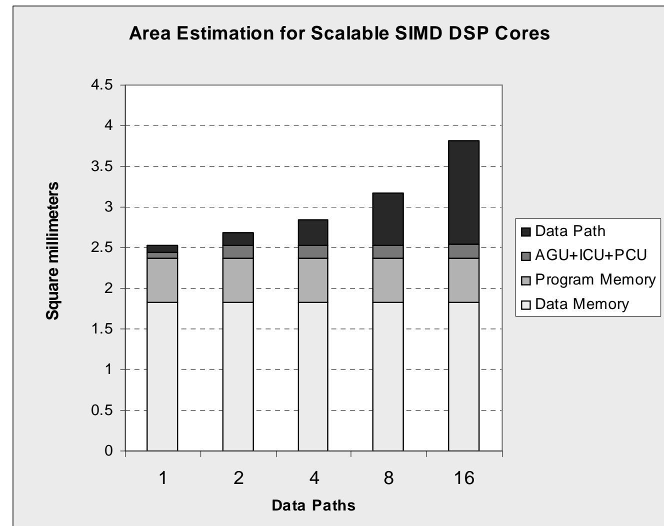

Fig. 6. Area vs. Level of Parallelism

of the processor. In figure 4 we can observe the processing stages we are applying to the machine description file. At first, an instruction set architecture layout is created. For this purpose, the binary encoding for instructions and multiplexer control are determined. The binary encoding establishes the bit width of the corresponding fields in the instruction word. Then, different processor models are generated. For processor simulation purposes a model in the ADL language *LISA* [8] is generated. RTL models of the processor are also generated. Since the declaration of the instruction template has a C-like behavior section, a complete SystemC model can be generated. The VHDL model contains the complete multiplexing interconnection network and stubs for the basic blocks. These stubs can be filled with optimized data paths.

As discussed in [3], an slice-based point of view of the design is important to obtain good quality results in the hardware synthesis. Thus, for the generation of the hardware models those basic blocks are recognized, which can be split into slices. From the hardware point of view, it is more efficient to encapsulate the logic in slices. Thus, only one slice has to be carefully placed and routed and then it can be replied to achieve the required level of parallelism. The RNA framework makes possible to support this kind of features that are specific to our architectural template. This approach enables us to take special measures that have a great impact on the quality of the generated hardware and which otherwise could not be possible with a more general hardware generation approach.

E. STA Synthesis Results

In [9] we have presented an STA DSP for the implementation of an OFDM communications system. Versions of this processor with different levels of SIMD parallelism were synthesized, placed and routed. The synthesis were carried out with a 130 nm standard-cell library by UMC using Synopsys Design Compiler. For place and route and back annotation we used Cadence SoC Encounter. For the implementation of the arithmetical data paths we used DesignWare components from Synopsys. For Memories we used SRAM macros from UMC. In figure 5, we can observe the layout for a processor with 4 data paths, a data memory with 4096 words of 16 bit each slice and a program memory with 3072 words of 64 bits. In figure 6, we can observe the estimation of area in mm^2 for different numbers of data paths. This values were estimated using the same amount of memory as indicated above.

According to the synthesis results the DSP core with 16 data paths achieves a clock rate of 250 MHz and its power consumption is 300 mW. The power estimation were done using Synopsys power compiler and wired-load models extracted from the placed and routed processor.

F. STA: A Compiler Friendly Architecture

Contrary to the sliced perspective for the hardware generation, from a compiler point of view the STA architecture resembles a synchronous network of basic blocks, which either execute some operation (data paths) or store a state (memories and register file). Moreover, the connection between basic block takes place at ports that deal with the same data types. The multiplexing interconnection network determines whether there is a direct connection between modules or whether data can be communicated to other basic blocks through the register file. At each clock cycle both the multiplexing interconnection network and the basic block are configured by the instruction. These characteristics of STA offer a high degree of freedom for the compiler, which can apply standard compiler techniques [10] [11] in order to find a solution for the routability of data in the processor. In the next section, we will present our approach for generating code for scalable versions of our DSP architecture.

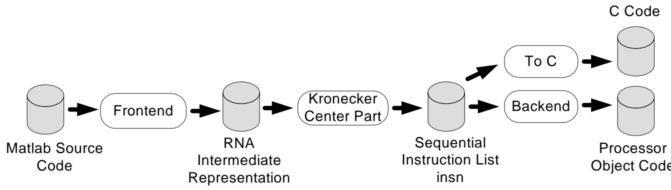

Fig. 7. Compiler Architecture

III. SW Design: A Matlab Compiler

For the processor software generation, we aim to convert Matlab source code into object code for scalable versions of our STA DSP cores. This means that the source code has to be parameterized in such a way that the compiler can generate object code for different levels of SIMD parallelism. In figure 7, we can observe the architecture of our compiler. The Matlab source code is converted into an intermediate representation by the compiler frontend. It is important to mention that the intermediate representation of our programs is expressed using the same syntax we used for the machine description files, namely RNA. For the compiler, RNA offers a framework in which annotated graphs can be described.

The *Kronecker* center part of our compiler plays an important role on the scalable code generation as we will explain further. The function of the center part is to recognize algebraic structures which are embedded in the matlab code and that describe the SIMD parallelism in order to generate vector instructions for this structures. These special algebraic structures captures the SIMD computational model and they can be recognized and translated into vector operations by the compiler center part. Moreover, this algebraic structures can be fully parameterized and thus, they are independent of the level of parallelism: Algebra is used in order to achieve an abstraction of the algorithms regarding the level of parallelism available on the processor.

The Kronecker center part generates a sequential list of SIMD-vector instructions. In fact, these RISC-like instructions contains the whole information related to the SIMD parallelism of the algorithm. The purpose of the compiler backend is to exploit the instruction level parallelism present on the sequential instruction list generated by the compiler center part. For details related to the compiler backend the reader is referred to [12].

In the next sections we concentrate only on the Kronecker center part.

A. Kronecker Compiler: Mathematical Background

Davio [13] in his classical paper developed an algebraic framework, where he established the connection between the *Kronecker* product of matrices and stride permutations. In his paper he proved the important *commutation theorem* of Kronecker products

$$A \otimes B = P(mn, m)(B \otimes A)P(mn, n), \tag{1}$$

where $P(L, s)$ is a matrix that represents the permutation of L elements with stride s and $A \otimes B$ is a matrix known as the Kronecker product of the $m_A \times n_A$ matrix A and the $m_B \times n_B$ matrix B that is defined as follows:

$$A \otimes B = \begin{bmatrix} a_{0,0}B & \dots & a_{0,m_A-1}B \\ a_{1,0}B & \dots & a_{1,m_A-1}B \\ \vdots & \dots & \vdots \\ a_{n_A-1,0}B & \dots & a_{n_A-1,m_A-1}B \end{bmatrix} \tag{2}$$

This remarkable result enabled researchers to formulate different fast algorithms for signal transformations like discrete fourier transforms, walsh-hadamard transforms, discrete cosine, sine transform, etc. using the same algebraic framework. Tolimieri [14] uses Kronecker products in order to formulate FFT algorithms for different parallel architectures. Tolimiere makes also the observation that different models of parallel computation can be described by means of Kronecker products. This has originated a lot of research in order to automatically generate code of signal transformations like FFT, Cosine and Sine transform for different parallel architectures [15][16]. Although there has been a lot of research on the generation of signal transformations, there has been scarce research on extending these ideas in order to support a wider range of digital signal processing algorithms.

In order to characterize the SIMD computational model let A be an $m \times m$ matrix, $\underline{x}$ a $Nm \times 1$ vector, I_N the $N \times N$ identity matrix and consider the following important expression:

$$\begin{aligned} \underline{y} &= (A \otimes I_N)\underline{x} \qquad (3) \\ &= \begin{bmatrix} a_{0,0}I_N & \dots & a_{0,m-1}I_N \\ a_{1,0}I_N & \dots & a_{1,m-1}I_N \\ \vdots & \dots & \vdots \\ a_{m-1,0}I_N & \dots & a_{m-1,m-1}I_N \end{bmatrix} \begin{bmatrix} \underline{x}_0 \\ \underline{x}_1 \\ \vdots \\ \underline{x}_{m-1} \end{bmatrix}, \end{aligned}$$

where using Matlab syntax we define the vector of N elements $\underline{x}_i = \underline{x}\big(iN : (iN + N - 1)\big)$ for $i = 0, 1, \dots, m-1$. Equation (3) is known as a Kronecker vector factor [14], since it expresses the computation of m vectors $\underline{y}_i$ of size $N \times 1$ as vector operations that deal with vectors of size N. This is better illustrated if we rewrite equation (3) as follows:

$$\underline{y}_i = \sum_{q=0}^{m-1} (a_{i,q} \otimes I_N)\underline{x}_q, \tag{4}$$

for $i = 0, 1, 2, \dots, m-1$ and using Matlab syntax again $\underline{y}_i = \underline{y}\big(iN : (iN + N - 1)\big)$. Equations (3) and (4) captures the SIMD computation model. Further we assume for the level of SIMD parallelism

$$N = 2^n \quad \text{for } n = 1, 2, 3, \dots$$

B. Application Example 1

In this section we show by means of a simple example how the algebraic notation presented in section III-A can be used in order to describe the SIMD computation of an algorithm.

Let $\underline{x}$ and $\underline{y}$ be input and output vectors defined as

$$\begin{aligned} \underline{x} &= [x(k) \quad x(k+1) \quad x(k+2) \quad x(k+3)]^T, \\ \underline{y} &= [y(k) \quad y(k+1) \quad y(k+2) \quad y(k+3)]^T, \end{aligned}$$

and a vector of coefficients defined as

$$\underline{h} = [h_0 \quad h_1 \quad h_2 \quad h_3]^T.$$

We consider the following computation

$$\underline{y} = H\underline{x}, \tag{5}$$

where the transformation matrix is formed taking the coefficients of the vector $\underline{h}$ and has a Toeplitz structure as follows:

$$H = \begin{bmatrix} h_0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix}$$

Assuming that two results can be computed in parallel, we can write for the computation of (5) the following equations

$$\underline{y}_0 = \begin{bmatrix} h_0 & 0 \\ 0 & h_0 \end{bmatrix}\begin{bmatrix} x(k) \\ x(k+1) \end{bmatrix} + \begin{bmatrix} h_1 & 0 \\ 0 & h_1 \end{bmatrix}\begin{bmatrix} 0 \\ x(k) \end{bmatrix} + \begin{bmatrix} h_2 & 0 \\ 0 & h_2 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} h_3 & 0 \\ 0 & h_3 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\underline{y}_1 = \begin{bmatrix} h_0 & 0 \\ 0 & h_0 \end{bmatrix}\begin{bmatrix} x(k+2) \\ x(k+3) \end{bmatrix} + \begin{bmatrix} h_1 & 0 \\ 0 & h_1 \end{bmatrix}\begin{bmatrix} x(k+1) \\ x(k+2) \end{bmatrix} + \begin{bmatrix} h_2 & 0 \\ 0 & h_2 \end{bmatrix}\begin{bmatrix} x(k) \\ x(k+1) \end{bmatrix} + \begin{bmatrix} h_3 & 0 \\ 0 & h_3 \end{bmatrix}\begin{bmatrix} 0 \\ x(k) \end{bmatrix},$$

where

$$\begin{aligned} \underline{y}_0 &= [y(k) \quad y(k+1)]^T \\ \underline{y}_1 &= [y(k+2) \quad y(k+3)]^T \end{aligned}$$

Using the notation introduced in section III-A, we can write for the computation of equation (5) the following

$$\underline{y}_i = \sum_{q=0}^{3}(h_q \otimes I_2) \cdot \Big[(Z_2)^{(q) \bmod 2}\underline{x}_{i-\lfloor \frac{q}{2} \rfloor} + (Z_2^T)^{(2-q) \bmod 2}\underline{x}_{i-1-\lfloor \frac{q}{2} \rfloor}\Big],$$

where $0 \leq i < 2$, $\lfloor \cdot \rfloor$ is the floor operator, Z_2 is a 2×2 shift matrix defined as

$$Z_N = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix},$$

and the vector $\underline{x}_j$ is defined as

$$\begin{aligned} \underline{x}_j &= \big[x\big(k+2j\big) \quad x\big(k+2j+1\big)\big]^T && \text{for } 0 \leq j < 2, \\ \underline{x}_j &= [0 \quad 0]^T && \text{otherwise.} \end{aligned}$$

The equations we developed above can be generalized for an arbitrary level of SIMD parallelism N and for a vector of coefficients:

$$\underline{h} = [h_0 \quad h_1 \quad \ldots \quad h_{p-1}],$$

where p/N is a positive integer greater that one. For the input and output vectors we can write

$$\begin{aligned} \underline{x} &= [x(k) \quad x(k+1) \quad \ldots x(k+L-1)]^T, \\ \underline{y} &= [y(k) \quad y(k+1) \quad \ldots y(k+L-1)]^T. \end{aligned}$$

Further, we assume that the number of input samples to be processed L is also a multiple of N, thus the following holds:

$$\frac{L}{N} = J \qquad \text{for } J = 1, 2, 3, \ldots$$

In order to formulate the scalable computation of equation (5), we partition the input and output vectors in sub-vectors of size N. Thus, for the input vectors we can write

$$\underline{x}_j = [x(k+Nj) \quad x(k+Nj+1) \ldots x(k+Nj+N-1)]^T,$$

for $0 \leq j < J$. Otherwise, $\underline{x}_j$ is a vector with N zero elements. For the output vectors, we can write

$$\underline{y}_i = [y(k+Nj) \quad y(k+Nj+1) \ldots y(k+Nj+N-1)]^T,$$

for $0 \leq i < J$. Finally the computation of (5) for an arbitrary level of SIMD parallelism can be expressed by means of the following expression:

$$\underline{y}_i = \sum_{q=0}^{p-1}(h_q \otimes I_N) \cdot \Big[(Z_N)^{(q) \bmod N}\underline{x}_{i-\lfloor \frac{q}{N} \rfloor} + (Z_N^T)^{(N-q) \bmod N}\underline{x}_{i-1-\lfloor \frac{q}{N} \rfloor}\Big] \tag{6}$$

In this equation I_N and Z_N are the $N \times N$ identity and shift matrix respectively. Thus, equation (6) describes an algorithm for the implementation of the core computations of an FIR filter with an scalable level of SIMD parallelism. This algorithm can be extended by any suitable partitioning technique like for example *overlapping save* in order to obtain the time-continuous computation of the filter.

C. Application Example 2

In [17], we presented a block formulation of pure recursive filters using Kronecker vector factors. The block formulation was derived using a lifting-isomorphism [18] with a raising factor N applied to the state-space equations of the serial formulation of the recursive filter. Our starting point for the derivation of the raised algorithm is the serial formulation of a recursive filter of order p, namely

$$y(k) = u(k) + \sum_{i=1}^{p} a_i y(k-i).$$

The state-space representation of this algorithm is given by

$$\begin{aligned} \underline{x}(k+1) &= A\underline{x}(k) + Bu(k) \\ y(k) &= C\underline{x}(k) + Du(k), \end{aligned}$$

where the state vector $\underline{x}(k) = [x_1(k) \quad x_2(k) \ldots x_p(k)]^T$ contains past computed samples. Thus,

$$x_i(k) = y(k-i) \quad \text{for } i = 1, 2, \ldots, p$$

For the system matrices we have

$$A = \begin{bmatrix} a_1 & a_2 & \ldots & a_p \\ 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 1 & 0 \end{bmatrix},$$

$$B = [1 \quad 0 \quad \ldots \quad 0]^T,$$

$$C = [a_1 \quad a_2 \quad \dots \quad a_p]^T,$$

$$D = 1$$

We define the input and output vectors of the raised algorithm as follows:

$$\underline{u}(k) = [u(Nk) \quad u(Nk+1) \quad \dots \quad u(Nk+N-1)]^T$$
$$\underline{y}(k) = [y(Nk) \quad y(Nk+1) \quad \dots \quad y(Nk+N-1)]^T,$$

where N is the raising factor. Thus, and according to [17], we can write for the Kronecker vector formulation of the algorithm the following

$$\underline{y}(k) = \sum_{i=1}^{p} \big(y(Nk-i) \otimes I_N\big)\underline{c}_i + (D \otimes I_N)\,\underline{u}(k) + \sum_{q=1}^{N-1} \big(CA^{q-1}B \otimes I_N\big)\, Z_N^q \underline{u}(k), \quad (7)$$

where $\underline{c}_i$ are vectors formed by the columns of a raised system matrix

$$C^{[R]} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{N-1} \end{bmatrix},$$

and Z_N is the $N \times N$ shift matrix.

The importance of equation (7) resides on the matter that it expresses the computation of the algorithm in terms of Kronecker vector factors. The algorithm deals with input vectors of size N and produces output vectors of size N and thus, it is independent of the available level of SIMD parallelism.

D. Automatic Code Synthesis

The automatic code synthesis is based on the matter that algorithms expressed as in equations (6) and (7) uses certain algebraic structures, which have a direct interpretation in processor instructions. For example, let us take the following expression of equation (7)

$$\big(y(Nk-i) \otimes I_N\big)$$

This expression means that the scalar $y(Nk-i)$ has to be broadcasted to all the data paths of the parallel DSP architecture. Thus, this algebraic structure will be translated into a broadcast data transfer instruction. Let us take a new example

$$\big(y(Nk-i) \otimes I_N\big)\underline{c}_i$$

This expression contains two actions that have to be carried out by the processor: 1.- broadcast of $y(NK-i)$, 2.- component-wise vector multiplication. As we can see from this example, the algorithm is expressed in such a way that enables scalability, since all the components involved in the equation have as a parameter the number of data paths of the processor N. Finally, we consider the expression

$$\sum_{i=1}^{p} \big(y(Nk-i) \otimes I_N\big)\underline{c}_i,$$

where p vector MAC operations are described. In the example above, we have presented a simple data transfer, namely broadcasting. However, we have extended the number of elementary algebraic structures in order to support a wider range of algorithms. This was the case of the $N \times N$ shift matrix Z_N, which resembles the *Zurich-Zip* data transfer. More complex transfers that are supported by our processor like stride permutations $P(L, s)$ are also part of our algebraic repertoire.

In the algorithm of equation (6), we find the operation $\lfloor \frac{q}{N} \rfloor$. This can be easily implemented in a processor as a right shift by n positions, where $N = 2^n$ holds. Moreover, this operation determines that a new vector of input samples $\underline{x}_{i-\lfloor \frac{q}{N} \rfloor}$ is loaded from memory only when the index q becomes a multiple of N. This makes clear the advantageous property of FIR filters implemented into SIMD processors by which the number of expensive memory accesses are decreased with an increasing level of SIMD parallelism.

Matlab is a very popular language among digital signal processing designers. The matrix oriented capability of the language allows for easily programming algorithms using the elementary algebraic structures of equations (6) and (7). For example, the first part of equation (7) can be written using matlab syntax as follows:

```
mac=zeros(N,1)
for i=1:p,
 mac=mac+kron(y(Nk-i),eye(N))*C_r(1:N,i)
end
```

In this example `kron()` and `eye()` are matlab functions that implement the Kronecker product and the identity matrix respectively. In this example is also important to note that the index `k` runs over the number of input vectors that have to be processed by the algorithm. In figure 8, we offer a closer view to the architecture of the Kronecker center part of our compiler. In a first processing stage data types and shape of the variables of the matlab program are determined and this information is annotated to the intermediate representation. For our programs, we have taken care of using variables with a unique definition. Supporting Matlab features of variable redefinition is a difficult problem to solve and therefore, we decided to guarantee the unique meaning of a variable by inserting assertions functions in the program. If at some stage of the program the variable has changed its original definition, the Matlab program will be interrupted with the respective assertion message.

After the data type information has been embedded into the program intermediate representation, the pattern matching stage begins to search for the algebraic structures. Finally, when the structures has been recognized SIMD high level sequential instructions are generated. The meaning of high level instructions in this context is that the sort of code that is generated at this stage is machine independent and resembles

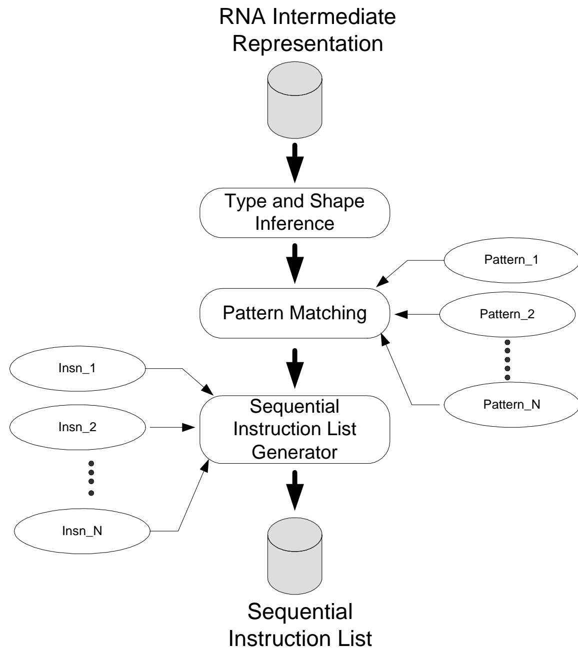

Fig. 8. Kronecker Center Part

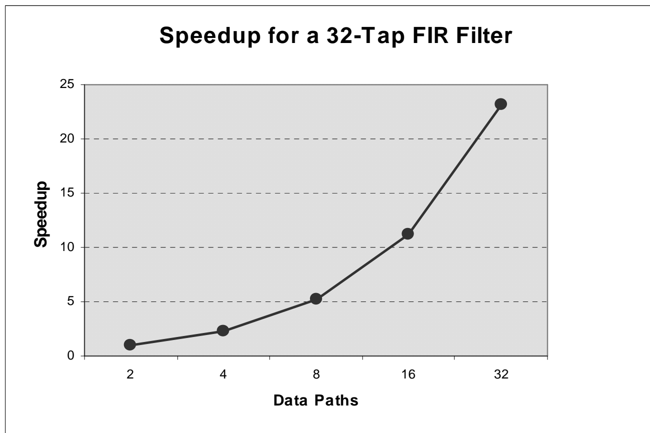

Fig. 9. Speedup Factor vs. Level of Parallelism for a 32-taps FIR Filter

an ideal SIMD processor. Further stages of the compiler will process this sequential instruction list in order to generate machine dependent details and exploit instruction level parallelism. In fact, the linear instruction list can be regarded as a low level intermediate representation, which contains the whole information regarding the SIMD parallelism and which can be processed in further stages for different target architectures. We have also implemented a conversion program that translates the sequential instruction list into C code as can be observed in figure 7.

E. Code Generation Results

Using equation (6), we have implemented an FIR filter in Matlab and processor code was automatically generated for different levels of parallelism. In figure 9, we can observe the resulting speedup factors for a 32-taps FIR filter. The speedup is defined as the relation between the time that is required to compute a certain number of input samples with a processor architecture with one data path and the time required to compute the same number of input samples on the parallel processor.

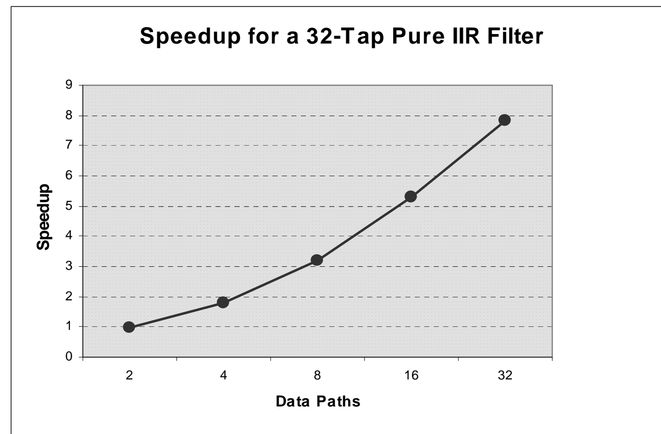

Fig. 10. Speedup Factor vs. Level of Parallelism for a 32-taps pure IIR Filter

Likewise, using equation (7) we have implemented a pure IIR filter in Matlab and processor code was automatically for different levels of parallelism. In figure 10, we can observe the resulting speedup factors for a 32-taps pure IIR filter.

F. Discussion

As we can observe from figures 9 and 10, the achievable speedup factor depends not only on the available processing power but it also depends on the data dependencies characteristics of the algorithm. In fact, due to the direct data dependencies of the pure IIR filter, the speedup factors we obtain for this algorithm are moderate in comparison with the speedup factors we obtain for the FIR filter. Moreover, in [17] we showed that the achievable speedup factor for pure recursive filters is upper bounded and once this upper bound has been reached no more gains on the speedup factor can be obtained even if the level of parallelism is incremented. These results illustrate the fact that in determining the optimal level of SIMD parallelism for a certain application not only power and area constraints have to be taken into account, but the characteristics of the algorithms play a paramount role. This stresses the necessity for a HW/SW design strategy, where this exploration can be supported.

IV. CONCLUSION

In this paper we have presented our HW/SW design methodology in order to support the design of scalable SIMD DSP cores. Taking as our starting point our novel STA microarchitecture, we explained how using our class library RNA we can express and manipulate STA-based processor cores in order to generate versions of our DSPs with different levels of parallelism. This enable us to build processor cores with the processing power needed to fulfill the requirements of a certain application in very short design cycles. An algebraic framework has been presented as a tool used to achieve the abstraction from the level of SIMD parallelism on the

algorithm design. Moreover, the algebraic framework provides the translation rules that enable the automatic generation from a high level matrix oriented language like Matlab. Finally, our results illustrated the necessity to explore not only hardware complexity issues but to also take into account the characteristics of the algorithm in order to establish the optimal level or parallelism.

ACKNOWLEDGMENT

We would like to thank the people of the CATS research group at the Technische Universitaet Dresden, from which we are honored to be members. Especially many thanks to Michael Hosemann for the hardware synthesis results.

REFERENCES

[1] G. P. Fettweis, "Embedded SIMD vector signal processor design," in *Proc. Third International Workshop on Systems, Architectures, Modeling and Simulation (Samos'2003)*, Samos, Greece, July 2003, pp. 71–76.
[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.
[3] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Synchronous transfer architecture (STA)," in *Lecture Notes on Computer Science*, S. Vassiliadis, Ed. Berlin, Germany: Springer-Verlag, July 2004, to be published.
[4] M. Weiss and G. Fettweis, "Dynamic codewidth reduction for vliw instruction set architectures in digital signal processors," in *Proceedings of the 3rd. Int. Workshop in signal and Image Processing IWSIP'96*, January 1996, pp. 571–520.
[5] RNA homepage. [Online]. Available: http://rna.cichon.com/
[6] A. Halambi and P. Grun, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proc. of the European Conference on Design, Automation and Test (DATE)*, Mar. 1999.
[7] V. Z. et al., "LISA - machine description language and generic machine model for HW/SW co-design," in *IEEE Workshop on VLSI Signal Processing*, 1996.
[8] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA - machine description language for cycle-accurate models of programmable DSP architectures," in *36th Design Automation Conference (DAC 99)*, New Orleans, June 1999.
[9] H. Seidel, E. Matus, G. Cichon, P. Robelly, M. Bronzel, and G. Fettweis, "Generated DSP cores for the implementation of an OFDM communications system," in *Lecture Notes on Computer Science*, S. Vassiliadis, Ed. Berlin, Germany: Springer-Verlag, July 2004, to be published.
[10] S. Muchnik, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers. Principles, Techniques, and Tools*. Redding, MA: Addison-Wesley, 1985.
[12] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Compiler scheduling for STA processors," In Proc. International Conference on Parallel Computing in Electrical Engineering PARELEC 2004. Dresden, Germany, Sept. 2004, to be published.
[13] M. Davio, "Kronecker products and shuffle algebra," *IEEE Trans. on Computers*, vol. C-30, no. 2, pp. 116–125, Feb. 1981.
[14] R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transform and convolution*. New York: Springer Verlag, 1997.
[15] M. Pueschel, B. Singer, M. Veloso, and J. Moura, "Fast automatic generation of DSP algorithms," in *Lecture Notes on Computer Science 2073*. Berlin, Germany: Springer-Verlag, 2001, pp. 97–106.
[16] M. Pueschel and J. Moura, "The algebraic approach to the discrete cosine and sine transforms and their fast algorithms," *SIAM Journal of Computing*, vol. 32, no. 5, pp. 1280–1316, 2003.
[17] J. Robelly, G.Cichon, H. Seidel, and G. Fettweis, "Implementation of recursive digital filters into vector SIMD DSP architectures," In Proc. International Conference on Acoustics, Speech and Signal Processing. ICASSP 2004. Montreal, Canada, May 2004, to be published.
[18] A. Feuer and G. C. Goodwin, *Sampling in Digital Signal Processing and Control*. Boston,Basel,Berlin: Birkhaeuser, 1996.

Pablo Robelly Pablo Robelly was born in Pasaje,Ecuador, in 1975. He received the B.S degree from the Army Polytechnic School ESPE, Quito, Ecuador in 1998, and the M.Sc./Dipl.-Ing. degree from the Technische Universitaet Dresden in 2001, all in electrical engineering.

During 1999 he joined the army research institute, Quito, Ecuador, where he worked developing real-time signal processing for army related projects. Since 2002, he is with the Vodafone chair for mobile communications systems at the Technische Universitaet Dresden as a research staff member.

His research interests are in the area of signal processing for mobile communications systems, signal processor architectures for mobile devices and compiler technology for signal processing.

Gordon Cichon Gordon Cichon was born in Munich, Germany in 1972. He received the Msc./Dipl.-Ing. from the Technische Universitaet Muenchen in 1996 in computer science.

From 1996 until 1999 he was with Philips Starnberg, where he worked on the development of chip sets for 3-D computer graphics. From 2000 until 2001, he joined Infineon in San Jose, USA, where he worked as system architect. Since 2001, he is research staff member at the Vodafone chair in Dresden, where he is working towards his Ph.D. on the area of low-power DSP architectures.

Hendrik Seidel Hendrik Seidel was born in Bochum, Germany, in 1977. He received the Msc./Dipl.-Ing. degree from the Technische Universitaet Dresden in 2002 in electrical engineering.

Since 2003 he is a research staff member at the Vodafone chair for mobile in Dresden. His research interests are in the are of SIMD-DSP processors, system design automation and embedded systems.

Gerhard Fettweis Prof. Dr. Gerhard Fettweis (Belgian and US American) was born in 1962, studied electrical engineering at the Aachen University of Technololgy (RWTH) in Germany and earned a PhD degree in 1990, having developed new integrated circuit architectures for high-speed digital communications.

From 1990 to 1991, he was Visiting Scientist at the IBM Almaden Research Center in San Jos, CA, developing signal processing innovations for IBM's 2.5" disk drive products. From 1991 to 1994, he was a Scientist with TCSI Inc., Berkeley, CA, responsible for signal processor development projects for mobile phone chip-sets. Since September 1994 he holds the Vodafone Chair at the Technische Universitaet in Dresden, Germany. In addition he cofounded Systemonic in 1999, he was the CTO of Systemonic (Dresden/San Jos/Boston), which at the size of nearly 100 employees, was successfully acquired by Philips Semiconductor in December 2002. Now he is Chief Scientist of Philips Semiconductors BL-C, and member of the management board (Vorstand) of Philips Semiconductors Dresden AG.

Gerhard Fettweis is an elected member of the IEEE Solid State Circuits Society's Board (Administrative Committee) since 1999, and was on the IEEE Communications Society Board of Governors from 1998 to 2000, respectively. Over the years he has organized and been on the program committee of numerous IEEE workshops and conferences, as well as given numerous invited talks. Furthermore, he serves on company supervisory boards and on industry as well as research institute's advisory committees.