UC Irvine

ICS Technical Reports

Title

High-level synthesis of scalable architectures for IIR filters using multichip modules

Permalink

<https://escholarship.org/uc/item/7373q63n>

Authors

Wang, Haigeng
Dutt, Nikil
Nicolau, Alexandru
et al.

Publication Date

1992

Peer reviewed

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES

Z

699

C3

no. 92-104

C.2

High-Level Synthesis of Scalable Architectures
for IIR Filters Using Multichip Modules

ICS Technical Report 92-104

Haigeng Wang Nikil Dutt Alexandru Nicolau Kai-Yeung S. Siu

Department of Information and Computer Science,
*Department of Computer and Electrical Engineering

High-Level Synthesis of Scalable Architectures for IIR Filters Using Multichip Modules *

Haigeng Wang Nikil Dutt Alexandru Nicolau Kai-Yeung Sunny Siu†
Department of Computer Science, †Department of Electrical & Computer Engineering
University of California, Irvine
Irvine, CA 92717

Abstract

We present a new technique for the high-level synthesis of *scalable*¹ MCM-based architectures implementing infinite-impulse response(IIR) filters. Our technique is based on the *regular schedules*, a class of parallel schedules for computing m th-order IIR filters. The simplicity of the regular schedules facilitates characterization of their inter-processor communications, which is generally difficult to express for parallel algorithms. The characterization of inter-processor communications of the regular schedules enables us to generate instruction-level behavior of the design that can be easily mapped onto MCM-based architectures. We illustrate this mapping of the regular schedules onto an MCM-based architecture by designing a special-purpose processor for the fifth-order elliptic wave filter. Our design yields a scalable performance measured in the filter's sample rate, which is not known to have been achieved by previously published designs. This work differs significantly from "traditional" high-level synthesis techniques in its emphasis on synthesizing scalable, high-performance multichip designs.

1 Introduction

High-performance IIR filters are demanded by many critical applications ranging from high-definition television, telecommunications to motion detection. The high-performance demanded by these applications has for long been predicted[4] to be achievable with the current VLSI technology using special-purpose parallel architectures. On the other hand, it has been known for long that the bottleneck of designing a scalable parallel architecture lies in the complicated data-dependences in the computation of IIR filters. However, the recent advance of techniques in parallelizing linear recurrence evaluation has enabled us to overcome the complicated data-dependences that exist in the IIR filtering process and hence provides a feasible path to building scalable architectures. In addition, the continuing progress of multichip modules(MCM) technology has provided an appealing platform for synthesizing parallel architectures for digital signal processing(DSP) applications. Combining parallel recurrence evaluation technique with MCM technology makes it feasible to synthesize scalable special-purpose parallel architectures for DSP applications containing recurrences.

The response $y(nT)$ of an m th-order IIR filter is described

with the following linear difference equations:

$$\begin{aligned} y(nT) = & a_0x(nT) + a_1x[(n-1)T] + a_2x[(n-2)T] + \cdots + \\ & + a_{k-1}x[(n-k+1)T] + b_1y[(n-1)T] + \\ & + b_2y[(n-2)T] + \cdots + b_my[(n-m)T]. \end{aligned} \quad (1)$$

Here the filter response at time nT , where T is the sample period and n is a running index, is the sum of product terms of m past responses, $y[(n-1)T], \dots, y[(n-m)T]$, where m is the *order* of the difference equation. The difference equation (1) is a special case of *band linear recurrences*, called *constant coefficient band linear recurrence*. The pattern of data dependences exhibited by (1) involves recurrences, since t is a linear combination of the outputs at time $t-1, t-2, \dots, t-m$. This recurrent dependence pattern is often referred to as a *loop-carried dependence*(LCD).

Difference equations are difficult to parallelize due to the LCD's, particularly when resource constraints are imposed. Previous research in scheduling for high level synthesis has exploited parallelism for signal processing in the forms of pipelining (i.e., overlapping the processing of a sample input with subsequent inputs), and concurrency (i.e., simultaneously computing operations that may be executed in parallel in processing one sample sequence while preserving data dependences). Although these scheduling techniques produce good results, we can further exploit parallelism by taking advantage of the recurrent patterns described by LCDs[16]. This is important for a difference equation (1) that exhibits LCDs, since there is not much parallelism within the LCD's: i.e., the speedup of throughput with current parallelization techniques is bounded by a small constant regardless of how many functional units are used.

In this paper, we derive a scalable algorithm and architecture for implementing IIR filters using MCM's. In Section 2, we briefly review the previous work in synthesizing parallel architectures for IIR filters. In Section 3, we present the *regular schedules*, a class of parallel schedules for computing m th-order IIR filters, derived using a technique for parallel linear recurrence evaluation[16]. In section 4, we describe the characteristics of our parallel processor on an MCM. We show how to generate designs of parallel processors for IIR filters starting with regular schedules by going through the design for a standard high-level synthesis benchmark, the 5th-order elliptic wave filter[2] in Section 5 and 6. In Section 7, we show the performance of our design using a practical number of number of processors implementable on an MCM in comparison with designs based on some of the previously published best scheduling techniques in design automation and digital signal processing. Section 8 concludes the paper.

*This work was supported in part by NSF grants CCR8704367, MIP9009239 and ONR grant N0001486K0215.

¹A parallel architecture or its performance is said to be *scalable* if the performance of the architecture improves proportionally as the number of processors or functional units increases.

2 Related Work

Related work has been done in two areas: scheduling algorithms for high level synthesis and parallel computing. This work differs in scope and significance from the previous work in high-level synthesis in two ways. First, we do not restrict ourselves to the synthesis of a single chip with a limited number of functional units. Instead, we address the synthesis of scalable MCM architectures for achieving high-performance. Second, we derive schedules that exploit parallelism of recurrences beyond LCD's.

Traditionally, scheduling algorithms for high level synthesis have been exploiting three forms of parallelism: concurrency, functional pipelining and loop pipelining[2]. Since in general the problem of scheduling with resource constraints is NP-complete, heuristic-based loop pipelining techniques [6, 8] have been developed to compact loops with given resource constraints. Percolation-based loop pipelining techniques [14] first compact a loop into its optimal parallel counterpart and then apply resource constraints on the parallel version. Static scheduling via optimum unfolding[13] in DSP respects the inherent lower bound on the achievable iteration period imposed by the recursions in the program, thus is essentially equivalent to aggressive software pipelining in terms of resulting parallelized programs. All of the techniques above preserve the data(true) dependences(LCD is a special case of data dependence), and hence do not try to reduce the critical path by exploiting parallelism beyond data dependences.

Techniques that extract parallelism beyond data dependences have just started to receive attention in the high-level synthesis and digital signal processing communities. A few techniques based on tree height reduction [7, 11, 9] have been proposed to shorten, using extra operations, the critical paths consisting of a sequence of associative operations, e.g., a sequence of additions or a sequence of multiplications. But these techniques do not extract parallelism in recurrent loops involving additions and multiplications beyond LCD's that have dependence distances of one, and hence can at best achieve a speedup bounded by constant as the number of functional units increases. These scheduling techniques therefore cannot be applied to efficiently exploit scalable MCM architecture. A scheduling method for successive approximation of optimal schedules for IIR filter computation described with linear differences beyond LCD's was first studied in [16] on an ideal machine model with a fixed number of identical multi-functional processors. However, generating scalable schedules for IIR filters beyond LCD's on more realistic architectures, such as one with inter-processor communications and with different types of functional units having different time constraints, has not been studied. The regular schedules presented in this paper aim to solve this problem.

In the field of parallel computing, parallel evaluation of linear recurrences has been studied for quite some time [1, 5, 17], for which the review is given in [18]. However, the underlying model differs from high-level synthesis, since parallel computing typically deals with a number of identical, multi-function processors, while high-level synthesis deals with a variety of functional units that differ in several attributes (e.g., functionality, cost, speed). We briefly review some results from parallel computing. Chen and Kuck developed an algorithm [1] for computing N m -th order band linear recurrence with a fixed number of p processors, independent of the problem size N . Gajski[5] lowered the time bound of Chen and Kuck's for computing the band linear recurrences with p processors. Wang and Nicolau[17] proposed the Harmonic Scheduling technique for deriving optimal schedules for band linear recurrences with p processors and their schedules improved upon Gajski's in execution time. In particular, they showed that their schedules achieved the strict time lower

bound for $m = 1, 2$. In addition, their method facilitates deriving schedules with best program-space efficiency (so that a schedule can better fit into caches), given an execution time. Because of the ideal architectural assumptions, these parallel processing techniques cannot be directly applied to the custom synthesis problem we address in this paper.

3 A Scalable Algorithm for IIR Filters

Given a linear difference equation (1) for m th-order IIR filters, the idea of a scalable algorithm is to express the linear difference equation in terms of matrix chain multiplication. Using associativity of matrix chain multiplication, we can compute *redundant* look-ahead results and produce multiple sample outputs in parallel. In principle, signal processing programs process infinite time series and produce infinite time series and thus are nonterminating in nature, i.e., the problem size in the DSP context is considered infinitely large.

The scalable algorithm to be described can be derived using the Harmonic Scheduling technique[16]. For illustration, we describe our parallel algorithm with the simplest linear recurrence, called *parallel prefix*, defined as follows, $x_0 = a_0$, $x_i = x_{i-1} + a_i$, $i > 0$. We express it in matrix chain multiplication as follows:

$$[x_0 \ 1] \begin{bmatrix} 1 & 0 \\ a_1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ a_2 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 \\ a_i & 1 \end{bmatrix} \cdots \quad (2)$$

where $X_i = [x_i \ 1]$ and

$$M_i = \begin{bmatrix} 1 & 0 \\ a_i & 1 \end{bmatrix}.$$

Similarly, linear recurrence equations for IIR filters of any order can be expressed in matrix chain multiplication.

Algorithm 3.1 gives a generic description of the regular schedule in terms of matrix representation for computing linear recurrence describing IIR filters. It is also called a *square* schedule as each iteration of its outermost loop produces p^2 sample outputs. The regular schedules give suboptimal yet scalable performance under SIMD concurrent-read-exclusive-write(CREW) parallel machine model.

Algorithm 3.1

[Regular Schedule]

Input: Matrix chain expression of the m th-order linear difference equation describing time response of the filter and sequence of sample inputs to the filter.

Output: Sequence of sample outputs of the filter.

```

    for ( $i = 0$ ; true;  $i += p^2$ )
    {
        for ( $j = i + 2 + (m - 1) \times p$ ;  $j \leq i + m \times p$ ;  $j += p$ )
            do in parallel on  $p$  processors
                for ( $k = j$ ;  $k \leq j + p$ ;  $k +=$ )
                     $M_k = M_{k-1} \times M_k$ ;
                for ( $j = i$ ;  $j \leq i + (p - 1) \times p$ ;  $j += p$ )
                    for ( $l = 1$ ;  $l \leq p$ ;  $l +=$ )
                        do in parallel on  $p$  processors
                             $X_{j+l} = X_j \times M_{j+l}$ ;
    }
```

□ The outermost loop partitions the problem size(i.e., the infinite sample input series) into chunks of size p^2 , called *period*, given p processors. Each iteration of the outermost loop computes p^2 final values(i.e., sample outputs) of a period, called *current period*. In an iteration of the outermost loop, two inner-loops in sequence partition a period further into p sections of size p each, called a *redundant tree*. The first inner-loop computes redundant values: its outer-loop computes in parallel the redundant values in p sections of the current period and its inner-loop computes sequentially the redundant values in each redundant tree. The second inner-loop of the outermost loop

computes the final values of the current period. Its outer-loop travels sequentially computing the final values of each of the p sections in the period, and its inner-loop (a do-all type of loop) computes all final values in a section in parallel.

Take $p = 4$ as an example. The computation tree for the current period is shown in Figure 1. The outermost loop iterates on periods of size $p^2 = 4^2 = 16$ and an iteration produces all 16 final values (the X 's in Figure 1) of a period in the following fashion. The outer-loop of the first inner-loop is do-all loop, it sends all of its $p = 4$ iterations, each computing $p - 1$ redundant values of a redundant tree, to execution in parallel on $p = 4$ processors. For example, R_2, R_3, R_4 are the three matrices of redundant values on the first redundant tree in Figure 1. Each iteration, i.e., the innermost loop computes $p - 1 = 3$ redundant values sequentially, thus it takes $(p - 1)T_{m-by-m}$ (time for a matrix-by-matrix multiplication) time to complete redundant computation in a period.

Figure 1: Parallel prefix computation tree for $p = 4$ processors.

The second inner-loop computes all final values in a period using redundant values produced by the first inner-loop of the algorithm. The outer-loop of the second inner-loop is a sequential loop, iterating on $p = 4$ redundant trees in a period, i.e., each iteration (the second innermost loop) produces the final results on top of a redundant tree in parallel in time of a vector-by-matrix multiplication. Since there are p redundant trees in a period, the time for computing all final values in a period equals pT_{v-by-m} (time for a vector-by-matrix multiplication).

In general, the performance of the algorithm can be determined as follows. First, the execution time for computation of a period is

$$\begin{aligned} T_p(\text{period of size } p) &= (p - 1)T_{m-by-m} + pT_{v-by-m} \quad (3) \\ &\leq (2p - 1) \max\{T_{m-by-m}, T_{v-by-m}\}. \end{aligned}$$

Since it takes the same amount of time to compute the final values of a period and the filtering process is nonterminating by nature, sample rate SR_p of the IIR filter on a design with p processors will be a good indicator of the performance, which is

$$\begin{aligned} SR_p &= \frac{\text{(number of outputs produced in a period)}}{T_p(\text{period of size } p)} \quad (4) \\ &= \frac{p^2}{T_p(\text{period of size } p)} \\ &\geq \frac{p^2}{(2p - 1) \max\{T_{m-by-m}, T_{v-by-m}\}}. \end{aligned}$$

Since T_{m-by-m} and T_{v-by-m} depend only on the order m of the filter, they become constants once m is given, so does $\max\{T_{m-by-m}, T_{v-by-m}\}$. Hence, the sample rate $SR_p = O(p)$ improves as the number of processors increases, i.e., the algorithm gives a scalable performance under the ideal SIMD CREW parallel machine model.

Although this technique looks promising, the introduction of realistic technological constraints and custom synthesis assumptions (e.g., dissimilar functional units on an MCM and inter-processor communications) may greatly affect this ideal performance. Hence the effect of realistic technology and synthesis

constraints on the scalable performance of this schedule remains an open problem.

4 The MCM Architecture Characteristics

Figure 2: MCM with p special-purpose processors.

The MCM technology is particularly suited for the implementation of scalable, high-performance architectures since additional processors can be added to an MCM without incurring a large penalty in off-chip delays. For the high-level synthesis of IIR filters using scalable MCM architectures, we first need to define the target architecture of a single MCM. Figure 2 shows our model of a multichip module that carries up to 16 processors of 32 bits. Each processor is built on a die. Every processor within the module is connected to multiple buses on the carrier, through which processors in the same module communicate. We assume that the inter-processor (i.e., inter-chip, on-substrate) delay is one cycle. A processor is composed of an adder, a multiplier, a register file (RF) and a control unit. Our simple processor does not need memory since the sample outputs can be stored in registers and transferred out to storage devices in parallel with computation such that no register file overflow will occur. For simplicity, we do not use functional pipelining for the functional units in a processor. There are 7 read ports and 5 write ports for the register file in a processor. Each functional unit (an adder or a multiplier) requires 2 operands (from the 2 dedicated read ports in the RF), and produces 1 output (to the dedicated write port in the RF). The remaining 3 read ports and 3 write ports in each RF are connected to the bidirectional buses. All processors in a multichip module operate synchronously.

Figure 3: Top: The format of a typical instruction. Bottom: The relative timing diagram of the instruction.

Figure 3 shows the format of a typical long-word instruction and timing for its execution (R and S are move instructions that receive from and send to buses). Each processor is similar to a VLIW machine [3], in that it issues a long instruction word each time. The execution time of a long instruction word equals the maximum of its constituent instruction times. To simplify the control unit design, vertical compaction is not allowed. Any of the component instructions in a long instruction can be a no-op.

We end this section with a summary [10] of our MCM architecture characteristics shown in Table 1. Note that the specific

ranges of the parameters are chosen as shown in Table 1 for simulating the example in the following sections with realistic constraints and in fact our technique applies to general ranges of the parameters.

parameter	specification
# of processors on die	up to 16 of 32 bits
FU's in a processor	an FP adder, an FP multiplier
cycle time	7ns
operation time	
integer A & L operations	1 cycle
FP addition	7 cycles
FP multiplication	10 cycles
inter-processor data transfer	1 cycle
inter-processor communication	3 data buses
# of ports on an RF	7 read ports, 5 write ports

Table 1: MCM architecture features.

5 Deriving the Schedule

This section generates the schedule for computing the filter using Algorithm 3.1 which guides the design of our architecture with a regular pattern. Derived from the original transfer function[12], the time response of the 5th-order elliptic filter is described with a 5th-order constant-coefficient linear recurrence equation as follows($x(t)$ and $y(t)$ in the equation are also written as x_t and y_t respectively):

$$y(t) = b_0x(t) + b_1x(t-1) + b_2x(t-2) + b_3x(t-3) + b_4x(t-4) + b_5x(t-5) + a_1y(t-1) + a_2y(t-2) + a_3y(t-3) + a_4y(t-4) + a_5y(t-5). \quad (5)$$

Equation (5) can be expressed as a matrix chain multiplication,

$$\begin{bmatrix} y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ 1 \end{bmatrix}^T \begin{bmatrix} a_1 & 1 & 0 & 0 & 0 & 0 \\ a_2 & 0 & 1 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 1 & 0 & 0 \\ a_4 & 0 & 0 & 0 & 1 & 0 \\ a_5 & 0 & 0 & 0 & 0 & 0 \\ \vec{b}\vec{x}_6 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \dots \begin{bmatrix} a_1 & 1 & 0 & 0 & 0 & 0 \\ a_2 & 0 & 1 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 1 & 0 & 0 \\ a_4 & 0 & 0 & 0 & 1 & 0 \\ a_5 & 0 & 0 & 0 & 0 & 0 \\ \vec{b}\vec{x}_t & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \dots, \quad (6)$$

where $\vec{b} = (b_0 b_1 b_2 b_3 b_4 b_5)$ is a vector of the constant coefficients for sample inputs x 's, and $\vec{x}_t = (x_t x_{t-1} x_{t-2} x_{t-3} x_{t-4} x_{t-5})$ is a vector of sample inputs. We use M_t to stand for matrix

$$\begin{bmatrix} a_1 & 1 & 0 & 0 & 0 & 0 \\ a_2 & 0 & 1 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 1 & 0 & 0 \\ a_4 & 0 & 0 & 0 & 1 & 0 \\ a_5 & 0 & 0 & 0 & 0 & 0 \\ \vec{b}\vec{x}_t & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (7)$$

We illustrate the derivation of the regular schedule for the elliptic filter on $p = 6$ processors per MCM. Applying Algorithm 3.1 to Equation (5), we obtained the computation tree(i.e., the

Figure 4: The computation tree in matrix chain multiplication for the 5th-order elliptic filter on $p = 6$ processors.

dependence tree) for a period of resulting schedule in terms of matrix chain multiplications as shown in Figure 4.

Since Equation (5) is a constant-coefficient linear recurrence, our final schedule should take advantage of the property that most elements in the redundant matrix products can be determined statically(i.e., off-line or at compile time) when the number of processors is given. Our schedule dynamically computes(i.e., on-line or in real time) only those elements of the matrix products dependent on the sample inputs and previously produced sample outputs. To determine what elements can be determined statically and what should be computed dynamically, it suffices to observe the first five redundant matrix products in a redundant tree, say M_6M_7 , $M_6M_7M_8$, $M_6M_7M_8M_9$, $M_6M_7M_8M_9M_{10}$, $M_6M_7M_8M_9M_{10}M_{11}$. In general, the number and the organization of statically-determined and dynamically-computed elements remains the same for a matrix chain product of greater than $(m-1)$ matrices for m th-order linear recurrences.

Table 2 shows the regular schedule generated using Algorithm 3.1 for Equation (5) realizing the 5th-order elliptic filter with $p = 6$ processors. Note that this schedule is expressed in terms of products of matrix elements to facilitate understanding. The instruction-level schedule can be expanded out of Table 2 in a straightforward fashion by replacing matrix elements with the corresponding operations. The columns titled "Processor 1" through "Processor 6" specify the computation all $p = 6$ processors do in each of the $2p = 12$ stages. The parallel schedule consists of a loop body iterating on the induction variable $k \geq 0$. The pairs of numbers in the first column count the number of multiplications and additions for a single processor in a stage by expanding the vector inner products with operations. Stage 1 computes linear combinations of the sample inputs. Stages 2 to $p = 6$ of the schedule, as placed in the second box, compute redundant values that depend on the sample inputs of the current period, with the j th processor computing the redundant values on the j th redundant tree of the period. In the second stage of the loop body, $(6,1)_{6^{2k+6j}} = \vec{b}\vec{x}_{6^{2k+6j}}$, $1 \leq j \leq 6$ and $k \geq 0$. The computations in stage 2 through 6 are dependent on increasing number of values previously computed and $p = 6$ current sample inputs $\vec{x}_t = (x_t x_{t-1} x_{t-2} x_{t-3} x_{t-4} x_{t-5})$.

The remaining $p = 6$ stages (step $p+1 = 7$ to $2p = 12$), as shown in the box at bottom, compute the final values of the current period with all processors computing the final values on the j th redundant tree in parallel in stage $p+j = j+6$. Vector $\vec{y}_{6^{2k+6j-1}} = (y_{6^{2k+6j-1}} y_{6^{2k+6j-2}} y_{6^{2k+6j-3}} y_{6^{2k+6j-4}} y_{6^{2k+6j-5}} 1)$ serves as the "parallel prefix" to be broadcast to all processors for producing final values in parallel. Vector $(i,1)_{6^{2k+6j+l}} = ((1,1)_{6^{2k+6j+l}}, (2,1)_{6^{2k+6j+l}}, (3,1)_{6^{2k+6j+l}}, (4,1)_{6^{2k+6j+l}}, (5,1)_{6^{2k+6j+l}}, (6,1)_{6^{2k+6j+l}})$, $1 \leq j \leq 6$ and $0 \leq l \leq 5$, consists of $p = 6$ elements of redundant values, of which the 6th element depends on the sample inputs and the first 5 elements do not. These redundant values are independent of the sample

Stage	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5	Processor 6
1 (6,6)	$(6,1)_{6^2k+6} =$ $\tilde{b}_{6^2k+6}$	$(6,1)_{6^2k+12} =$ $\tilde{b}_{6^2k+12}$	$(6,1)_{6^2k+18} =$ $\tilde{b}_{6^2k+18}$	$(6,1)_{6^2k+24} =$ $\tilde{b}_{6^2k+24}$	$(6,1)_{6^2k+30} =$ $\tilde{b}_{6^2k+30}$	$(6,1)_{6^2k+36} =$ $\tilde{b}_{6^2k+36}$
2 (7,7)	$(6,1)_{6^2k+7} =$ $a_1(6,1)_{6^2k+6} +$ $\tilde{b}_{6^2k+7}$	$(6,1)_{6^2k+13} =$ $a_1(6,1)_{6^2k+12} +$ $\tilde{b}_{6^2k+13}$	$(6,1)_{6^2k+19} =$ $a_1(6,1)_{6^2k+18} +$ $\tilde{b}_{6^2k+19}$	$(6,1)_{6^2k+25} =$ $a_1(6,1)_{6^2k+24} +$ $\tilde{b}_{6^2k+25}$	$(6,1)_{6^2k+31} =$ $a_1(6,1)_{6^2k+30} +$ $\tilde{b}_{6^2k+31}$	$(6,1)_{6^2k+37} =$ $a_1(6,1)_{6^2k+36} +$ $\tilde{b}_{6^2k+37}$
3 (8,8)	$(6,1)_{6^2k+8} =$ $a_1(6,1)_{6^2k+7} +$ $a_2(6,1)_{6^2k+6} +$ $\tilde{b}_{6^2k+8}$	$(6,1)_{6^2k+14} =$ $a_1(6,1)_{6^2k+13} +$ $a_2(6,1)_{6^2k+12} +$ $\tilde{b}_{6^2k+14}$	$(6,1)_{6^2k+20} =$ $a_1(6,1)_{6^2k+19} +$ $a_2(6,1)_{6^2k+18} +$ $\tilde{b}_{6^2k+20}$	$(6,1)_{6^2k+26} =$ $a_1(6,1)_{6^2k+25} +$ $a_2(6,1)_{6^2k+24} +$ $\tilde{b}_{6^2k+26}$	$(6,1)_{6^2k+32} =$ $a_1(6,1)_{6^2k+31} +$ $a_2(6,1)_{6^2k+30} +$ $\tilde{b}_{6^2k+32}$	$(6,1)_{6^2k+38} =$ $a_1(6,1)_{6^2k+37} +$ $a_2(6,1)_{6^2k+36} +$ $\tilde{b}_{6^2k+38}$
4 (9,9)	$(6,1)_{6^2k+9} =$ $a_1(6,1)_{6^2k+8} +$ $a_2(6,1)_{6^2k+7} +$ $a_3(6,1)_{6^2k+6} +$ $\tilde{b}_{6^2k+9}$	$(6,1)_{6^2k+15} =$ $a_1(6,1)_{6^2k+14} +$ $a_2(6,1)_{6^2k+13} +$ $a_3(6,1)_{6^2k+12} +$ $\tilde{b}_{6^2k+15}$	$(6,1)_{6^2k+21} =$ $a_1(6,1)_{6^2k+20} +$ $a_2(6,1)_{6^2k+19} +$ $a_3(6,1)_{6^2k+18} +$ $\tilde{b}_{6^2k+21}$	$(6,1)_{6^2k+27} =$ $a_1(6,1)_{6^2k+26} +$ $a_2(6,1)_{6^2k+25} +$ $a_3(6,1)_{6^2k+24} +$ $\tilde{b}_{6^2k+27}$	$(6,1)_{6^2k+33} =$ $a_1(6,1)_{6^2k+32} +$ $a_2(6,1)_{6^2k+31} +$ $a_3(6,1)_{6^2k+30} +$ $\tilde{b}_{6^2k+33}$	$(6,1)_{6^2k+39} =$ $a_1(6,1)_{6^2k+38} +$ $a_2(6,1)_{6^2k+37} +$ $a_3(6,1)_{6^2k+36} +$ $\tilde{b}_{6^2k+39}$
5 (10,10)	$(6,1)_{6^2k+10} =$ $a_1(6,1)_{6^2k+9} +$ $a_2(6,1)_{6^2k+8} +$ $a_3(6,1)_{6^2k+7} +$ $a_4(6,1)_{6^2k+6} +$ $\tilde{b}_{6^2k+10}$	$(6,1)_{6^2k+16} =$ $a_1(6,1)_{6^2k+15} +$ $a_2(6,1)_{6^2k+14} +$ $a_3(6,1)_{6^2k+13} +$ $a_4(6,1)_{6^2k+12} +$ $\tilde{b}_{6^2k+16}$	$(6,1)_{6^2k+22} =$ $a_1(6,1)_{6^2k+21} +$ $a_2(6,1)_{6^2k+20} +$ $a_3(6,1)_{6^2k+19} +$ $a_4(6,1)_{6^2k+18} +$ $\tilde{b}_{6^2k+22}$	$(6,1)_{6^2k+28} =$ $a_1(6,1)_{6^2k+27} +$ $a_2(6,1)_{6^2k+26} +$ $a_3(6,1)_{6^2k+25} +$ $a_4(6,1)_{6^2k+24} +$ $\tilde{b}_{6^2k+28}$	$(6,1)_{6^2k+34} =$ $a_1(6,1)_{6^2k+33} +$ $a_2(6,1)_{6^2k+32} +$ $a_3(6,1)_{6^2k+31} +$ $a_4(6,1)_{6^2k+30} +$ $\tilde{b}_{6^2k+34}$	$(6,1)_{6^2k+40} =$ $a_1(6,1)_{6^2k+39} +$ $a_2(6,1)_{6^2k+38} +$ $a_3(6,1)_{6^2k+37} +$ $a_4(6,1)_{6^2k+36} +$ $\tilde{b}_{6^2k+40}$
6 (11,11)	$(6,1)_{6^2k+11} =$ $a_1(6,1)_{6^2k+10} +$ $a_2(6,1)_{6^2k+9} +$ $a_3(6,1)_{6^2k+8} +$ $a_4(6,1)_{6^2k+7} +$ $a_5(6,1)_{6^2k+6} +$ $\tilde{b}_{6^2k+11}$	$(6,1)_{6^2k+17} =$ $a_1(6,1)_{6^2k+16} +$ $a_2(6,1)_{6^2k+15} +$ $a_3(6,1)_{6^2k+14} +$ $a_4(6,1)_{6^2k+13} +$ $a_5(6,1)_{6^2k+12} +$ $\tilde{b}_{6^2k+17}$	$(6,1)_{6^2k+23} =$ $a_1(6,1)_{6^2k+22} +$ $a_2(6,1)_{6^2k+21} +$ $a_3(6,1)_{6^2k+20} +$ $a_4(6,1)_{6^2k+19} +$ $a_5(6,1)_{6^2k+18} +$ $\tilde{b}_{6^2k+23}$	$(6,1)_{6^2k+29} =$ $a_1(6,1)_{6^2k+28} +$ $a_2(6,1)_{6^2k+27} +$ $a_3(6,1)_{6^2k+26} +$ $a_4(6,1)_{6^2k+25} +$ $a_5(6,1)_{6^2k+24} +$ $\tilde{b}_{6^2k+29}$	$(6,1)_{6^2k+35} =$ $a_1(6,1)_{6^2k+34} +$ $a_2(6,1)_{6^2k+33} +$ $a_3(6,1)_{6^2k+32} +$ $a_4(6,1)_{6^2k+31} +$ $a_5(6,1)_{6^2k+30} +$ $\tilde{b}_{6^2k+35}$	$(6,1)_{6^2k+41} =$ $a_1(6,1)_{6^2k+40} +$ $a_2(6,1)_{6^2k+39} +$ $a_3(6,1)_{6^2k+38} +$ $a_4(6,1)_{6^2k+37} +$ $a_5(6,1)_{6^2k+36} +$ $\tilde{b}_{6^2k+41}$
7 (5,5)	$y_{6^2k+6} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+6}$	$y_{6^2k+7} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+7}$	$y_{6^2k+8} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+8}$	$y_{6^2k+9} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+9}$	$y_{6^2k+10} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+10}$	$y_{6^2k+11} =$ $\tilde{y}_{6^2k+5}$ $(i,1)_{6^2k+11}$
8 (5,5)	$y_{6^2k+12} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+12}$	$y_{6^2k+13} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+13}$	$y_{6^2k+14} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+14}$	$y_{6^2k+15} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+15}$	$y_{6^2k+16} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+16}$	$y_{6^2k+17} =$ $\tilde{y}_{6^2k+11}$ $(i,1)_{6^2k+17}$
9 (5,5)	$y_{6^2k+18} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+18}$	$y_{6^2k+19} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+19}$	$y_{6^2k+20} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+20}$	$y_{6^2k+21} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+21}$	$y_{6^2k+22} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+22}$	$y_{6^2k+23} =$ $\tilde{y}_{6^2k+17}$ $(i,1)_{6^2k+23}$
10 (5,5)	$y_{6^2k+24} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+24}$	$y_{6^2k+25} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+25}$	$y_{6^2k+26} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+26}$	$y_{6^2k+27} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+27}$	$y_{6^2k+28} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+28}$	$y_{6^2k+29} =$ $\tilde{y}_{6^2k+23}$ $(i,1)_{6^2k+29}$
11 (5,5)	$y_{6^2k+30} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+30}$	$y_{6^2k+31} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+31}$	$y_{6^2k+32} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+32}$	$y_{6^2k+33} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+33}$	$y_{6^2k+34} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+34}$	$y_{6^2k+35} =$ $\tilde{y}_{6^2k+29}$ $(i,1)_{6^2k+35}$
12 (5,5)	$y_{6^2k+36} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+36}$	$y_{6^2k+37} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+37}$	$y_{6^2k+38} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+38}$	$y_{6^2k+39} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+39}$	$y_{6^2k+40} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+40}$	$y_{6^2k+41} =$ $\tilde{y}_{6^2k+35}$ $(i,1)_{6^2k+41}$

Table 2: The parallel schedule generated using Algorithm 3.1 for Equation (5) realizing the 5th-order elliptic filter.

inputs and outputs and have been determined statically.

6 Mapping Schedule onto MCM

Once the schedule is fixed, we are ready to bind it to our target MCM architecture. In order to map the schedule onto our multichip module and to specify the design of the control units, we need to place into the schedule input, output and data transfer instructions as mentioned above. Recall that one of the advantages of the regular schedule is its simple inter-processor data communication pattern. Two kinds of inter-processor data communications are involved in the computation: point-to-point data transfer and broadcast. Point-to-point data transfers are effected during and after redundant value computation. Referring to Figure 4, every first value on the top fringe of a redundant tree in a period is sent to the first processor before it is used, and in general, all of the i th values on the top fringe of the p redundant trees are sent to the i th processor before they are used, $1 \leq i \leq p$. Equivalently, in Table 2, all the results produced in the i th stage are sent to the i th processor before they are used.

Broadcasts send the “prefix vectors” of the last $m = 5$ final

values on top of a redundant tree to all processors for producing final values on top of next redundant tree in final value computation in the loop body (see Figure 4). Specifically in Table 2, the final values produced by processor 2 through 5 are broadcasted after each of the stages $p = 6$ through $2p = 12$.

Figure 5 illustrates sections of the timing diagram of instruction execution of our schedule in Table 2 on $p = 6$ processors. The 0th cycle through 66th cycle (recall that multiplication, addition and inter-processor communication take 10, 7 and 1 cycle respectively) does the computation, 6 multiplications and 6 additions, specified in the first stage of Table 2. In general, the multiplications and additions can be overlapped as shown, which can be seen by expanding the matrix-level computation in Table 2 with operation-level multiplications and additions. The instructions of the second stage that overlap with the trailing instructions of the first stage are shaded in Figure 5. The second stage starts at the 60th cycle with its first multiplication and the trailing addition of the first stage is issued in the same long instruction. The data transfers of redundant values produced in the first stage are issued in succeeding instructions on the 70th and the 80th cycle respectively, after the trailing

Figure 5: Timing diagram of instruction execution of the loop body of the schedule in Table 2.

Figure 6: Instructions issued in the 70th and the 80th cycles in Figure 5.

Figure 7: Instructions issued in the 576th and 577th cycles in Figure 5.

addition of the preceding stage completes—this is enclosed by a dotted rectangle and detailed in Figure 6. In Figure 6, for the instructions issued at the 70th cycle, processors P2, P3 and P4 send their redundant values to the three buses (bus B1 to B3) and processor P1 receives them in the same cycle. In the instructions issued at the 80th cycle, processors P5 and P6 send their redundant values to bus B1 and bus B2 and P1 receives them in the same cycle.

By counting the number of multiplications and additions in each stage, we determine that the trailing addition of the 6th stage completes by the 517th cycle, and that two cycles of data transfers of the redundant values produced in that stage follow immediately and complete at the 519th cycle. we chose not to issue other operations in parallel with the last two cycles of redundant data transfers for simplicity.

The first stage of final value computation, i.e., the 7th stage starts at the 519th cycle and completes by the 576th cycle, referring to the count of operations in Table 2. Two cycles of broadcasting the final values produced by Processor 2 through Processor 6 are issued and no operations of the next stage are issued in parallel because each final computation except for the first has $m = 5$ data dependences on the preceding stage. The

remaining $p - 1 = 5$ stage of final value computation can be done similarly. The broadcasting of final values in the first final computation stage, as enclosed by a dotted rectangle, is detailed in Figure 7. In Figure 7, in the instructions issued at the 576th cycle, processors P4, P5 and P6 send their final values to all three buses and each processor receives in the same cycle the data on the buses broadcasted by the other processors. In the instructions issued at the 577th cycle, processors P2 and P3 broadcast their final values to bus B2 and bus B3 respectively and each processor receives in the same cycle the data broadcasted by the other processors.

We presented the technique used to map a regular schedule for IIR filters onto our MCM architecture outlined in Section 4.

7 Results

We now describe our results. Table 3 displays the performance measured by sample period of the regular schedules, percolation scheduling with tree-height reduction (PS+THR) and column sweep algorithm. Table 4 shows the performance improvement obtained by our regular schedules over the other two algorithms. The sample period is the average time in number of cycles required to produce a sample output, the reciprocal of which is the filter's sample rate. The details are given in [18]. The column titled “# FU's” gives the number of adders and multipliers used. Recall that a single processor in our MCM architecture has an adder and a multiplier. The column titled “our schedule” displays the sample period yielded by our regular schedule for each number of processors. The column titled “PS+THR” lists the sample period achieved using percolation scheduling with tree-height reduction and the improvements by our schedule on PS+THR, determined by the ratio of the sample period by PS+THR to that of our schedule. The column titled “column sweep” shows similar attributes to its left column.

# FU's	our schedule sample period	PS+THR sample period	column sweep sample period
2*, 4+		112	
6*, 6+	23.97	112	33
7*, 7+	21.22	112	28.29
8*, 8+	19.19	112	24.75
9*, 9+	17.59	112	22
10*, 10+	16.32	112	19.8
11*, 11+	15.29	112	18
12*, 12+	14.42	112	18
13*, 13+	13.69	112	18
14*, 14+	13.07	112	18
15*, 15+	12.53	112	18
16*, 16+	12.06	112	18

Table 3: Performance measured by sample period.

# FU's	improves on PS+THR	improves on on column sweep
2*, 4+		
6*, 6+	4.67	1.38
7*, 7+	5.28	1.33
8*, 8+	5.84	1.29
9*, 9+	6.37	1.25
10*, 10+	6.86	1.21
11*, 11+	7.33	1.18
12*, 12+	7.77	1.25
13*, 13+	8.18	1.32
14*, 14+	8.57	1.38
15*, 15+	8.94	1.44
16*, 16+	9.29	1.49

Table 4: Improvements in performance.

PS+THR[11] scheduled on the dependence graph of the ladder implementation of the 5th-order elliptic filter[2] and achieved optimal sample rate with respect to LCD's with 2 multipliers and 4 adders. But it does not yield further performance improvement beyond 2 multipliers and 4 adders, as

shown in Table 3. Both our regular schedule and the column sweep algorithm schedule on the parallel linear difference implementation of the filter and the sequential linear difference implementation is faster than the ladder implementation of the filter. The column-sweep algorithm is among the fastest for parallel evaluation for linear differences with respect to LCD's and has superior or equal time complexity to the many DSP scheduling algorithms [13] we surveyed. The sample period of column sweep reduces when number of processors increases from $p = 6$ to $p = 11$ and is bounded below by 18 cycles thereafter, i.e., it does not yield more performance improvements beyond $p = 11$ processors. The non-scalability of column sweep results from the fact that it does not introduce redundant computations and hence the scope of its look-ahead computation is limited to producing a single sample output per column sweep.

8 Conclusion

We have presented an efficient high-level synthesis technique for IIR digital filters using a realistic scalable MCM-based architecture. The design is based on regular schedules of linear difference equations, derived using techniques for parallel linear recurrence evaluation. The scalability of the parallel architecture follows as a result of the regular schedule. Our design offers a performance that improves proportionally as the number of processing elements increases on a single MCM. This is a result not known to have been achieved by previously published designs, as shown in the illustration of design for the 5th-order elliptic filter. We are currently studying the scalable designs using multiple MCM's with a wider range of architectural parameters.

One major difficulty in automatically generating designs of parallel processors for IIR filters is in characterizing the inter-processor communications, which is not expressed by the algorithms in high-level languages. The simplicity of the regular schedules facilitates characterization of their inter-processor communications, which enables us to generate instruction-level behavioral descriptions of designs combined with the computations specified in the high-level algorithm, i.e., we efficiently map the regular schedules onto an MCM-based architecture. The characterization of inter-processor communications combined with precise time complexity analysis of the algorithm allows us to determine the performance formula of our designs with closed form as shown, and to trade off design choices.

We have not addressed the important issue of numerical stability in digital filter design. Many "fast" implementation of digital filters may achieve speedup at the expense of numerical instability. On the other hand, there are some well known numerically stable realizations of digital filters that seem to prevent from efficient parallel implementation. For example, although realizing wave digital filters in lattice or ladder configurations often exhibits good numerical properties concerning coefficient accuracy requirements, dynamic range, and other aspects of stability under finite-precision arithmetic, its complex dependences may intrinsically hinder scalable parallel implementations, as previously published designs have indicated [9, 11]. It has been proven recently in [15] that a parallel algorithm of linear recurrences based on tree-height reduction, on number of processors proportional to the problem size, achieves equivalent numerical stability as a sequential algorithm. Furthermore, the success of parallel linear recurrence routines in the linear algebra packages on many commercially-available parallel supercomputers such as Cray series and Convex series in the past few years provide more evidence for the numerical stability of linear difference equation implementation of IIR filters.

Acknowledgements

The authors wish to thank Professors Dewilde, Deprettere and Nouta at Technical University of Delft for their helpful email on the 5th-order elliptic filter benchmark, and Professor Barry Pangrle at Penn State University for discussions on MCM-based designs. The authors also wish to thank the anonymous reviewers for their valuable comments.

References

- [1] S. C. Chen, D. Kuck and A. H. Sameh, "Practical Parallel Band Triangular System Solvers", *ACM Transactions on Math. Software*, Vol. 4, pp. 270-277, Sept, 1978.
- [2] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms", pp. 257-276, *VLSI and modern signal processing*, S. Y. Kung, H. J. Whitehouse, T. Kailath, editors, Prentice-hall, 1985.
- [3] J. A. Fisher, "Very Long Instruction Word architectures and ELI-512", *Proc. 10th Symposium on Computer Architectures*, p. 342, Stockholm, Sweden, June, 1983.
- [4] K. Frenkel, "HDTV and the computer industry", p. 1307, *CACM*, Vol. 32, No. 11, November 1989.
- [5] D. Gajski, "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines", *IEEE Transactions on Computers*, Vol. c-30, No.3, March 1981.
- [6] G. Goossens, J. Vandewalle and H. De Man, "Loop optimizations in register-transfer scheduling for DSP systems", *Proceedings of the ACM/IEEE 26th Design Automation Conference*, 1989.
- [7] R. Hartley and A. Casavant, "Tree height minimization in pipelined architectures", *Proceedings of 1989 IEEE International Conference on CAD*, Santa Clara, California, November 1989.
- [8] C. T. Hwang, Y. C. Hsu and Y. L. Lin, "Scheduling for functional pipelining and loop winding", pp. 764-769, *Proceedings of the ACM/IEEE 28th Design Automation Conference*, San Francisco, California, June 1991.
- [9] D. A. Lobo and B. M. Pangrle, "Redundant operation creation: a scheduling optimization technique", *Proceedings of the ACM/IEEE 28th Design Automation Conference*, pp. 775-778, San Francisco, California, June 1991.
- [10] *Proceedings of IEEE Multi-Chip Module Conference*, Santa Cruz, California, 1992.
- [11] A. Nicolau and R. Potasman, "Incremental tree height reduction for high-level synthesis", *Proceedings of the ACM/IEEE 28th Design Automation Conference*, pp. 770-774, San Francisco, California, June 1991.
- [12] R. Nouta, a personal correspondence with the authors, Technical University of Delft, October, 1992.
- [13] K. Parhi and D. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding", pp. 178-195, *IEEE Trans. Computers*, Vol. 40, No. 2, February, 1991.
- [14] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis", *Proceedings of the ACM/IEEE 27th Design Automation Conference*, Orlando, Florida, June 1990.
- [15] N. K. Tsao, "Solving triangular system in parallel is accurate", pp. 633-638, *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, Edited by Gene Golub and Paul Van Dooren, NATO Series F: Computer and Systems Sciences, Vol. 70, Springer-Verlag, 1991.
- [16] H. Wang, N. Dutt and A. Nicolau, "Harmonic scheduling of linear recurrences in digital filter design", *The first European Design Automation Conference (Euro-Dac '92)*, Hamburg, Germany, September 7-10, 1992.
- [17] H. Wang and A. Nicolau, "Speedup of band linear recurrence in the presence of resource constraints", *Proceedings of 1992 the 6th International Conference on Supercomputing*, Washington, D. C., July 1992.
- [18] H. Wang, N. Dutt, A. Nicolau and K. S. Siu "High-level synthesis of scalable architectures for IIR filters using multi-chip modules", UCI ICS TR 92-104, October 1992.