SIMD PROCESSOR BASED IMPLEMENTATION OF RECURSIVE FILTERING EQUATIONS

Jaewoo Ahn, Hoseok Chang, Junho Cho, and Wonyong Sung

School of Electrical Engineering, Seoul National University
Kwanak-gu, Seoul 151-742 KOREA
Email: ahnjw72@naver.com, chs@dsp.snu.ac.kr, juno@dsp.snu.ac.kr, wysung@snu.ac.kr

ABSTRACT

Implementation of recursive equations using parallel computer architecture has long been of interest because the dependency problem makes it difficult to achieve significant speed-up. In this paper, efficient implementation of recursive filtering equations on partitioned data-path SIMD (Single Instruction Multiple Data) processors is studied. Especially, three parallel computation techniques, which are the block filtering, recursive doubling, and multi-block filtering methods, are implemented and their performances are compared using a Pentium CPU based system. The performance evaluation result of the multi-block processing method on a scalable SIMD processor is also presented.

Index Terms— parallel computation, recursive filtering, partitioned data-path, SIMD processor

1. INTRODUCTION

Recursive equations are used for many applications, such as recursive and adaptive filtering in digital signal processing. Parallel computation of recursive equations is known to be very difficult or inefficient because of the dependency problem. One example is computing multiple output samples of $y[n] = a*y[n-1] + x[n]$ at a time. The dependency problem arises because computing $y[n+1]$ needs the previous output sample, $y[n]$; this means that $y[n+1]$ and $y[n]$ cannot be computed simultaneously when using this equation. Although the history of research on parallel computation of recursive equations is quite long, the interest on this topic seems not diminished even in these days [1][2][3]. This is partly because parallel computation algorithms cannot be equally applied to all the parallel computer architectures, such as pipelined, VLIW, superscalar, multi-core, and multi-processor systems. Efficient parallel computation of recursive equations will be more important in the future as the degree of parallelism for the architecture of a CPU increases. In recent years, the partitioned data-path SIMD architecture is widely used because this one is very hardware efficient. The partitioned data-path architecture can be built easily by increasing the width of the data-path, however the flexibility is considered low because the data-path needs well aligned input operands as illustrated in Fig. 1. The number of data that is processed at a time is called the number of SIMD-ways, and is denoted as P.

In this research, we have implemented sequential and parallel computing algorithms for recursive filtering equations on SIMD processors with partitioned data-path, and compared the efficiency of them. Not only the effects of the number of SIMD-ways, but also those of the memory access patterns are examined because many previous SIMD implementations suffer from the increased data-alignment overheads. The implementation examples with constant coefficient recursive equations are given; however, this approach can be extended to time-varying coefficients recursive equations that are used for adaptive filtering.

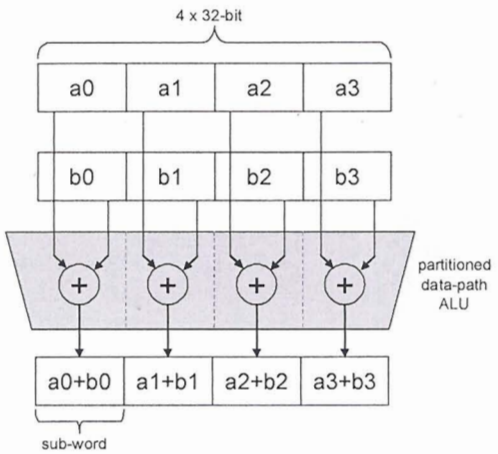

Fig. 1. Partitioned ALU structure

In Section 2, the partitioned data-path SIMD architecture is described. Section 3 shows the implementation of sequential recursive filtering equations on SIMD processor architecture. The implementation procedures employing three parallel algorithms are explained in Section 4. The experimental results are presented in Section 5. Concluding remarks are given in Section 6.

2. ARCHITECTURE OF PARTITIONED DATA-PATH SIMD PROCESSORS

The partitioned data-path SIMD processor architecture has the advantage of small hardware overhead for increasing the parallelism. The size of program memory and the instruction bandwidth are also much reduced when compared with other parallel architectures, such as VLIW, superscalar, or multi-core systems. The partitioned data-path based SIMD architecture is especially useful for multimedia applications, such as image or video processing, because they usually require short word-length, such as 8bit or 16bit, data. Nowadays, the partitioned data-path can be found in various architectures such as CPU's for personal computers and workstations, programmable digital signal processors, and embedded CPU's. Recently developed CPU's employing the VLIW or multi-core architecture mostly support SIMD arithmetic instructions.

However, the SIMD architecture has its own limitations. In order to perform a SIMD operation, the data must be aligned at first, which may require pack, unpack, or shuffle operations and can be a problem to the efficiency enhancement of the SIMD architecture.

Most computer systems employ a single bank memory or cache system due to its simplicity. Single bank memory systems can only efficiently handle aligned data access. For example, a single-bank memory system with the memory access width of 8 can access D[0]~D[7] at a time, but it needs additional cycles even for a simple unaligned data access, such as reading D[1]~D[8]. Unaligned data access not only takes extra memory cycles but also incurs data arrangement overhead in the CPU.

To reduce the data arrangement overhead, some SIMD processors equip vector memory systems that can efficiently support unaligned and stride data access [4]. Vector memory systems are usually built using multi-bank or multi-port memory based systems. In the multi-bank memory based system, shown in Fig. 2, the data is stored in an interleaving scheme. The bank number of the data is determined by '*(address) mod (P),*' and the address within a bank is calculated by '*(int)(address/P).*' Because the number of SIMD-ways, P, is usually a power of 2, the division by P can be replaced with shift operations. This structure does not have the unaligned access problem, but this can suffer from the bank conflict problem when some of the needed data are at the same bank. There are some other structures that can prevent bank conflicts, such as the prime bank memory system. Of course, the best performance can be obtained with the multi-port memory based system. The multi-port memory enables simultaneous access of P data items, and as a result there is no bank-conflict or unaligned access problem. However, a multi-port memory system usually occupies more chip area than a multi-bank structure, and its scalability is considered low.

As explained above, the partitioned data-path SIMD processor architecture can be more clearly classified by the memory architecture it employs. When a system only equips a single bank memory system, it is very needed to reduce the number of unaligned and stride accesses as much as possible.

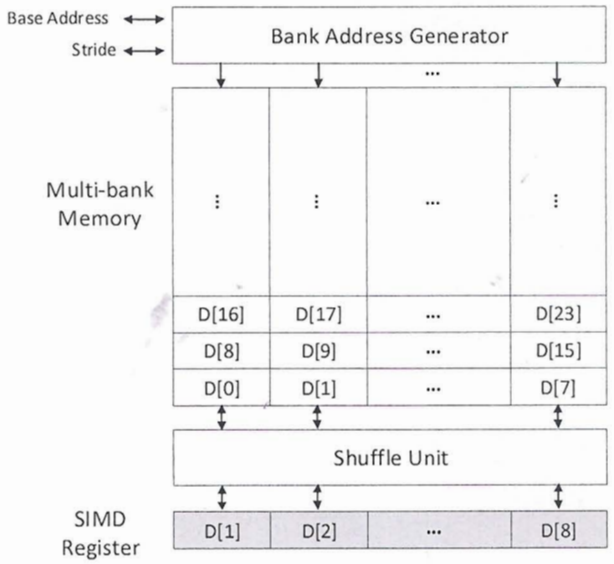

Fig. 2. Multi-bank memory system

3. SIMD PROCESSOR IMPLEMENTATION OF A SEQUENTIAL RECURSIVE EQUATION

The M-th order recursive equation considered in this work has the following form.

$$y[n]=\sum_{i=1}^{M} a_i y[n-i]+x[n] \qquad (1)$$

The equation needs M multiply-add operations for each output. One straightforward approach is computing one output sample at a time using SIMD instructions conducting inner product operations. When M is not larger than P, the needed SIMD arithmetic operations for each output data can be reduced to one assuming that the SIMD instruction set supports inner product operations. If inner product operations are not supported, implementing a vector reduction can take several cycles.

Another idea is to compute multiple output samples at a time, which is, so called, the outer product method in matrix multiplication algorithms. It seems that the feed-back term cannot be computed using the outer product method due to the dependency problem. However, the outer product method can be applied by rearranging the computation in a skewed way. Eq. (2) depicts the outer product computation flow for the M-th order recursive filtering equation. M multiply-add and one load operations are conducted concurrently.

$$
\begin{aligned}
&y[n] \mathrel{+}= a_1 * y[n\text{-}1] \\
&y[n\text{+}1] \mathrel{+}= a_2 * y[n\text{-}1] \\
&\cdots \\
&y[n\text{+}M\text{-}1] \mathrel{+}= a_M * y[n\text{-}1] \\
&y[n\text{+}M] = x[n\text{+}M]
\end{aligned} \qquad (2)
$$

When M is smaller than P, the above operations can be conducted using one SIMD instruction. The vector reduction operation is not needed. However, this method cannot be efficient in terms of the expected speedup when the order of the equation, M, is much smaller than P.

4. VECTORIZATION OF PARALLEL TRANSFORMED RECURSIVE EQUATIONS

When the order of a recursive equation, M, is not much smaller than the number of SIMD-ways, P, it would be best to employ the sequential computation methods explained above. However, if the number of SIMD-ways is larger than the order of the recursive equation, it needs a clever vectorization strategy. Vectorization includes binding of arithmetic operations and proper ordering of data. In this work, we compare the vectorization approaches which are based on three different parallel computation methods. Note that some algorithms demand much overhead for data alignment, hence comparing only the number of arithmetic steps is not sufficient. Since parallel computation of the first order recursive filtering equation is the most difficult problem, we will explain three algorithms with a first order recursive equation.

4.1. Block filtering method

The basic idea behind the block filtering method is to compute a block of output samples at a time using blocks of input data and previously computed blocks of output data [5]. The idea of block filtering method for a first order constant coefficient recursive equation is illustrated in Eq. (3), where the size of a block is assumed as 4. The initial condition for this block is $y[-1]$, and the block input data is $[x[0], x[1], x[2], x[3]]$, and the output samples are $[y[0], y[1], y[2], y[3]]$.

$$
\begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = \overset{\text{Step1}\ \text{Step2}\ \text{Step3}\ \text{Step4}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & 0 & 0 \\ a^2 & a & 1 & 0 \\ a^3 & a^2 & a & 1 \end{bmatrix}} \cdot \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \end{bmatrix} + \overset{\text{Step5}}{\begin{bmatrix} a \\ a^2 \\ a^3 \\ a^4 \end{bmatrix}} \cdot y[-1] \qquad (3)
$$

The above equation can be computed as follows:

$$
\begin{aligned}
&\text{step0: } [y[0], y[1], y[2], y[3]] = x[0]*[1, a, a^2, a^3] \\
&\text{step1: } [y[0], y[1], y[2], y[3]] \mathrel{+}= x[1]*[0, 1, a, a^2] \\
&\text{step2: } [y[0], y[1], y[2], y[3]] \mathrel{+}= x[2]*[0, 0, 1, a] \\
&\text{step3: } [y[0], y[1], y[2], y[3]] \mathrel{+}= x[3]*[0, 0, 0, 1] \\
&\text{step4: } [y[0], y[1], y[2], y[3]] \mathrel{+}= y[-1]*[a, a^2, a^3, a^4].
\end{aligned} \qquad (4)
$$

Note that the computation steps shown in Eq. (4) use very regular data structures if we keep separate copies of the coefficients vectors. There are not many overhead operations for data rearrangement even with a conventional single bank memory system. However, there seems no speed-up achieved in terms of the number of arithmetic steps. It takes five arithmetic (multiple-add) cycles for computing four outputs, which corresponds to 5/4 cycles per output sample. Even if the number of SIMD-ways, P, increases further, there is no gain in speed-up when considering the number of arithmetic steps.

4.2. Recursive doubling method

Kogge and Stone applied the recursive doubling technique to the computation of recursive equations [6]. The recursive doubling method splits the computation of a function into two equally complex sub-functions whose evaluation can be performed in parallel, and the sub-functions are split again and again to increase the degree of parallelism. Assume that a first order constant coefficient recursive equation for n = 0, ... , 6 is to be computed using input samples, $x[0]$, ..., $x[6]$, with the initial condition $y[-1]$. The computation step is illustrated in Fig. 3.

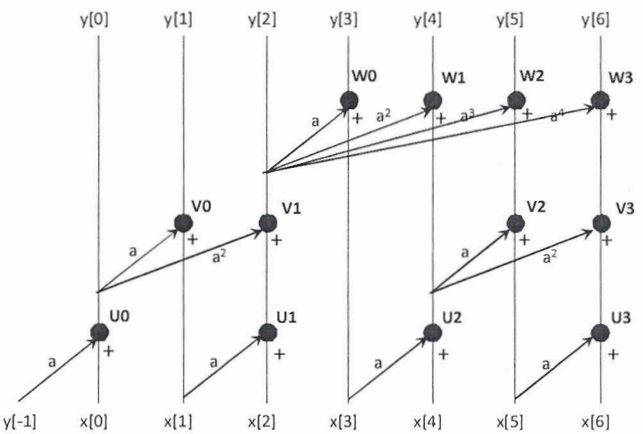

Fig. 3. Recursive doubling method for 1st order equation

The computation steps of Fig. 3 can be conducted as follows:

$$
\begin{aligned}
&\text{step0: } [U0, U1, U2, U3] = [a*y[-1]+x[0],\ a*x[1]+x[2], \\
&\qquad\qquad a*x[3]+x[4],\ a*x[5]+x[6]] \\
&\text{step1: } [V0, V1, V2, V3] = [a*U0+x[1],\ a^2*U0+U1, \\
&\qquad\qquad a*U2+x[5],\ a^2*U2+U3] \\
&\text{step2: } [y[3], y[4], y[5], y[6]] = [a*V1+x[3],\ a^2*V1+U2, \\
&\qquad\qquad a^3*V1+V2,\ a^4*V1+V3].
\end{aligned} \qquad (5)
$$

Note that $y[0]=U0$, $y[1]=V0$, $y[2]=V1$. This method demands 3 SIMD arithmetic operations for computing 7

output samples. This translates only 3/7 arithmetic operations per output sample with P of 4. Thus, this method seems much more efficient than the block filtering method.

However, we can find that the operands for arithmetic operations are not well aligned. The input operands for the step 0 are [y[-1], x[1], x[3], x[5]] and [x[0], x[2], x[4], x[6]]. The second and the third steps show more complex memory access patterns as illustrated in Fig. 3. However, the stride for memory access is smaller than P. Thus, the multi-bank vector memory unit with P banks can support these memory accesses without a bank conflict.

4.3. Multi-block filtering method

This method processes multiple blocks simultaneously so that it utilizes not only intra-block but also inter-block parallelism. Usually, the number of blocks which is processed at the same time is equal to the degree of parallelism, P. Thus, the number of data that is processed at a time is equal to P^2. Originally, Gajski developed a parallel computation algorithm for recurrence equations with a two dimensional layout of data based on a suffix problem for a semi-group which is used for high speed binary adders [7]. Sung and Mitra derived the parallel computation method in a different way, which separately computes the transient and particular solutions, and further improved the complexity of computation for constant coefficient recursive equations [8][9]. Note that the particular solution is the term that can be computed without the initial condition, while the transient solution is the term only affected by the initial condition, y[-1].

This method processes P blocks at a time, which can be placed as shown in Fig. 4. The first block contains the data of [x[0], x[1], x[2], …, x[P-1]], and the second block holds the data of [x[P], x[P+1], …., x[2P-1]].

x[P-1]	x[2P-1]	•••	x[P^2-1]
•	•		•
•	•	•••	•
•	•		•
x[1]	x[P+1]	•••	x[P^2-P+1]
x[0]	x[P]	•••	x[P^2-P]

Fig. 4. Data layout for multi-block filtering

Note that the initial conditions for these blocks are not known except for the first block. Thus, this method assumes the initial conditions of zero for all the blocks, except for the first block, and computes the particular solutions for each block in parallel. Although the initial conditions for the blocks are not known, except for the first block, each block of computation can be conducted in parallel because the particular solution does not need the initial conditions. In SIMD processors, each sub-unit of the partitioned data-path takes care of each block of computation. Thus, with a P-way SIMD processor, P blocks can be processed concurrently. The computation procedure with a P-way SIMD processor is as follows:

Initialize: $z[-1] = y[-1]$, $z[P-1]=0, \ldots., z[P^2-P-1]=0$
1st step: $z[0]=a*z[-1]+x[0]$, $z[P]=a*z[P-1]+x[P]$, …
2nd step: $z[1]= a*z[0]+x[1]$, $z[P+1]=a*z[P]+x[P+1]$, … (6)
…
P^{th} step: $z[P-1]=a*z[P-2]+x[P-1]$, $z[2P-1]=a*z[2P-2]+x[2P-1]$, …

Note that the above computation utilizes *inter-block* parallelism, which means that $z[n]$, $z[n+P]$, …, $z[n+(P-1)P]$ are computed simultaneously. After completing this computation, the output data in the first block are the complete solutions and y[P-1] is now available, but the results for other blocks are only the particular solutions. Then, the homogeneous solutions for the second block can be computed as follows:

$$\begin{bmatrix} y[P] \\ y[P+1] \\ \cdot \\ \cdot \\ \cdot \\ y[2P-1] \end{bmatrix} = \begin{bmatrix} z[P] \\ z[P+1] \\ \cdot \\ \cdot \\ \cdot \\ z[2P-1] \end{bmatrix} + \begin{bmatrix} a \\ a^2 \\ \cdot \\ \cdot \\ \cdot \\ a^P \end{bmatrix} \cdot y[P-1] \quad (7)$$

The above computation can be conducted at one step with a P-way SIMD processor, and y[2P-1] is obtained. Then, the homogeneous solutions for the third block are computed at the next cycle using y[2P-1], and yields y[3P-1]. Thus, computing the homogeneous solutions for all the blocks can be finished in just P-1 steps. This stage of computation uses the *intra-block* parallelism, which means that all the homogeneous solutions for each block are evaluated at a time. As a result, the number of arithmetic steps for one sample can be modeled as $(2P-1)/P^2$ for this method. Thus, the speed-up with this method is approximately $P/2$, and is much better than the previously explained methods when P is large.

This method is very efficient in terms of the number of arithmetic operations, and can achieve a linear speed-up with the number of processors, P. For computing 16 samples with a 4x4 data layout, it consumes only 7/16 arithmetic steps per sample. Although the number of arithmetic steps seems comparable with that of the recursive doubling, the multi-block processing method is much more efficient as P increases. However, the first stage of computation needs to pack the input data which are P samples away. For example, input data x[0], x[P], x[2P], … x[(P-1)P] are used together as illustrated in Eq. (6). In other words, the memory access pattern has the stride of P. Even with a P-way vector memory shown in Fig. 2, this pattern of

data access incurs the bank-conflict problem. However, the multi-port memory based vector memory or the prime-bank vector memory can support this type of memory access without the bank-conflict problem. The second stage of computation that adds the homogeneous solutions can be conducted with a simple one-bank memory structure because this uses sequentially ordered data.

5. EXPERIMENTAL RESULTS

We conducted experiments with two architectures. One is the performance comparison of parallel computation methods for a first order recursive equation using the Intel Pentium-SSE3 SIMD instruction set. The other is the performance estimation of the multi-block filtering algorithm for an M-th order recursive filter using a hypothetically modeled scalable SIMD computer, where two different memory systems are equipped to know their performance effects.

5.1. Pentium-SSE3 based implementations

We used an Intel Pentium Dual-Core CPU based system with 2.0 GHz clock frequency for this experiment. The Intel C++ compiler (ver.10.1) that supports the intrinsic functions for the SSE3 instructions is used for this experiment. Data arithmetic is conducted in single precision floating-point arithmetic. The first order recursive filtering programs that are based on the sequential form, sequential form with 4-times loop unrolling, block filtering, recursive doubling, and multi-block filtering algorithms are developed. Except for the sequential form, SIMD instructions are used as much as possible. The dual-core feature of the target CPU is not exploited in this experiment to isolate the effect of the SIMD data-path for the recursive filtering algorithm implementations. The multi-block filtering method uses the four by four matrix transpose macro included in the Intel C++ compiler to rearrange data.

The number of execution cycles and instruction profiling results are shown in Table 1. To measure the execution cycles, $2x10^6$ samples are processed 256 times. The cycles show the number of CPU core clock sampling events occurred while the program runs in that procedure. The event occurs 1,000 times in a second. In the profiling result, the numbers inside parentheses show the numbers of instructions per output sample.

The execution-cycle results show that the block filtering algorithm uses the largest amount SIMD arithmetic instructions (2.3 instructions per sample), while the recursive doubling and the multi-block filtering methods need quite small SIMD arithmetic instructions. However, the recursive doubling method requires the largest number of cycles for SIMD data rearrangement, which results in the poorest performance among the parallel processing algorithms. This confirms that reducing the data arrangement overhead is very important in SIMD processor based implementations. Overall, the speed-up performance of the parallel processing algorithms is not very impressive in two reasons. One reason is that the number of SIMD-ways for the Pentium SSE3 is only four, thus the maximum speed-up due to SIMD processing is bounded by four. The other reason is that the Pentium CPU does not equip a vector memory unit, and as a result large data rearrangement overhead cycles hamper the performance increase.

Table 1. Comparison of developed computation methods

	Sequential	Loop unroll seq.	Block filter	Recur. doubl.	Multi-block filt.
Cycles	3,820	2,671	1,982	2,290	1,960
Speedup	1.00	1.43	1.93	1.67	1.95
CPI	2.37	2.08	1.493	1.856	2.11
# total inst.	6(6)	19(4.8)	20(5)	32(4.6)	56(3.5)
# SIMD inst.	0	0	19(4.8)	29(4.1)	53(3.3)
# SIMD arith.	0	0	9(2.3)	6(0.86)	14(0.88)
# SIMD rearr.	0	0	5(1.3)	14(2.0)	19(1.2)
# SIMD others	0	0	5(1.3)	9(1.3)	20(1.3)
# of samples per iteration	1	4	4	7	16

5.2. Scalable SIMD processor based implementations

In order to evaluate the performance with the increasing number of SIMD-ways and also know the effects of a vector memory unit, the performance is measured using a model of a scalable SIMD processor. The SIMD processor for the performance evaluation is based on ARMv4 architecture and includes a Px16bit SIMD ALU and 16 vector registers. The detailed architecture and the corresponding SIMD instruction set are explained in [4]. All SIMD instructions except for SIMD memory instructions take one cycle delay. The architecture equips a multi-port vector memory unit. The multi-port vector memory unit allows to access vector data without overhead for alignment and reorganization, while the single-ported conventional memory unit consumes overhead cycles for non-aligned or complex vector access operations.

The program for an M-th order recursive filtering is shown in the Appendix. The performance is evaluated for a 2^{nd} order recursive digital filter. The program was generated using a vectorizing C compiler.

The performance for the implementation of a second order digital filter is shown in Table. 2. The numbers of clock cycles are shown, and the speed-up results are also given inside of the parentheses. The performances with and without a vector memory unit are compared. The results show that it is possible to obtain a quite good speed-up with a vector memory unit, but the speed-up without a vector memory unit saturates quite rapidly as the number of SIMD-ways increases.

Table 2. Number of cycles with scalable SIMD processor

	P=1	P=4	P=8	P=16	P=32
Sequential alg.	7,167 (1)				
Multi-block filtering without vector memory		6,147 (1.17)	4,355 (1.65)	2,947 (2.43)	2,243 (3.20)
Multi-block filtering with vector memory		4,866 (1.47)	2,434 (2.94)	1,218 (5.88)	610 (11.75)

6. CONCLUDING REMARKS

We have compared three parallel computation methods for recursive filtering equations using the SIMD instruction set on a Pentium CPU. The block filtering method needs a large number of arithmetic operations, but the data structure is quite regular. The recursive doubling method reduces the number of arithmetic operations by reusing intermediate results, but this method demands rather complex addressing of operands. The multi-block filtering method, which separately computes the particular and transient solutions, requires the least number of arithmetic operations when the number of SIMD-ways increases, however this needs data transpose operations that are very inefficient in a SIMD processor without a vector memory unit. This study shows that not only the number of arithmetic steps but also data access patterns affect the speed-up performance of parallel recursive equations on a SIMD processor very much.

7. REFERENCES

[1] J Robelly, G Cichon, H Seidel, G Fettweis, "Implementation of recursive digital filters into vector SIMD DSP architectures," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2004.

[2] M. Horst, K. Berkel, J. Lukkien, R. Mak, "Recursive filtering on a vector DSP with linear speedup," *16th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2005.

[3] R. Kutil, "Parallelization of IIR filters using SIMD extension," *International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2008.

[4] H. Chang, J. Cho, and W. Sung, "Compiler-based performance evaluation of an SIMD processor with a multi-bank memory unit," *Journal of Signal Processing Systems*, Vol. 56, No. 2-3, Sep, 2009.

[5] C. S. Burrs, "Block implementation of digital filters," *IEEE Tr. Circuit Theory*, vol. 18, no. 6, pp. 697-701, Nov. 1971.

[6] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Tr. Computers*, vol. C-22, pp. 786-792, Aug. 1973.

[7] D. D. Gajski, "An algorithm for solving linear recurrence systems on parallel and pipelined machines," *IEEE Tr. Computers*, vol. C-30, pp. 190-206, March 1981.

[8] W. Sung and S. K. Mitra, "Efficient multi-processor implementation of recursive digital filters," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 1986.

[9] W. Sung and S. K. Mitra, "Implementation of digital filtering algorithms using pipelined vector processors," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1293-1302, Sep. 1987.

[10] *Intel C++ Compiler Professional Edition for Windows ver. 10.1* http://software.intel.com/en-us/intel-compilers/

8. APPENDIX

```
static int z[12+INPUT_SIZE];
static int f[INPUT_SIZE];
static int y[12+INPUT_SIZE];
int        i, j, k, l;


for (l=0; l<INPUT_SIZE; l+=VEC_LEN*VEC_LEN) {
        for (j=0; j<VEC_LEN; j++) {
                for (k=1; k<NTAPS+1; k++) {
                        for (i=0; i<VEC_LEN; i++) {
        y[NTAPS+VEC_LEN*VEC_LEN*l+VEC_LEN*i+j]
+= a[k]*y[NTAPS+VEC_LEN*VEC_LEN*l+VEC_LEN*i+j-k];
                                        }}}}
// particular solution, loop-carried dependency resolved, vectorized

for (l=0; l<INPUT_SIZE; l+=VEC_LEN*VEC_LEN) {
        for (i=0; i<VEC_LEN; i++) {
                for (k=0; k<NTAPS; k++) {
                        for (j=0; j<VEC_LEN; j++) {
                y[VEC_LEN*VEC_LEN*l+VEC_LEN*i+j]
+= T[j][k]*y[VEC_LEN*VEC_LEN*l+VEC_LEN*i+j-(k+1)];
                                        }}}}
// transient solution, vectorized
```