Computationally efficient implementation of sarse-tap FIR adaptive filters with tap-position control on intel IA-32 processors

Computationally Efficient Implementation of Sparse-Tap FIR Adaptive Filters with Tap-Position Control on Intel IA-32 Processors

Akihiro Hirano and Kenji Nakayama
Graduate School of Natural Science and Technology,
Kanazawa University,
Kanazawa, 920–1192, Japan
Email: {hirano,nakayama}@t.kanazawa-u.ac.jp

***Abstract*—This paper presents an computationally efficient implementation of sparse-tap FIR adaptive filters with tap-position control on Intel IA-32 processors with single-instruction multiple-data (SIMD) capability. In order to overcome random-order memory access which prevents a vectorization, a block-based processing and a re-ordering buffer are introduced. A dynamic register allocation and the use of memory-to-register operations help the maximization of the loop-unrolling level. Up to 66percent speedup is achieved.**

I. Introduction

Recent years, personal computer (PC) based communication systems such as Skype and Messenger becomes very popular. PC-based systems are useful not only for personal communications, but also for business systems such as teleconferencing. Hands-free voice communication over the Internet shown by Fig. 1 without acoustic echo cancellers (AEC's) causes an echo path with very long flat-delay, which might be more than 4000 taps. Fig. 2 depicts an impulse response of such an echo path. There are multiple flat-delay sections and dispersive regions.

To reduce a large number of computations for a long-tap adaptive filter which can cope with an unknown number of multiple echoes, sparse-tap adaptive FIR filters with tap-position control have been proposed[1], [2]. These algorithms distribute a small number of filter coefficients to significant regions of the echo-path impulse response. For tap-position control, scrub taps waiting in a queue (STWQ) algorithm[1] and grouping algorithm[2] have been proposed. Though an implementation of such filters on digital signal processors (DSP's) has been reported, implementations on PC-based system have not been reported.

Modern processors for PC's have powerful instruction set for multimedia processing. Intel IA-32 architectures[3] have MMX (Multi Media eXtention) and also SSE (Streaming Single instruction multiple data Extension, Streaming SIMD Extension). Four-way vector operations are supported for 32-bit floating-point (FP) data.

In this paper, an efficient implementation of sparse-tap adaptive FIR filters on Intel IA-32 processors is discussed. Section II describes sparse-tap adaptive FIR filters. IA-32 processor is briefly described in Sec. III, followed by some implementation issues. The proposed implementation is shown by Sec. V. Section VI compares the performance.

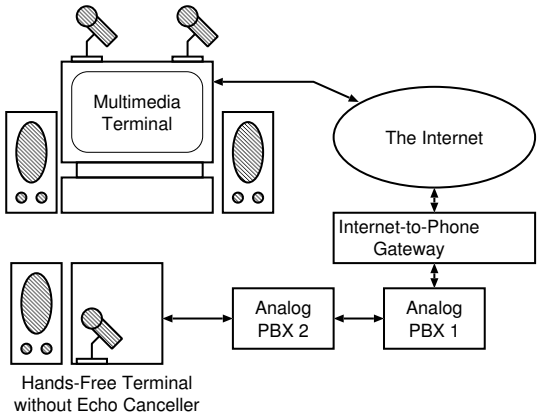

Fig. 1. Teleconferencing over Internet

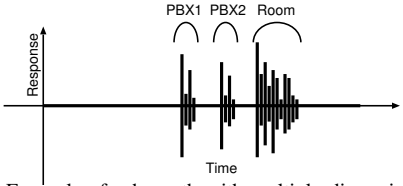

Fig. 2. Example of echo path with multiple dispersive regions

II. Sparse-Tap Adaptive FIR Filters

Sparse-tap adaptive FIR filters shown by Fig. 3 have N_A active taps with filter coefficients while we have N_T total taps. The other taps (inactive taps) do not have coefficients. When $N_A << N_T$, the computational costs can be reduced. This filter calculates the echo replica $y(n)$ at time index n from the input signals $x(n)$ and the filter coefficients $w_k(n)$ for the k-th active tap by

$$y(n) = \sum_{k=0}^{N_A-1} w_k(n)x(n-i_k). \quad (1)$$

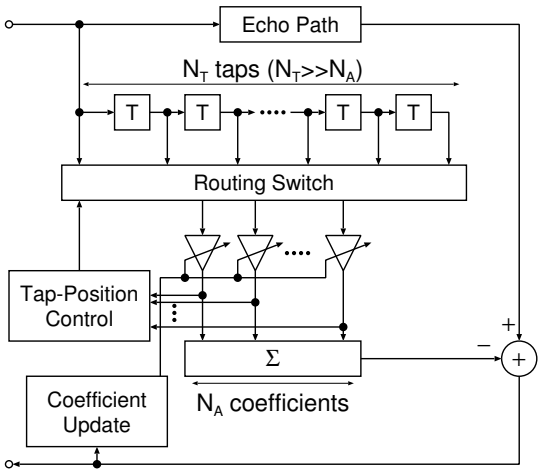

Fig. 3. Echo cancellation by sparse-tap adaptive FIR filter

The active-tap index i_k designates the location of the k-th active tap. The filter coefficient $w_k(n)$ is updated by a Normalized Least-Mean-Square (NLMS) algorithm [4] by

$$w_k(n+1) = w_k(n) + \frac{\mu e(n) x(n-i_k)}{\sum_{k=0}^{N_A-1} x^2(n-i_k)}. \qquad (2)$$

μ is a step-size parameter which controls the convergence. $e(n)$ is an error signal.

The tap-position control algorithms[1], [2] select N_A active taps from N_T taps. The active taps are selected based on the coefficient values; after T_U iterations of coefficient updates, active taps with the N_R smallest coefficient in magnitude are made inactive. Then, N_R new active taps are selected.

STWQ algorithm[1] stores inactive tap indices in a queue. N_R indices are taken from the queue and inactive taps with these indices are activated.

In the grouping algorithm[2], N_T taps are divided into N_G equisized subgroups. N_G queues corresponding to the subgroups contain the inactive tap indices. For tap assignment, one subgroup is selected at a time. New active tap indices are taken from the queue for the selected subgroup. The selected subgroup hops from one subgroup to another.

The grouping algorithm achieves fast convergence by controlling the staying time $T_S(i)$ and the selection order for the i-th subgroup. Initially, selection order and staying time $T_S(i)$ are set to the order of subgroup number i and a constant T_{S0}, respectively. If all subgroups have been selected once, $T_S(i)$ and the order are re-determined based on the sum of the absolute coefficient values in each subgroup. The order of subgroups is determined in the order of the sums. $T_S(i)$ is determined in proportion to the sum.

III. Intel IA-32 Processors[3]

In this implementation, Intel Core microarchitecture, e.g. Core2 Duo, is assumed. Main features are listed below.

- Five execution pipelines, up to fourteen stages
  - ALU, FP/MMX/SSE Move, Branch
  - ALU, FP/MMX/SSE Add
  - ALU, FP/MMX/SSE Multiply
  - Load
  - Store
- Executes up to five instructions per cycle
  - Up to three ALU operations per cycle
  - Up to three SSE operations per cycle
- Branch prediction
- 32kB instruction + 32kB data L1 cache
- 2MB, 4MB or 6MB L2 cache
- Hardware prefetchers, which predict data access sequence and automatically load data from external memory into cache
- Eight general-purpose integer registers
- Eight FP registers
- Eight SSE registers

The MMX and SSE instructions[5], [6] provide some vector operations. For 32-bit floating-point data, simultaneous calculations on four independent data sets can be carried out. Therefore, up to four-times speed-up might be possible if data bandwidth allowed.

IV. Considerations on Implementation

There are many considerations on implementation using general-purpose processors with SIMD capability. Examples are listed below.

- Efficient Vectorization
- Data alignment for SIMD load/store operations
- Implementation of tapped delay lines
- Memory hierarchy, especially slow external memory
- Long latency for memory load: Core2 processor requires additional six cycles even for L1 cache
- Few data registers: Only eight for IA-32
- Memory-to-register operations: Source operand for computation can be memory

The vectorization and the data alignment are common to implementation on DSP's, while the others might be specific for general-purpose processors. Most DSP's are equipped with the address generators for the tapped delay lines, multiple data memories with no-wait access. Few data registers, memory-to-register operations are specific for IA-32 processors.

In the four-way vectorization for FIR filtering, simultaneous calculation for k-th tap through $(k+3)$-th tap are common way. However, such approach cannot be applied directly for sparse-tap filters. This is because load operations of the input signals through the routing switch prevent the vectorization. Generally, the order of the active-tap positions is in random. Therefore, simultaneous load of the input signals commonly used in vector operations is not possible.

Fig. 4 shows a two-way SIMD case. Two input signals $x(n-i_0)$ and $x(n-i_1)$ are located at two distant addresses. Though multiple scalar-load operations followed by a vector calculation is possible, this approach often degrades the performance[7]. The misalignment problem[8] on the tapped delay lines also occurs.

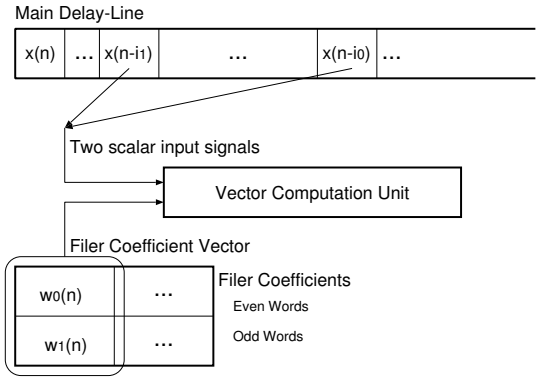

Fig. 4. Problems on Vectorization

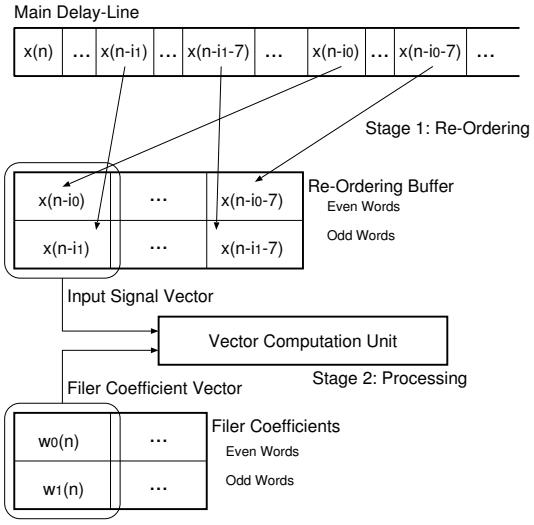

Fig. 5. Vectorization Strategy

Such data access through the routing switch causes other problems. In order to load the input signal $x(n-i_k)$ for k-the active tap, two load operations, one for the active-tap index i_k and one for the input signal $x(n-i_k)$, are necessary. Thus the number of load operations and also the number for address calculations increase. For random-order memory accesses, the efficiency of the data cache might degrade.

For long pipeline latency, the loop unrolling[9] is widely used. This technique requires a large number of data registers; usually n registers are needed to overcome n-cycle latency. Since the minimum latency for a load operation is seven instruction cycles, at least eight-way loop unrolling would be feasible to hide the latency. Eight registers might not be sufficient for some applications.

V. Implementaion of Sparse-Tap Adaptive FIR Filters

This implementation focuses on how to vectorize sparse-tap adaptive FIR filters and also on how to optimize the program with only a few data registers. The vectorization utilizes a property of PC-based communication systems. The optimization exploits IA-32 specific operations.

A. Vectorization Strategy

In PC-based comunication systems, a signal processing would be carried out on a block of multiple input samples rather than a sample-by-sample manner. This is because the input/output (I/O) buffers and packet buffers cause a block-based processing. Such a block-based processing can be utilized to vectorization; multiple vectors of the input signals for an active-tap can be prepared at a time.

A two-stage processing, which consists of a re-ordering stage and a processing stage, is introduced in this implementation. Fig. 5 demonstrates the strategy. In the re-ordering stage, the input signals for the active taps are copied into a re-ordering buffer. When re-ordering buffer size is N_B, N_B samples for k-th active tap, $x(n-i_k)$ through $x(n-i_k-N_B+1)$, will be successively copied into the buffer. The processing stage can utilize vector operations because the input signals have been located in the order of the processing.

A block-based processing might make such a two-stage strategy computationally efficient even when the overhead for the re-ordering stage is introduced. The first reason is, of course, four-way SIMD processing which might increase the processing speed by up to four times. The second reason is a cache efficiency. For the successive access $x(n-i_k)$ through $x(n-i_k-N_B+1)$ will make the hardware cache prefetch easy to operate.

The third reason is the reduction of the memory accesses. For example, the load operations for the active-tap index i_k can be reduced. Only once per N_B samples is necessary. On the other hand, twice per one sample is necessary for a sample-by-sample operation.

B. Optimization

In order to maximize the level of the loop unrolling, the memory-to-register operations are used if possible. For the coefficient updates by the NLMS algorithm, all multiply operations and all add operations are the memory-to-register operations. For the FIR filtering and power calculation, which are performed in a same loop, the half of the multiply operations are the memory-to-register operations. This saves the data registers. Since the compiler tend to generate a load-store style code, the optimization has been carried out by using an in-line assembler.

Fig. 6 demonstrates the dynamic register allocation for the partial filter output $y_k(n)$ defined by

$$y_k(n) = \sum_{l=0}^{k} w_l(n)x(n-i_l) \tag{3}$$

and the partial input power $p_k(n)$ defined by

$$p_k(n) = \sum_{l=0}^{k} x^2(n-i_l). \tag{4}$$

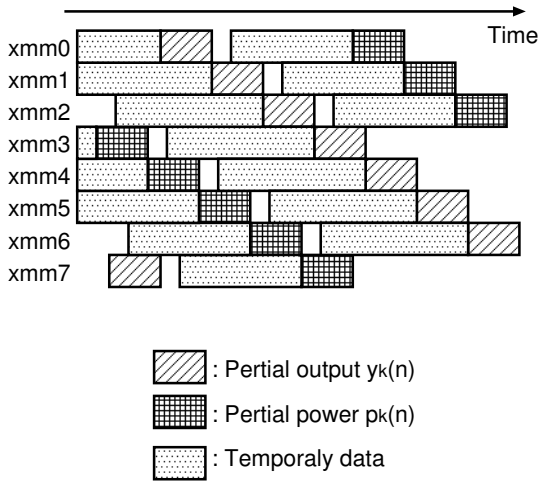

Fig. 6. Register Map for FIR and Power

TABLE I
SPECIFICATIONS OF ADAPTIVE FILTER

Sampling frequency	16kHz
Block size	320
Number of taps N_T	16000
Number of active taps N_A	800
Adaptation	NLMS
Precision	32-bit floating point

These partial results work through eight data registers xmm0 through xmm7, not staying at a single register. This also saves the data registers. These register-saving techniques makes eight-way loop unrolling by using only eight data registers possible.

VI. PERFORMANCE COMPARISONS

The grouping algorithm has been implemented and tested. The specifications of the implemented sparse-tap adaptive filter is shown in Tab. I. Table II depicts the specifications of the platform. The critical part of the adaptive filter has been programmed in an assembly language. Instructions have been scheduled for highest speed on Core2 processors. For performance comparison, a program in C language with SSE scalar operations is used.

Table III compares the performance. The calculation times for 1200seconds of input data have been measured. By the re-ordering and the vectorization, the computation speed is increased by almost 66percent. The vectorized version is almost five times faster for the FIR filtering and power calculation, while that is almost three times faster for the NLMS algorithm.

Though the improvements depend on the re-ordering buffer size, 40percent speedup can be achieved for a wide range of the block size; from 10 to 160. If only the re-ordering is introduced, the performance is degraded because the overhead is larger than the improvements.

TABLE II
SPECIFICATIONS OF PLATFORM

	Core2 Duo
Type	E8200
Core Clock	2.66GHz
FSB Clock	1333MHz
L1 Data Cache	32kB
L1 Inst. Cache	32kB
L2 Cache	6MB
Chipset	Intel G33

TABLE III
COMPARISON OF CPU TIME

Buffer Size	Vector	Total	Ratio	FIR+ Power	LMS	Re-ordering
None	No	79.48	1.00	33.17	28.44	–
320	No	139.49	1.76	31.33	17.64	68.54
80	No	85.14	1.07	31.33	17.43	17.30
320	Yes	103.38	1.30	6.42	6.77	68.45
160	Yes	50.09	0.63	6.38	6.73	24.40
80	Yes	48.84	0.61	6.33	6.48	17.39
40	Yes	50.19	0.63	6.43	6.45	24.14
10	Yes	57.82	0.72	6.47	6.23	26.15
4	Yes	76.19	0.96	5.88	5.97	44.29
1	Yes	157.78	1.99	6.35	6.05	120.26

VII. CONCLUSIONS

A computationally efficient implementation of sparse-tap FIR adaptive filters has been proposed. To vectorize, the block-based processing and the re-ordering operations are introduced. The introduction of the dynamic register allocation and the use of memory-to-register operations maximize the loop-unrolling level. Almost 66percent of the execution speed is achieved.

REFERENCES

[1] S. Kawamura and M. Hatori, "A tap selection algorithm for adaptive filters," *Proc. of ICASSP '86*, pp. 2979–2982, 1986.
[2] S. Ikeda and A. Sugiyama, "A fast convergence algorithm for sparse-tap adaptive fir filters for an unknown number of multiple echoes," *Proc. of ICASSP '94*, pp. 41–44, 1994.
[3] "Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," May 2007.
[4] J. Nagumo and A. Noda, "A learning method for system identification," *IEEE Trans. AC*, vol. 12, no. 3, pp. 282–287, Mar 1967.
[5] "Intel 64 and IA-32 architectures software developer's manual volume 2a: Instruction set reference, a-m," May 2007.
[6] "Intel 64 and IA-32 architectures software developer's manual volume 2b: Instruction set reference, n-z," May 2007.
[7] "ADSP-2136x SHARC processor hardware reference," May 2006.
[8] B. Juurlink A. Shahbahrami and S. Vassiliadis, "Performance impact of misaligned accesses in SIMD extensions," *Proc. of ProRISC 2006*, pp. 334–342, 2006.
[9] David A. Patterson and John L. Hennessy, "Computer organization and design," .