International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:1, 2008

# Optimization of SAD Algorithm on VLIW DSP

Hui-Jae You, Sun-Tae Chung, and Souhwan Jung

***Abstract*—**SAD (Sum of Absolute Difference) algorithm is heavily used in motion estimation which is computationally highly demanding process in motion picture encoding. To enhance the performance of motion picture encoding on a VLIW processor, an efficient implementation of SAD algorithm on the VLIW processor is essential. SAD algorithm is programmed as a nested loop with a conditional branch. In VLIW processors, loop is usually optimized by software pipelining, but researches on optimal scheduling of software pipelining for nested loops, especially nested loops with conditional branches are rare. In this paper, we propose an optimal scheduling and implementation of SAD algorithm with conditional branch on a VLIW DSP processor. The proposed optimal scheduling first transforms the nested loop with conditional branch into a single loop with conditional branch with consideration of full utilization of ILP capability of the VLIW processor and realization of earlier escape from the loop. Next, the proposed optimal scheduling applies a modulo scheduling technique developed for single loop. Based on this optimal scheduling strategy, optimal implementation of SAD algorithm on TMS320C67x, a VLIW DSP is presented. Through experiments on TMS320C6713 DSK, it is shown that H.263 encoder with the proposed SAD implementation performs better than other H.263 encoder with other SAD implementations, and that the code size of the optimal SAD implementation is small enough to be appropriate for embedded environments.

***Keywords***—Optimal implementation, SAD algorithm, VLIW, TMS320C6713.

## I. Introduction

RECENTLY, multimedia applications which necessitate mul timedia stream processing (motion picture compression /display, etc.) in embedded environments such as smart phone, digital camera, digital camcorder, embedded web camera server, PDA, and etc. have been increasing [1,2]. Embedded processors need more computing power as demand of multimedia stream processing ability is increasing. Currently, adoption of VLIW (Very Long Instruction Word) architecture becomes the trend of high-performance DSP processor [1,3,4] since VLIW supports instruction level parallelism by low-cost compiler compared to the dynamically hardware-scheduled superscalar processors. TMS320C6x, which is a high-performance DSP ASSP popularly adopted for many multimedia applications, is also based on VLIW architecture.

Porting multimedia applications developed on workstations into a VLIW embedded processor simply may not achieve full capability since VLIW architecture is not fully utilized.

Most of multimedia streaming applications need motion picture encoding and decoding. Motion estimation is one of the most important processes in motion picture coding since achievement of high compression ratio depends on how well the motion estimation is accomplished. However, motion estimation is computationally highly demanding process, and SAD (Sum of Absolute Difference) algorithm is the most heavily computed part in motion estimation process. Thus, an efficient implementation of SAD algorithm is very important. SAD algorithm is usually implemented in a nested loop with conditional branch. The conditional branch in SAD algorithm is for exit from the nested loop when the computed SAD value exceeds the minimum SAD value among previously computed SAD values. This conditional branch saves the computational time in finding the best matching macroblock. Loop on VLIW processors is implemented optimally by utilizing software pipelining technique. Optimal scheduling of software pipelining of single loops has been investigated a lot in the past [4]. However, researches on optimal scheduling of software pipelining of nested loops, especially nested loop with branch are rare [5, 6, 7].

In this paper, we propose an optimal scheduling and implementation of SAD algorithm with a conditional branch on a VLIW DSP processor. The proposed optimal scheduling first transforms the nested loop with conditional branch into a single loop with conditional branch with consideration of utilizing ILP capability of the VLIW processor fully and realizing earlier escape from the loop. Next, the proposed optimal scheduling applies a modulo scheduling technique developed for single loop.

Based on this optimal scheduling strategy, optimal implementation of SAD algorithm on TMS320C67x, a VLIW DSP is presented. Through experiments on TMS320C6713 DSK, it is shown that H.263 encoder with the proposed SAD implementation performs better than other H.263 encoder with other SAD implementations, and that the code size of the optimal implementation of SAD algorithm is small enough to be appropriate for embedded environments.

In the past, sizable research works on the implementation of block matching have been done, but most of them have dealt with efficient implementation of the algorithm itself irrespective of processor architecture [8, 9]. Also, research works on optimal implementation and performance of H.263/MPEG 4 encoder have been reported [2,10,11,12,13], but most of them deal with overall S/W design and performance analysis on SIMD processors and VLIW processors, but rather than optimization of specific algorithms on VLIW architecture [14]. Texas Instrument DSP Image Library [15] provides an assembly implementation of SAD algorithm. This implementation deals with SAD without branch, thus the

Hui-Jae Yu, Sun-Tae Chung and Souhwan Jung are with School of Electronic Engineering, Soongsil University, Seoul, Korea (e-mail: cst@ssu.ac.kr).

performance of the motion picture encoding using TI SAD algorithm is worse than the motion picture encoding using SAD algorithm with conditional branch to exit from the nested loop. For performance analysis of SAD implementations, we use UBC H.263 encoder [16] where the SAD algorithm in motion estimation module will be replaced by various SAD implementations including the proposed one for each experiment. We employ TMS320C6713 for a VLIW DSP. Through experiments where 1 intramode and 2 intermode encoding are performed using each SAD implementation for a standard video sequence, foreman.qcif, it is shown that H.263 encoder using the proposed SAD implementation is much faster than H.263 encoder adopting any other implementations of SAD.

The rest of the paper is organized as follows. Section 2 introduces background, and Section 3 presents our proposed optimal scheduling of SAD algorithm, and optimal implementation of SAD algorithm on TMS320C6713, a VLIW DSP. Experiment results are discussed in Section 4, and finally the conclusion is presented in Section 5.

## II. Background

### A. Motion Picture Codecs, Motion Estimation, and SAD

In this section, we briefly introduce motion estimation, block matching and SAD algorithm. For more details on motion picture codecs, H.261, H.263, H.264, and MPEG, please refer to [17]. The fundamental difference between motion picture encoding and still picture encoding is that motion picture encoding enhances compression ratio by removing the temporal visual information redundancy and encoding the only residual visual information.

The redundant visual information is captured by motion estimation which seeks the best matched macroblock in the search region of the reference image frame to the current macroblock in the current image frame. The best matched macroblock in the reference frame to the current macroblock is the macroblock in the search area in the reference frame which has the minimum SAD value with the current macroblock.

Since several hundred repeated calculations of SAD algorithm is necessary in order to find the best matched macroblock in the search region of the reference frame for each current macroblock and there are numerous macroblocks in a frame (eg. 99 macroblocks for 1 QCIF size frame), it is important to seek an efficient implementation of SAD algorithm. Encoding utilizing motion estimation is called intermode encoding, and the other case of encoding is called intramode ending. Still image encoding like JPEG does intramode encoding only.

### B. VLIW (Very Long Instruction Word) Processors [3, 4]

ILP (Instruction Level Parallelism) means capability of processing several independent instructions in parallel. Superscalar processors and VLIW processors are representative ILP processors. In VLIW processors, compiler checks the dependency among instructions, schedules instructions, and constructs instruction word packet (Very Long Instruction Word) consisting of instructions which can be executed in parallel. In running, instruction word packet passes through pipeline stages such as instruction fetch and instruction interpretation, and then instructions in the instruction word packet are separately issued into several function blocks and are executed in parallel.

Since determining the order of execution of instructions and parallel execution of instructions is handled by the compiler, the processor does not need the scheduling hardware that the superscalar processors require. Thus, VLIW processors have less complex hardware architecture and more suitable for embedded processor than superscalar processors.

### C. Software Scheduling, Loop Unrolling and Software Pipelining

The purpose of software scheduling is to rearrange and group instructions under constraints so that as many instructions as possible are executable in parallel [4, 18, 19]. Scheduling constraints are resource constraints (finite number of registers and function units), data dependency among instructions, function unit latency, and branch instruction, and recurrence constraint. A loop is called to have recurrence constraint if an operation in one iteration of the loop has direct or indirect dependence upon the same operation from a previous iteration. Loop unrolling and software pipelining are representative scheduling techniques for optimal scheduling of loop in codes [4, 18, 19]. Loop unrolling increases loop body by unwinding many iterations of loop into a new iteration. Bigger loop body will have higher chance of more efficient scheduling. Moreover, loop unrolling reduces the branch instructions and overhead instructions due to loop iterations. The major side effects of loop unrolling are: a) the increased register usage in a single iteration to store temporary variables, which may hurt performance; and b) the code size expansion after the unrolling, which is undesirable for embedded applications.

Software pipelining is a technique used to schedule instructions from a loop so that multiple instructions of different loop iterations can execute in parallel.

### D. Software Pipelining Scheduling

Software pipelined loop consists of 3 code areas: prolog, kernel, and epilog. The kernel of the software pipelined loop is the repeating code area, the prolog is the code area above the kernel (to enter the kernel), and the epilog is the area below the kernel (to exit from the kernel). The constant interval between the start of successive iterations in the kernel is termed the initiation interval (usually denoted as II). As II is smaller, the execution time of the loop becomes shorter since the ILP of the software pipeline becomes higher. Therefore, the purpose of the scheduling of software pipelining is to make II shorter as much as possible under constraints and to achieve the maximum performance. The minimum initiation interval is denoted as MII, and seeking MII under constraints is well known to be NP-complete [18]. Thus, heuristic approach is needed in finding MII as described in the below.

1) Modulo Scheduling [4, 19]

Modulo scheduling is to schedule the instructions of the loop in a manner that all iterations have identical schedules (single II) except that each iteration is scheduled to start some fixed number of cycles later than the previous iteration. Modulo scheduling is the most popular one among software pipeline scheduling techniques. Basically, the modulo scheduling is proceeded as follows.

1. Initially, set II as II = max(ResMII, RecMII)
2. Search for a modulo schedule satisfying II by using DDG (data dependence graph) and MRT (Modulo Resource Reservation Table).
3. If one find a schedule satisfying 2, then stop. If not, set II=II+1 and go back to step 2.

ResMII means the minimum value by resource constraints and RecMII means the minimum value by recurrence constraints. For practical determination of ResMII and RecMII, please refer to [19]. DDG is a graph showing dependency among instructions and MRT is a table which records and checks resource utilization at a time slot {C mod II} and at a cycle C.

### *E. TMS320C6713*

1) Architecture of TMS320C67x

The TMS320C67x is a family of 32-bit TMS320C6x DSP with floating-point operation capability added. The TMS320C67x supports execution of 8 instructions in parallel. It contains two floating-point data paths. Each data path contains two ALUs (S and L units), a multiplier (M unit), and adder/subtractor for address generation (D unit), and 16 registers ( A0 ~ A15 or B0 ~ B16). For more details about TMS320C6713, please refer to [20].

2) Instruction Set of TMS320C6x [21]

Instruction set of TMS320C6x is RISC-like and supports load-store, and memory reference is only allowed in load and store instructions. The pipeline latency of an instruction is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading. Multiplication of 16x16 has 1 delay slot, branch instruction has 5 delay slots, load instruction has 4 delay slots, but store and single cycle instructions (add, subtract, etc.) has 0 delay. The following shows an example of TMS320C6x assembly instructions.

```
         LDBU   .D1    *A4++,     A7 ;
|| [A1]  ADD    .S2    B13,       A8,    B12 ;
```

As seen in the above example, TMS320C6x assembly programming is complex since it needs to designate parallel processing of instructions using ||, conditional execution of instruction ([A1]), and function units to be used (.D1, .S2), and register usage (source registers and destination register) of instructions, and also needs to consider pipeline latency of instructions. Texas Instruments provides linear assembly which does not need to designate parallel execution, function units, register usage, and pipeline latency. If a user writes linear assembly, then the linear assembly optimizer provided by TI translates the linear assembly into proper assembly instructions.

3) Resource Constraints of Instructions and Memory Bank Conflicts in TMS320C6x

For resource constraints in TMS320C6x, please refer to [21]. In implementation, one should consider memory bank conflicts in TMS320C6x [21], and most C6x processors use interleaved memory banks, and access to the same bank in the same cycle is penalized.

## III. Optimization of SAD Algorithm on TMS320C6x

### *A. Analysis of Software Pipeline Scheduling of SAD Algorithm*

C implementation of SAD algorithm used in UBC H.263 encoder [16] adopted in this paper is as follows.

```
int SAD_Macroblock (unsigned char *ii, unsigned char *act_block,
                    int h_length, int Min_FRAME)
{   //  ii ; memory pointer of search region
    //  act_block ; memory pointer of current macroblock
    //  h_length ; width of search region
    //  INT_MAX ; maximum of unsigned integer (2,147,483,647)
    //  Min_FRAME ; the minimum SAD value among the previously
    //  calculated SAD , initial value = INT_MAX

    int i, j;
    int sad = 0;
    unsigned char *kk;
    kk = act_block;
    i = 16;
    while (i--)
    {
        for (j=0; j<16; j++)    // inner loop
            sad+= abs(*(ii+j)-*(kk+j)); // inner loop
        ii += h_length;
        kk += 16;
        if (sad > Min_FRAME) // conditional branch
            return INT_MAX;
    }
    return sad;
}
```

Fig. 1 C implementation of SAD algorithm

In order to calculate SAD about a pixel pair of a pixel in the current macroblock in the current frame and a pixel in a macroblock in the reference frame, we need two memory load operations, one subtract operation, one absolute operation, and one add operation. Thus, for SAD calculation of 1 macroblock (16 x 16 size), we need 16 x 16 x 2 memory load operations, 16 x 16 x 1 subtract operations , 16 x 16 x 1 absolute operations, and 16 x 16 x 1 add operations. In TMS320C6x, load operations need 5 cycles to finish, and absolute/subtract/add operations need 1 cycle to finish. Thus, if these operations are sequentially processed, that is, one operation is processed after one operation is finished; maximally 3,327 cycles are needed to finish SAD calculation of a macroblock since 16 x 16 x 5 x 2 + 16 x 16 x 1 x 2 + 255 = 3,327. Thus, one may need ILP capability (maximally 8 operations in parallel in one cycle) in TMS320C6x provided by VLIW architecture.

SAD algorithm of Fig. 1 is implemented in a nested loop, and several approaches toward scheduling software pipelining for nest loop are considered as follows.

*M1) One first applies software pipelining techniques to the inner loop and applies again software pipelining techniques to the outer loop whose body consists of software pipelined codes of the inner loop.*
*M2) One first unfolds the inner loop fully and makes a single loop whose body consists of unfolded codes of the inner loop and applies the software pipelining scheduling technique developed for single loop optimization*
*M3) One transforms the nested loop problem into a multi-dimensional software pipelining problem, and applies scheduling technique developed for multi-dimensional software pipelining problem [7]*

TI compiler takes the first approach, but the approach is not efficient since filling and emptying the prolog and epilog of the software pipelined code of the inner loop is repeated whenever outer loop repeats. That is, even if the inner loop is optimally software pipeline scheduled, that does not necessarily mean that the whole nested loop is optimally software pipelined scheduled. The second approach increases code size a lot since fully unfolded inner loop makes a much bigger loop body of outer loop. [14] takes the second approach. But, the increased code size is disadvantageous for embedded implementation since embedded processor has limited memory. Also, big loop body makes optimal scheduling more difficult since one has to consider register renaming and resource constraints more. As for third approach, a general technique has not been developed which can be easily applied for a nested loop with conditional branch.

*B. Optimal Scheduling of SAD Algorithm on VLIW Processor and Implementation on TMS320C67x*

1) Optimal Scheduling of SAD Algorithm on VLIW Processor

In this paper, we propose an improved scheduling for a nest loop with conditional branch of SAD algorithm as follows.

First, we transform the nested loop with conditional branch into a single loop with conditional branch with consideration of utilizing ILP capability of the VLIW processor fully and realization of earlier escape from the loop.

Next, we apply a modulo scheduling technique developed for single loop.

The loop body of the transformed loop will be smaller than that of a single loop made from fully unfolding as in the approach M2). Also, the transformed loop body will be large enough so that the loop body can have enough independent operations to utilize ILP of the VLIW processor. TMS320C6x has ILP capability of executing maximally 8 operations in parallel. Hereafter, we present optimal implementation of SAD algorithm based on this optimal scheduling strategy on TMS320C67x, a VLIW DSP family.

2) Optimal Implementation of SAD Algorithm on TMS320C67x

Considering the optimal scheduling strategy proposed in the above, we transform the nested loop with conditional branch of the SAD algorithm presented in Fig. 1 into a single loop with conditional branch as in Fig. 2.

```
int SAD_Macroblock (  unsigned char *ii, unsigned char *act_block, int
                                h_length, int Min_FRAME  )  {
   // ii ; memory pointer of search region
   // act_block ; memory pointer of current macroblock
   // h_length ; width of search region
   // INT_MAX ; maximum of unsigned integer (2,147,483,647)
   // Min_FRAME ; the minimum SAD value among the previously
    // calculated SAD , initial value = INT_MAX

    int i, l=0;

    int sad = 0, sad_e=0, sad_o=0;

    unsigned char *kk;

    kk = act_block;

    for (i=0 ; i < 64 ; i+=1)   {
        sad_e = abs (*(ii+l) - *kk) + abs (*(ii+2+l) - *(kk+2));
        sad_o = abs (*(ii+1+l) - *(kk+l)) + abs (*(ii+3+l) - *(kk+3));
        sad=sad_e+sad_o;
        if (sad >=Min_FRAME)
                return sad ;
        ii+=4;
        kk+=4;
        if ((i+1)%4 == 0)     //(1)
                l += h_length-16;    //(1)
    }
    return sad;
}
```

Fig. 2 Transformed C implementation of SAD algorithm

The basic idea behind C codes in Fig.2 is the following.

With consideration of the ILP capability of TMS320C67x which can execute maximally 8 operations in parallel and of achievement of earlier escape from loop, the C codes in Fig. 2 is programmed so as to execute SAD operations in parallel as much as possible for two even pixel pairs and two odd pixel pairs, and so as to escape from loop as earlier as possible by checking escape condition right after SAD calculation for every 4 pixel pairs is finished.

Linear assembly codes corresponding to the C codes in bold face in the Fig. 2 can be roughly written as follows.

```
   ZERO     j, sad_e, sad_o, sad
   MVK      0x40, i
   MVK      0x2000, j_init
   SUB      h_length,  16, A_p
   loop:
       LDBU  *A_srcImg[1],      odd_sp1
       LDBU  *B_srcImg[1],      odd_sp3
       LDBU  *A_refImg[1],      odd_rp1
       LDBU  *B_refImg[1],      odd_rp3

       LDBU  *A_srcImg++[4],   even_sp0
       LDBU  *B_srcImg++[4],   even_sp2
       LDBU  *A_refImg++[4],   even_rp0
       LDBU  *B_refImg++[4],   even_rp2

       SUB   even_sp0,  even_rp0,  even_d0
       SUB   even_sp2,  even_rp2,  even_d2
       SUB   odd_sp1,   odd_rp1,    odd_d1
       SUB   odd_sp3,   odd_rp3,    odd_d3

       ABS   even_d0,  even_a0
       ABS   even_d2,  even_a2
       ABS   odd_d1,   odd_a1
       ABS   odd_d3,   odd_a3

       ADD   sad_e,    even_a0,   sad_e
       ADD   sad_o,    odd_a1,    sad_o
       ADD   sad_e,    even_a2,   sad_e
       ADD   sad_o,    odd_a3,    sad_o

       ADD   sad_e,     sad_o,     sad

       CMPLTU sad,   Min_FRAME,   temp
  [temp] B    Get_Out                    //(B.1)
       SUB   i,  1,  i

  [temp] MPY  i,  0,  i                   // (A.1)

         MPY   j,  2,  j                  //  (A.2)
     [!j]  ADD   A_refImg,   A_p,    A_refImg
         ADD   A_refImg,   2,      B_refImg

     [!j]  MPY    j_init,    1,    j          // (A.3)

     [i]   B    loop                        //(B.2)

     Get_Out:
     .return       sad
```

Fig. 3 Linear assembly codes for boldface C codes in Fig. 2

The idea behind linear assembly codes of Fig.3 is that the SAD algorithm is transformed into a single loop so as to utilize the 8 function unit of TMS320C6x fully, and so as to achieve the minimum resource constraint by using remaining M function units (MPYs in (A.1), (A.2), and (A.3) in Fig. 3) for the codes (1) in Fig. 2.

Let us schedule linear assembly codes in Fig. 3 according to the modulo scheduling technique described in Section II.D. .

If we consider that the symbolic registers use different registers, we need 18 registers which is less than 32 registers of TMS320C6x, thus there is not register constraints. Now, let's calculate the MII. Since there are not precedence constraints, we have to consider resource constraints. In Fig. 3, required function blocks are as follows. 8 D function blocks for executing 8 LDBU instructions, 4 L function blocks for 4 ABS, 12 L or S or D function blocks for 7 ADD and 5 SUB, 3 M function blocks for 3 MPY, 1 L function block for CMPGT, 2 S function blocks for 2 B. Thus, the minimum cycles required to schedule instructions using each function block in the software pipeline kernel for Fig. 3 are as follows. $\left\lceil \frac{8}{2} \right\rceil = 4$ for D function blocks, and $\left\lceil \frac{19}{4} \right\rceil = 5$ for L or S function block, and $\left\lceil \frac{3}{2} \right\rceil = 2$ for M function block. Here, $\lceil x \rceil$ is defined to be a least integer greater than or equal to x. Therefore, ResII=5, and II=5. Now, let's search a modulo schedule for II=5 using dependency graph and MRT under constraints: dependency among instructions, delay slots of instructions, and memory bank conflict.

However, in order to find a modulo schedule with II=5, we first have to deal with delay slot of branch instruction. Since the SAD routine in Fig. 3 has two branches: one for repeating loop (B.2), and one for escaping from loop (B.1), and delay latency is 5 for branch instruction, we may not escape from the loop if we are not careful. The reason is that when the branch instruction for escape is executed, the branch instruction for repeating loop is already loaded into the pipeline because II=5 and one cannot separate two branch instructions into more than 4 cycles. Thus the loaded branch instruction will be eventually executed and execution flow returns back to loop. We solve this problem by utilizing conditional branch. The branch instruction for repeating loop is not executed when the loop count becomes 0, thus, we set the loop count as 0 (A.1) when the branch instruction for escape is loaded (B.1). After solving the problem of delay slot of 2 branches, we can find a software pipelining schedule with II=5 for SAD routine in Fig. 3 under constraints. Final optimally scheduled assembly codes are omitted due to the limited paper space.

## IV. Experiments

### A. Experiment Environments

For experiments in this paper, we use TMS320C6713 DSK board from Texas Instruments. C6713 DSK board is provided with 'code composer studio', integrated environments for developing a TI DSP system. Code composer studio supports assembler, C compiler, linker, profiler, debugger, and disassembler. C6713 DSK has 225 MHz TMS320C6713 processor and 8MB 100MHz SDRAM. L2 internal memory of C6713 is set to 240 KB, L2 cash is set to 16 KB. Heap size and stack size are set to 7MB and 64KB, respectively.

The H.263 encoder adopted for experiments is UBC (University of Columbia) Version 2 (H.263+) [16] and we tested baseline mode only. Test raw video sequence is a standard video sequence, foreman.qcif. QCIF size is 176x144.

We use complier option -o2, which does software pipelining.

### B. Experiment Results

For performance analysis, we encoded 3 raw frames with 1 frame encoded in intramode and 2 frames encoded in intermode on C6713DSK board using four H.263 encoders for foreman.qcif. The four H.263 encoders are as follows. 1) UBC encoder: the original UBC H.263 encoder where SAD algorithm of Fig. 1 is used, 2) TI SAD encoder: UBC H.263 encoder but with SAD algorithm replaced by TI library SAD assembly implementation, 3) [14] SAD encoder: UBC H.263 encoder but with SAD algorithm replaced by SAD assembly implementation of [14], 4) Proposed encoder: UBC H.263 encoder but with SAD algorithm replaced by the SAD assembly implementation proposed in this paper.

Table I shows the comparisons of worst required cycles and code size among SAD implementations.

TABLE I
COMPARISONS OF WORST REQUIRED CYCLES AND CODE SIZE AMONG SAD ALGORITHMS

| SAD implementations | Worst cycles | Code size |
|---|---|---|
| UBC SAD (-o2 option) | 739 | 324 |
| TI Library SAD | 258 | 240 |
| [14] SAD | 275 | 808 |
| Proposed SAD | 319 | 252 |

The code size means size of assembly codes of SAD algorithm. The code size of the proposed SAD implementation is far less than that of [14], and 12 bytes more than that of TI library implementation of SAD algorithm. 12 byte difference is very small compared to the whole H.263 code size. The worst cycles mean the worst CPU cycles required to execute SAD routine once. The worst cycles for TI library SAD routine is shorter than that of the proposed SAD, but this is the case about executing SAD once. Since best block matching needs to find minimum SAD macroblock after hundreds of SAD calculations, H.263 encoding using the proposed SAD implementation performs much better than that using TI library SAD, which is shown in Table II.

TABLE II
COMPARISONS OF CPU CYCLES FOR ENCODING 3 FRAMES

| SAD Implementation | SNR | | | cycles |
|---|---|---|---|---|
| | Y | Cr | Cb | |
| C version (with -o2 option) | 31.25 | 38.78 | 38.82 | 150,925,893 |
| TI Library SAD | 31.55 | 38.58 | 39.15 | 124,580,125 |
| [14] SAD | 31.34 | 38.85 | 39.06 | 109,273,707 |
| Proposed SAD | 30.90 | 38.25 | 38.58 | 76,484,977 |

The experimental data in Table II are obtained when we encoded 3 raw frames from a video sequence, foreman.qcif with 1 frame in intramode encoding and 2 frames in intermode encoding using UBC H.263 baseline encoder with various SAD implementations on TMS320C6713 DSK board.

We can see the proposed SAD implementation works better than any other SAD implementation in Table II, and the H.263 encoder using the proposed SAD implementation is 97% faster than the one using original C version of SAD compiled with level two optimization for encoding 3 frames of foreman.qcif with 1 frame in intramode and 2 frames in intermode.

## V. CONCLUSION

In this paper, we investigated an optimal scheduling and implementation of SAD algorithm with conditional branch on a VLIW DSP processor. SAD algorithm is usually implemented as a nested loop with a conditional branch. Scheduling a nested loop with conditional branches optimally is not an easy job. We proposed a transformed approach. The proposed optimal scheduling in this paper first transforms the nested loop with conditional branch into a single loop with conditional branch with consideration of utilizing ILP capability of the VLIW processor fully and realization of earlier escape from the loop. Next, the proposed optimal scheduling applies a modulo scheduling technique developed for single loop. Based on this optimal scheduling strategy, optimal implementation of SAD algorithm on TMS320C67x, a VLIW DSP was presented. Through experiments on TMS320C6713 DSK, it was shown that H.263 encoder with the proposed SAD implementation performs better than other H.263 encoder with other SAD implementations, and that the code size of the optimal SAD implementation is small enough to be appropriate for embedded environments.

## ACKNOWLEDGMENT

This work was supported by the Soongsil University Research Fund and BK21.

## REFERENCES

[1] P. G. Paulin, et al., “Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends,” Proc. of IEEE, Vol. 85, No.3, pp. 419-435, Mar. 1997.
[2] M. Budagavi et. al., “Wireless MPEG-4 Video Communication on DSP chips,” IEEE Signal Processing Magazine, pp. 36-53, Jan. 2000.
[3] J. Hennessy and D. Patterson, Computer Architecture A Quantitative Approach, 3rd ed., MK Pub. 2003.
[4] J. A. Fisher, P. Farabosch, and C. Young, Embedded Computing A VLIW Approach to Architecture, Compilers, and Tools, Morgan Kaufmann, 2004.
[5] K. Muthukumar and G. Doshi, “Software pipelining of nested loops,” In R. Wilhelm, editor, *CC 2001*, LNCS 2027, pages 165-181. Springer-Verlag, Berlin Heidelberg, 2001.
[6] R. Scales, *Nested loop optimization on the TMS320C6x. Application Report SPRA519*, Texas Instruments, Feb. 1999.
[7] Q. Zhuge, Z. Shao, and E. H.-M. Sha, “Optimization of Nest-Loop Software Pipelining,” Submitted paper, http://www.utdallas.edu/~edsha/
[8] G. Gupta, and C. Chakrabarti, “Architectures for hierarchical and other block matching algorithms,” Circuits and Systems for Video Technology, IEEE Transactions on Volume 5, Issue 6, pp. 477-489, Dec. 1995.
[9] W. Hwang, Y. Lu, Y. Zeng, “Fast block-matching algorithm for video coding,” Electronics Letters Volume 33, Issue 10, pp. 833 - 835, May 1997
[10] D. Talla, L.K. John, V. Lapinskii, and B.L. Evans, “Evaluating signal processing and multimedia applications on SIMD, VLIW and Superscalar architectures,” Proceedings of 2000 International Conference on Computer Design, pp. 163-172, Sept. 2000.
[11] S. M. Akramullah, I. Ahmad, I. and M.L. Liou, “Optimization of H.263 video encoding using a single processor computer: performance tradeoffs

and benchmarking," Circuits and Systems for Video Technology, IEEE Transactions on Volume 11, Issue 8, pp. 901-915, Aug. 2001.
[12] H. Miyazawa, H.263 Encoder: TMS320C6000 Implementation, Application Report SPRA721, Texas Instruments, December, 2000
[13] O. Lehtoranta, T. Hamalainen, J. Saarinen, "Real-time H.263 encoding of QCIF-images on TMS320C6201 fixed point DSP," Circuits and Systems, Proceedings of IEEE International Symposium on Circuits and Systems, Volume 1, 28-31 pp. 583 - 586, 2000.
[14] S. Bangerjee, et.al., "VLIW DSP vs. Superscalar Implementation of a Baseline H.263 Video Encoder," 34th IEEE Alsilomar Conf. on Signals, Systems and Computers, vol. 2, pp. 1665-1669, 2000.
[15] TMS320C62x Image Library, http://focus.ti.com/docs/toolsw/ folders/ print/sprc093.html
[16] B. Erol, F. Kossentini, and H. Alnuweiri, "Implementation of a fast H.263+ encoder/decoder," Proc. IEEE Asilomar Conf. Alsilomar Conf. on Signals, Systems and Computers, vol.1, pp.462-466, Nov. 1998.
[17] Iain E G Richardson, *Video CODEC Design Developing image and video compression systems*, John Wiley & Sons, 2002.
[18] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 318-328, Jun. 1988.
[19] B. R. Rau, "Iterative Modulo Scheduling," International Journal of Parallel Processing, Vol. 24, No. 1, Feb. 1996.
[20] TMS320C6713 Datasheet, No. SPRU186D, Texas Instruments, May, 2003.
[21] TMS320C6000 CPU and Instruction Set Reference Guide, No. SPRU189F, Texas Instruments, Oct., 2000.