# Efficient Vectorization of SIMD Programs with Non-aligned and Irregular Data Access Hardware

Hoseok Chang
Seoul National University
599 Gwanak-ro
Gwanak-gu, Seoul, Korea
chs@dsp.snu.ac.kr

Wonyong Sung
Seoul National University
599 Gwanak-ro
Gwanak-gu, Seoul, Korea
wysung@snu.ac.kr

## ABSTRACT

Automatic vectorization of programs for partitioned-ALU SIMD (Single Instruction Multiple Data) processors has been difficult because of not only data dependency issues but also non-aligned and irregular data access problems. A non-aligned or irregular data access operation incurs many overhead cycles for data alignment. Moreover, this causes difficulty in efficient code generation and hinders automatic vectorization. In this paper, we employ special memory access hardware for improving the performance of SIMD processors; one is the split line buffer and the other is the packing buffer. The former solves the non-aligned memory access problem, while the latter simplifies irregular and stride data access. The addition of these hardware units not only requires very small changes to the instruction set architecture but also contributes to the significant performance improvement by vectorizing more loops and reducing the overhead cycles. We have also developed an auto-vectorization compiler which utilizes these special hardware units. Experiments have been conducted to compare the proposed method with the conventional one, which show 50% increase in the number of vectorized loops and 77% increase in the total performance of an MPEG2 encoder program.

## Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids; D.3.4 [**Programming Languages**]: Processors—*Optimization*

## General Terms

Design, Performance

## Keywords

SIMD, split line buffer, packing buffer, non-aligned access, irregular access, vectorization, compiler

**Table 1: Statistics of SIMD memory accesses in MPEG2 encoder with a 64-bit SIMD processor**

| Function | Number of aligned access | Number of non-aligned access |
|---|---|---|
| ME | 2.67M (20%) | 10.73M (80%) |
| MC | 0.04M (25%) | 0.12M (75%) |
| DCT | 2.63M (51%) | 2.46M (48%) |
| IDCT | 0.15M (43%) | 0.41M (57%) |
| Quant/IQuant | 0.07M (33%) | 0.14M (66%) |
| VLC | 0.01M (38%) | 0.02M (62%) |
| Total | 5.73M (29%) | 13.88M (71%) |

## 1. INTRODUCTION

The SIMD (Single Instruction Multiple Data) processor architecture not only requires a relatively simple hardware, but is also very effective in executing programs containing large data-level parallelism such as multimedia applications. In recent days, the SIMD architecture adopting partitioned data-paths is widely employed in both personal computing and embedded processors such as Intel Pentium 4, Texas Instruments TMS320C64x, Intel PXA27x and ARM Cortex [2][3][12][20]. The partitioned data-path can process multiple aligned data elements using a single processor instruction. However, the data elements are not always stored in an aligned or regular manner. Complex data access in SIMD processors can be categorized into non-aligned and irregular access operations. In the non-aligned data access, the start address is not on the alignment boundary, thus shift operations are needed for alignment. The irregular data access is more complex because the vector elements are not in order although their addresses are usually within some small bound. Table 1 shows the distribution of aligned and non-aligned/irregular data access in the MPEG2 encoder program when executed with a 64-bit SIMD processor. We can find that 71% of SIMD memory access operations are non-aligned or irregular. Not only additional memory access operations but also special instructions, such as *pack* and *shuffle*, are needed for aligning the data. Moreover, these overhead operations add another level of complexity for automatic vectorization.

Several auto-vectorizing compilers such as IBM's XL compiler, Intel compiler, ARM's RealView compiler, and GNU's gcc compiler have been introduced for efficient SIMD code generation [4][7][17][19]. However, this software based approach cannot easily eliminate all overheads. Even manually optimized SIMD programs contain many overhead instructions for data alignment. Moreover, when an SIMD compiler is not certain about the data alignment, it usually generates inefficient scalar codes to avoid dynamic alignment overhead.

One of the alternate approaches for improving the code quality is to employ special hardware for automatic data alignment. In our proposed method, we have added two hardware units to an SIMD processor to reduce alignment overhead; one is the split line buffer for non-aligned data, and the other is the packing buffer for irregular data. The split line buffer consists of one line buffer and a merge unit. The proposed packing buffer contains a small size multi-port memory block for which multiple addresses are provided by a vector index register. Since the size of the packing buffer is small, it neither requires complex hardware resources nor increases the CPU cycle time. We have also developed a compiler framework to utilize the small sized packing buffer. This compiler framework also supports performance measurements, for which multimedia benchmarks are used.

The rest of this paper is organized as follows. Section 2 briefly describes the related work. The non-aligned access problem and supporting hardware are presented in Section 3. The irregular access problem and the packing buffer architecture are explained in Section 4. The compiler framework is discussed in Section 5. The performance evaluation results are shown in Section 6. Finally concluding remarks are made in Section 7.

## 2. RELATED WORKS

There have been many research works on improving the SIMD architecture to overcome the memory access bottlenecks. The classical pipelined vector processors employed interleaved memory units to increase the bandwidth; however the memory units suffered from bank conflicts. Some pipelined vector processors equipped the gather and scatter unit that stores the data address sequence at a separate buffer. This separate buffer is used for the vectorization of irregular data access operations such as those found in the sparse matrix computation [6]. Recently, the SIMdD (Single Instruction Multiple disjoint Data) architecture contains a multi-port memory unit which allows accessing disjoint data [18]. Although the SIMdD supports non-aligned and irregular data access efficiently, it is based on costly multi-port memory. The other memory architectures such as multi-bank scratch-pad memory and dual-bank cache memory are studied in [8][5]. The multi-bank memory units perform non-aligned and stride access efficiently, while the dual-bank cache can efficiently support only non-aligned access. A decoupled architecture named as MediaBreeze was proposed in [22]. The MediaBreeze contains an execute processing unit and an access processing unit. While the execute processing unit is carrying out the SIMD arithmetic operations, the access processing unit separately conducts the data access and alignment.

Many compiler algorithms have been studied for reducing the memory access overhead problems. In [10], several strategies are proposed to minimize the number of shift operations needed to remove the alignment offset of vector data. In [25], the runtime variation of the alignment offset is considered. The alignment of array access in a loop can be simplified by employing loop transformation methods. The loop peeling transformation is a popular method for handling non-aligned data access [15]. In order to auto-vectorize interleaved data access, another proposed method is to generate overhead instructions when the access stride is powers of 2 [14]. In addition, most of SIMD processors provide pack, permutation and shuffle instructions to arrange data within vector registers in various patterns. The methods for generating pack or permutation instructions are presented in [14][21]. Although these compiler algorithms reduce the overhead of complex data access, the performance is not always satisfactory due to the inherent hardware related restrictions on data access.

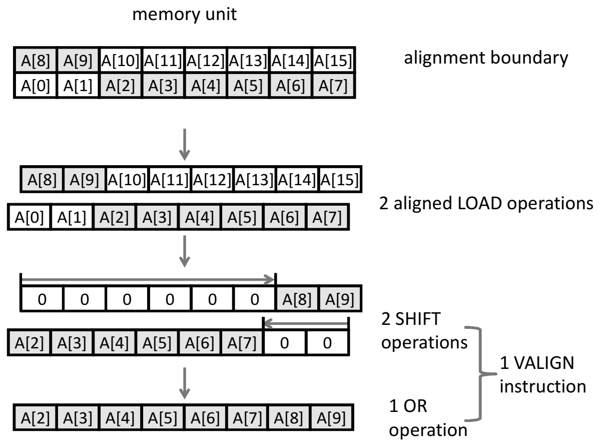

memory unit
A[8] A[9] A[10] A[11] A[12] A[13] A[14] A[15]
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]
alignment boundary
2 aligned LOAD operations
0 0 0 0 0 0 A[8] A[9]
A[2] A[3] A[4] A[5] A[6] A[7] 0 0
2 SHIFT operations
1 VALIGN instruction
1 OR operation
A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]

**Figure 1: Example of non-aligned access**

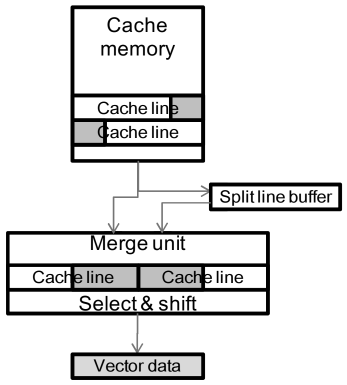


**Figure 2: Single-bank cache with a split line buffer**

## 3. NON-ALIGNED DATA ACCESS AND HARDWARE SUPPORT

Non-aligned data access is very frequent in multimedia application programs. The frequency of non-aligned access increases as the width of SIMD ALU lanes widens. In most programs, non-aligned data access is much more frequent than irregular data access.

In simple SIMD processors, a non-aligned data read requires an extra load followed by data manipulation operations, which include shifting, masking, and merging of two vector data. An example of non-aligned data access is shown in Fig. 1.

Non-aligned data access causes difficulties for efficient auto-vectorization. The alignment of array access in a loop can be decided by the base address of the array and the offset

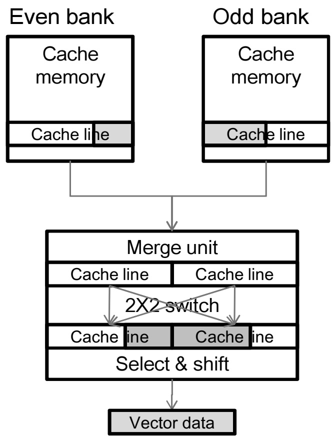


**Figure 3: Dual-bank cache with non-aligned access support**

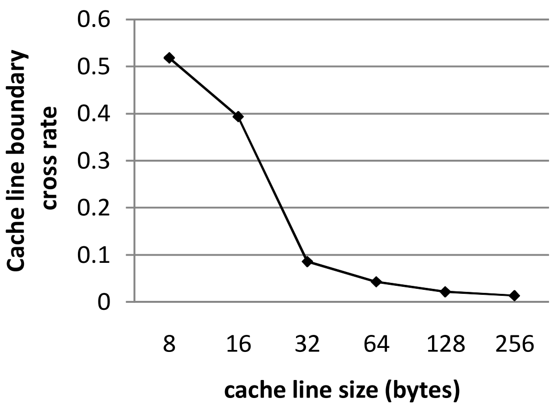


**Figure 4: Cache line boundary cross rate according to the cache line size**

value obtained from the induction variable analysis. If the alignment of a data access is known at compile time, the compiler can generate overhead instructions when needed. However, in many cases, the alignments information is not available at compile time. For example, if the array to be accessed is passed from a caller function, the base address cannot be known. In this case, even if the data is actually aligned, the compiler should generate overhead instructions handling the alignment offset dynamically. Since dynamic offset compensation is much more complex, many vectorization candidate loops are not auto-vectorized. Therefore, the hardware support for non-aligned access is critical for efficient auto-vectorization.

In order to support non-aligned access in hardware, a 'select and shift' unit is needed for arranging the vector data. The cache memory structure equipping a 'select and shift' unit with a split line buffer is shown in Fig. 2. The aligned vector data always resides within a cache line, however, the non-aligned vector may be laid across the cache line boundary. To handle the cache line split situation, the split line buffer is used [26]. When the vector data spans through two cache lines, the split line buffer stores one cache line temporarily while the other one is being accessed. Then, those two cache lines are merged to produce the desired vector data. Although the split line buffer does not alter the existing cache structure much, it needs additional cycles in case of the cache line split.

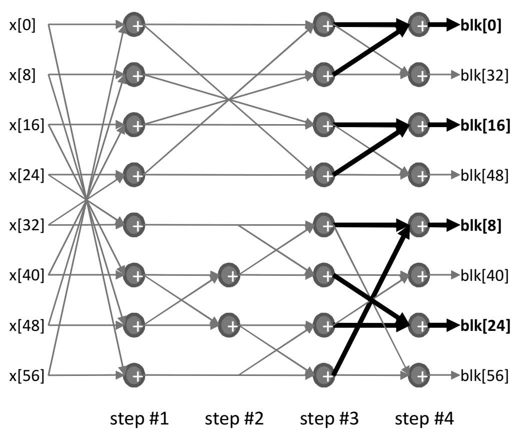
The other hardware architecture for solving the non-aligned access problem is the dual-bank cache, the structure for which is shown in Fig. 3 [5][11]. The dual-bank cache is composed of odd and even memory banks. If the desired vector spans across the cache line boundary, the addresses for even and odd banks are calculated. The even/odd bit in the address indicates the first bank that holds the beginning part of the vector. The other bank stores the remaining part of the vector. The two banks are accessed in parallel and the two outputs are merged into a vector in the merge unit. The dual-bank cache has a performance advantage when both cache lines are hit because of simultaneous access of two cache lines. As the cache line size increases, the possibility of cache line split becomes lower; this applies both for the split line buffer cache and the two-bank cache. The cache line boundary cross rate in the MPEG2 encoder is shown in Fig. 4. The rate drops rapidly when the cache line is longer than 16 bytes. Since the additional overhead of sequential load in the split line buffer cache is not significant when the cache line boundary cross rate is low, we have employed the split line buffer cache in our proposed mechanism.

## 4. IRREGULAR DATA ACCESS AND HARDWARE SUPPORT

The signal flow diagram of Chen-Wang DCT is displayed in Fig. 5 (a) [24]. There exists a significant amount of data parallelism in it because each step of DCT contains common arithmetic operations. However, the data access is irregular, thus several special instructions are needed for preparing vectors in desired patterns and extra registers are also required for storing intermediate results. Irregular access is also shown in table-based arithmetic, in which the access pattern of data is dependent on the input. Due to the large overhead of arranging such access, it is very difficult to vectorize basic blocks containing irregular access in conventional SIMD processors.

In order to vectorize irregular data access, we adopt a small multi-port memory unit with an index register which is shown in Fig. 6. For an irregular data access in an array, we store the array indexes into the index register, and conduct a SIMD load with the input operands of the base register and the index register. The address of each data element is generated by adding the base address and the index value. Then, the calculated addresses are sent to the multi-port memory and the desired vector is obtained. The multi-port memory needs to be integrated into the conventional memory system to obtain a high performance gain with a small capacity. This hardware support is named as the packing buffer. The size of the packing buffer needs not be large. About the size of 128 bytes is enough for MPEG2 video encoding.

The packing buffer can be managed in two different ways; one is as an on-chip memory block and the other is as cache. When the packing buffer is used as on-chip memory, a dynamic memory management scheme is needed. The dynamic memory management software, which includes copy and copy-back functions, is supported at the developed SIMD



(a) Simplified signal flow diagram of Chen-Wang DCT in vertical way

```
for (i = 0; i < 8; i ++) {
  …
  blk[0] = (x[0]+x[8])>>8;
  blk[8] = (x[32]+x[56])>>8;
  blk[16] = (x[16]+x[24])>>8;
  blk[24] = (x[48]+x[40])>>8;
  …
}
```

(b) C source code from 8×8 Chen-Wang DCT

**Figure 5: Example of irregular data access**

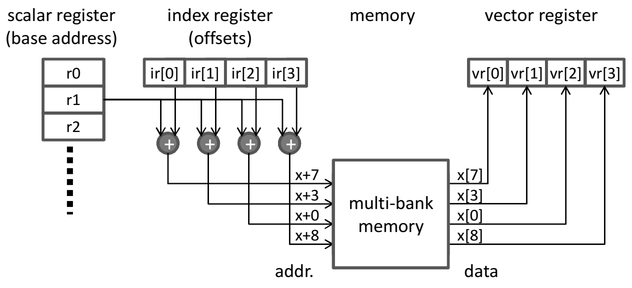


**Figure 6: Index register-based 4-port memory**

compiler framework. Although this scheme needs an extra software overhead, its operation at runtime is simple and efficient because it provides a separate address space to the packing buffer and there is no need of concerning the coherency problem.

When the packing buffer operates as cache, it needs a tag memory block and tag-comparing logic in addition to the multi-port memory with the index register. Because of this, it can be called the packing cache. The structure of the cache memory system including the packing cache is shown in Fig. 7. When an irregular data access is requested, the base address and the index values are sent to the packing cache and then the corresponding upper bits are compared with tags. In order to simplify the tag compare logic, the cache line in the packing cache needs to be large enough to cover the whole range of the index. If each index is $i$-bits, the cache line needs to be $2^{i+1}$ bytes to make room for entire indexed data. For example, when the index offset is represented in a 6-bit unsigned integer, the packing buffer should contain at least 64 data elements, which corresponds to 128 bytes for 16 bit data. The packing cache structure has a coherency problem with the cache memory. For coherency, the packing cache is accessed even in ordinary memory access, and the data is served only when the desired data already exists in the packing cache. The packing cache does not need copy or copy-back functions, which are automatically conducted. Although the packing cache can be more stable in dynamic program execution environments, it needs extra tag comparison for every memory access. Thus it is desired to inhibit this cache when not needed.

## 5. COMPILER FRAMEWORK

A compiler framework that supports the split line and the packing buffers has been developed based on the GCC v.4.2 [8]. The vectorizer of GNU's GCC v4.2 are depicted in Fig. 8. An inner-most loop that has a larger, or equal, loop count than the number of SIMD ALU lanes becomes a vectorization candidate loop. The vectorization steps are composed of two phases. The first contains the vectorization analysis step that examines dependence and access pattern to filter out loops that cannot be vectorized. If a loop passes the vectorization analysis phase, the loop is converted into SIMD codes in the vectorization transform phase.

The first step of the vectorization analysis phase is to find a candidate loop and analyze the data access pattern which can be found with the induction variable analysis. Next, the compiler tests the scalar variables in a loop and finds the reduction operations such as accumulation or summation of unit data. After that, the data dependency test is conducted to find dependency relations among loop iterations that might prohibit vectorization. The next step is to analyze the memory access patterns. Generally, most of the array indexes and pointer offsets are derived from the induction variables in the loops. The stride and the alignment of memory accesses are evaluated by the induction variable analysis. The alignment analysis step can be skipped because the designed memory system supports non-aligned access as efficiently as the aligned one. In conventional SIMD processors with non-aligned access support, even though the non-aligned access instructions allow aligned addresses also, the compiler should maximize the usage of aligned load instructions for obtaining good performance.

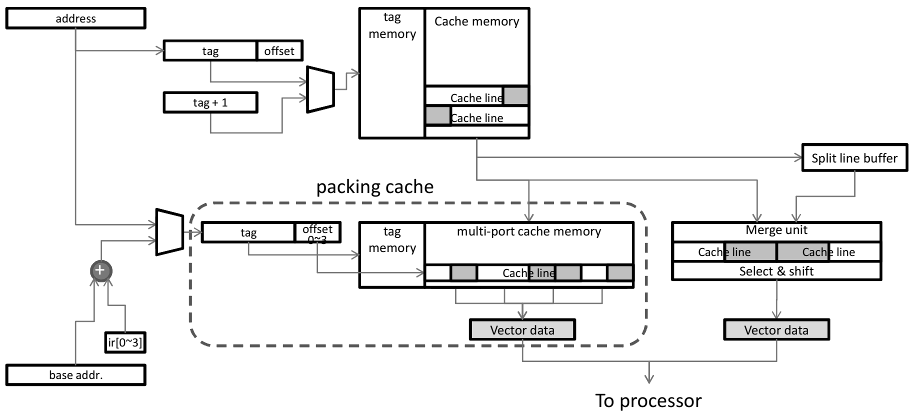


**Figure 7: Cache memory system for non-aligned and irregular data access**

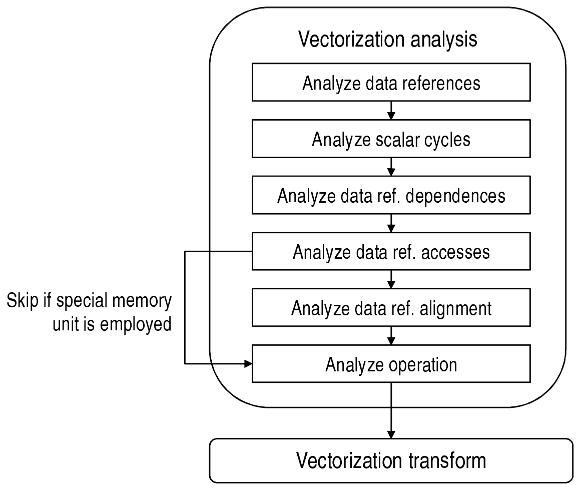

Vectorization analysis
Analyze data references
Analyze scalar cycles
Analyze data ref. dependences
Analyze data ref. accesses
Analyze data ref. alignment
Analyze operation
Skip if special memory unit is employed
Vectorization transform

**Figure 8: Automatic vectorization part in GCC v4.2**

The examples in Fig. 9 illustrate several vectorization techniques for handling non-aligned data access. The original C code in Fig. 9 (a) shows that the array 'in' is accessed in a non-aligned manner, and the alignment offset is varying because it is a function of the index for the outer loop, 'j.' Non-aligned access can be removed by a special data layout or program restructuring. Fig. 9 (b) shows a code that eliminates non-aligned access by using multiple versions of coefficients. The constant versioning is one of frequently used manual assembly techniques. Although this needs more memory space, it shows a much better performance. Fig. 9 (c) shows a vectorized program using the dynamic loop peeling technique. Both prologue and epilogue codes are included. The loop peeling is also employed in many SIMD compilers. Both the constant versioning and the loop peeling are used in commercial performance optimized libraries [1]. If non-aligned access is supported by hardware, the vectorization is the simplest and best among the cases as examplified in Fig. 9, which shows that the loop can be fully vectorized without employing advanced loop transformations.

With the proposed packing buffer, the compilers can even vectorize loops containing irregular data access. The ex-

```
for (j = 0; j < 1000; j ++) {
  for (i = 0; i < 128; i ++) {
    out[j] += in[j + i] * c[i];
  }
}
```

(a) Original C code

```
c_v[0] = {c[0], c[1], …, c[127], 0, 0, 0, 0};
c_v[1] = {0, c[0], c[1], …, c[127], 0, 0, 0};
c_v[2] = {0, 0, c[0], c[1], …, c[127], 0, 0};
c_v[3] = {0, 0, 0, c[0], c[1], …, c[127]};

for (j = 0; j < 1000; j +=4) {
  for (i = 0; i <= 128; i += 4) {
    out[j] = mac(out[j], in[j + i:j + i + 3],
c_v[j % 4][i: i + 3]);
    out[j + 1] = mac(out[j + 1], in[j + i:j
+ i + 3], c_v[j % 4][i: i + 3]);
    out[j + 2] = mac(out[j + 2], in[j + i:j
+ i + 3], c_v[j % 4][i: i + 3]);
    out[j + 3] = mac(out[j + 3], in[j + i:j
+ i + 3], c_v[j % 4][i: i + 3]);
  }
}
```

(b) Vectorized by constant versioning

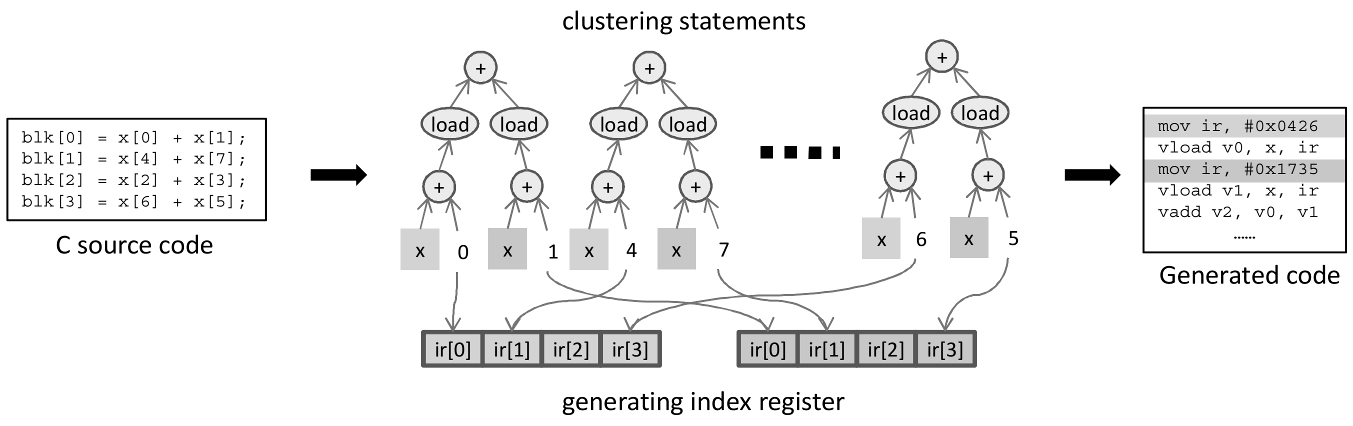
```
for (j = 0; j < 1000; j ++) {
  for (i = 0; (i + j) % 4 != 0; j ++) {
    out[j] += in[j + i] * c[i];
  }
  for (; i < 128; i += 4) {
    out[j] = mac(out[j], in[j + i:j + i + 3],
c[i: i + 3]);
  }
  for (; i < 128; j ++) {
    out[j] += in[j + i] * c[i];
  }
}
```

(c) Vectorized by dynamic loop peeling

```
for (j = 0; j < 1000; j ++) {
  for (i = 0; i < 128; i += 4) {
    out[j] = mac(out[j], in[j + i:j + i + 3],
c[i: i + 3]);
  }
}
```

(d) Vectorized with hardware support of non-aligned access

**Figure 9: Vectorization of FIR filter kernel**



```
blk[0] = x[0] + x[1];
blk[1] = x[4] + x[7];
blk[2] = x[2] + x[3];
blk[3] = x[6] + x[5];
```

```
mov ir, #0x0426
vload v0, x, ir
mov ir, #0x1735
vload v1, x, ir
vadd v2, v0, v1
……
```

**Figure 10: Vectorization flow for irregular data access**

ample of irregular access vectorization is described in Fig. 10. Firstly, the compiler searches for the statements that contain the identical operations. Secondly, if the searched statements uses the same operand array with a common base address, these statements can be clustered and translated to SIMD codes. The packed data are addressed by the elements of the index register. The compiler needs to generate instructions that manage the index register, where the index register values are obtained by the induction variable analysis.

The start and the end addresses of data needed for the packing buffer are also calculated. In previous on-chip memory management works [13][23], a data peeling algorithm is used to accommodate arrays larger than the on-chip memory size. In our case, data peeling is not employed because the range of irregular access is small.

The examples in Fig. 11 show vectorization results for the Chen-Wang DCT code in Fig. 5 (b) which contains irregular data access. In a conventional SIMD case, the operand vectors need to be rearranged with pack and shuffle instructions. The pack instruction merges two vectors into one and the shuffle instruction changes the order of the elements within a vector. In the case of the packing buffer or the packing cache, the rearrangement is conducted using the index register instructions. The contents of the packing buffer are controlled by 'fetch' and 'evict' instructions. In the packing cache, the contents are controlled by cache mechanism, and the miss penalty is close to the copy cost in the packing buffer case. The conventional SIMD processor consumes 171.4% more cycles for the displayed part of the inner-most loop when compared with the SIMD equipping the packing buffer or the packing cache. In terms of the register usage, the conventional SIMD uses 8 vector registers, while the packing buffer or cache case needs only 3 vector registers plus one index register. The reduced register usage gives advantages in lowering the register spill cost and function call overheads.

## 6. EXPERIMENTAL RESULTS

The SIMD processor for the performance evaluation is designed with the proposed hardware support. The processor is based on ARMv4 architecture and includes a 128-bit SIMD ALU and 16 128-bit vector registers. The SIMD instruction set is shown in Table 2. All SIMD instructions except for SIMD memory instructions take one cycle delay.

**Table 2: Instruction set of proposed SIMD architecture**

| Category | | Instruction |
|---|---|---|
| Arithmetic | Logical operation | VOR, VXOR, VAND, VNAND |
| | Shift operation | VSRA, VSLL, VSRL, VROR |
| | Compare | VCMPEQ, VCMPGT |
| | Computation | VADD, VSUB , VABS, VMUL |
| | Multiply & accumulate | VMAC |
| | Reduction operation | VSUM |
| Memory | Load/store | VLD, VLDSH, VST, VSTSH |
| Special | | VALIGN, VPACK, VUNPACK, VSHUF |

DSP kernels and an MPEG2 encoder are used as benchmarks. The DSP kernels include a 16-tap FIR filter, 12th order IIR filter, 16×16 2D-DCT and 1024-point FFT. These benchmarks contain complex data access patterns. The FIR filter includes non-aligned access for input data. The IIR filter needs both non-aligned and stride access. The 2D-DCT benchmark uses the Chen-Wang algorithm [24], and the access pattern is irregular. The FFT benchmark is based on the Cooley-Tukey algorithm [9], and it contains stride and irregular access. The MPEG2 encoder benchmark is included in the MediaBench [16]. The MPEG2 encoder benchmark consists of motion estimation, motion compensation, DCT, IDCT and so on.

The memory access cycles for each cache memory are shown in Table 3. We assume that the external memory is a 166MHz, 32-bit wide SDRAM module and the processor runs at 500MHz. Due to the burst operation mode of external SDRAM, the miss cycle counts are not linear to the line size. Note that, in the case of the split line buffer, maximum two misses can be occurred in one non-aligned vector access.

```
LOOP
  …
  vload v0, x, #0
  vload v1, x, #32
  vload v2, x, #16
  vload v3, x, #48
  vpackl v4, v0, v1
  vpackl v5, v2, v3
  vpackl v6, v4, v5
  vshuf v6, #0x0213
  vload v0, x, #8
  vload v1, x, #56
  vload v2, x, #24
  vload v3, x, #40
  vpackl v4, v0, v1
  vpackl v5, v2, v3
  vpackl v7, v4, v5
  vshuf v6, #0x0312
  vadd v0, v6, v7
  vshr v0, v0, #8
  vstore blk,  v0
  …
  b LOOP
```

(a) Compiled code using conventional SIMD instructions

```
 fetch x
LOOP
  …
  mov ir, #0x00201030
  vload v0, x, ir
  mov ir, #0x08381828
  vload v1, x, ir
  vadd v2, v0, v1
  vshr v2, v2, #8
  vstore blk,  v2
  …
  b LOOP
  evict x
```

(b) Compiled code using packing buffer instructions

```
LOOP
  …
  mov ir, #0x00201030
  vload v0, x, ir
  mov ir, #0x08381828
  vload v1, x, ir
  vadd v2, v0, v1
  vshr v2, v2, #8
  vstore blk,  v2
  …
  b LOOP
```

(c) Compiled code using packing cache instructions

**Figure 11: Compiled code example of Chen-Wang DCT**

**Table 3: Memory access cycles according to the cache memory type**

| | Split line buffer cache (32B line) | Packing buffer (128B line) |
|---|---|---|
| Hit | 1 cycle (2 cycles for line split) | 1 cycle |
| 1 miss | 36 cycles | 108 cycles (entire line fetch from L1 cache) |
| 2 misses | 60 cycle | N/A |

Table 4 shows the performance results for DSP kernels. The conventional SIMD processor is assumed to contain ordinary 16Kbytes cache memory without the non-aligned and irregular access supports. With the conventional SIMD processor, one non-aligned vector access requires two aligned vector access and one merge (VALIGN) operation. Moreover, loops containing irregular access are not vectorized. The FIR benchmark needs non-aligned access for input data. Therefore, the split line buffer case shows much better performance than the reference. The IIR benchmark contains non-aligned and stride access. The stride access is well supported by the packing buffer. As a result, the performance with the split line and packing buffers is better than that with the split line buffer only. In the 2D-DCT benchmark, almost all vector data access operations are irregular, hence, the split line buffer shows a small speed-up compared to the conventional SIMD, but the packing buffer achieves a large improvement. The bit-reverse shuffling part in the FFT benchmark is very difficult to vectorize and shows a high cache miss rate of 27.4%. By adopting the packing buffer, the shuffling operations can be vectorized with index register based memory access instructions. In addition, the packing buffer reduces the cache miss rate of the shuffling part into 19.8%. Since multimedia workloads contain low temporal locality but high spatial locality, the packing buffer with long line size shows good performance.

**Table 4: Performance of DSP kernels**

| | | Conventional SIMD | With split buffer | With split and packing buffer |
|---|---|---|---|---|
| FIR | Cycles | 70,674 | 39,703 | 39,703 |
| | Speed-up | 1 | 1.78 | 1.78 |
| IIR | Cycles | 126,142 | 43,235 | 33,562 |
| | Speed-up | 1 | 2.92 | 3.76 |
| 2D-DCT | Cycles | 73,268 | 63,442 | 38,696 |
| | Speed-up | 1 | 1.15 | 1.89 |
| FFT | Cycles | 243,664 | 243,664 | 205,752 |
| | Speed-up | 1 | 1 | 1.18 |

The performance of the MPEG2 encoder is shown in Fig. 12. We assume that the cache memory is 16 Kbytes and the packing cache is 1 Kbytes with 128 bytes line size. With the split line buffer, the performance is greatly increased and shows the speed-up of 155%. The performance enhancement is mainly from the motion estimation part. The motion estimation is memory access intensive, and 80% of the memory access operations are non-aligned. The performance with the split and the packing buffers shows the speed-up of 177% when compared to the conventional SIMD. Since the packing buffer is efficient for irregular data access, the DCT part also shows improved performance.

The effects on the number of vectorized loops according to the memory access hardware supports are shown in Table 5. This table clearly shows that more loops are vectorized by employing the memory access hardware support. This is due to reduced overheads for non-aligned and irregular data access.

The performance according to the width of SIMD ALU is shown in Fig. 13. The capacity of the L1 cache is 16 Kbytes and that of the packing buffer is 1 Kbytes. The performance increase according to the width of SIMD ALU is dependent

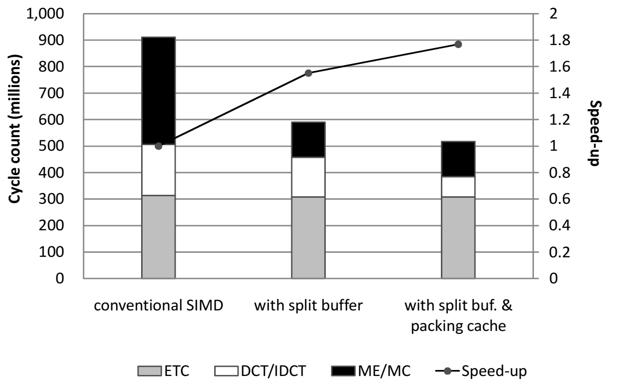


**Figure 12: Performance of MPEG2 encoder according to the cache configuration**

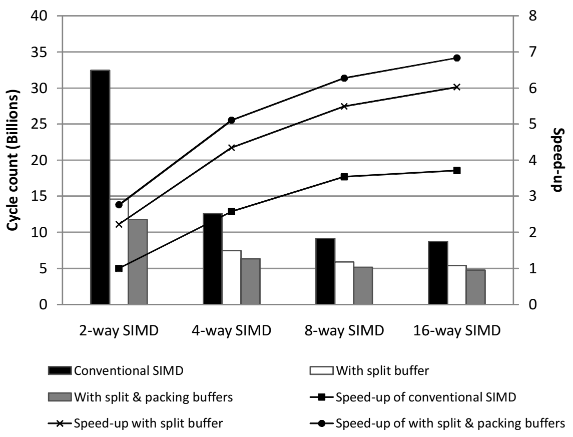


**Figure 13: Performance of MPEG2 encoder according to the width of SIMD ALU**

on the memory type. As the width of SIMD ALU widens, the chance of non-aligned access and the overhead of alignment also grow. Hence, the performance of the conventional SIMD architecture almost saturates in the 8-way SIMD configuration, while the split line and the packing buffer cases exhibit increased performance even in the 16-way SIMD processor.

## 7. CONCLUDING REMARKS

The vectorization efficiency of an SIMD compiler is usually bounded by the memory related bottlenecks, which are the overheads for preparing aligned vector data. Although improved performance can be achieved by the careful restructuring of programs or the data layout, which are usually conducted in manual assembly programming, current SIMD compilers have difficulties in performing these complex tasks.

In our proposed mechanism, an efficient memory system with the split line buffer and the packing buffer/cache is designed for reducing the non-aligned and irregular data access overhead. The split line buffer efficiently supports non-aligned data access, which is very frequent in signal processing, while the packing buffer/cache reduces the overhead of irregular and stride data access.

We showed that the GCC v4.2 compiler can be easily modified for our SIMD processor equipping the memory access hardware support. Due to this hardware support, the vectorization analysis steps become simpler and the compiler can exploit more data-level parallelism. An MPEG2 encoding application was tested with the proposed architecture and the designed SIMD compiler. The 128-bit SIMD processor with the split line and the packing buffer units shows a speed-up of 1.77 when compared to the SIMD processor with a conventional memory system.

**Table 5: Number of vectorized loops**

| | Number of vectorization candidate loops | Number of vectorized loops | | |
|---|---|---|---|---|
| | | Conventional SIMD | Split line buffer | Split line & packing buffers |
| ME | 20 | 10 | 19 | 19 |
| MC | 7 | 3 | 7 | 7 |
| DCT | 5 | 4 | 4 | 5 |
| IDCT | 3 | 2 | 2 | 3 |
| (I)QUANT | 7 | 4 | 4 | 4 |
| etc | 17 | 7 | 7 | 7 |
| total | 59 | 30 | 43 | 45 |

## 8. ACKNOWLEDGMENTS

This work was supported in part by the Samsung Electronics Inc. and in part by the Ministry of Education, Science and Technology (MEST), Republic of Korea, under the Brain Korea 21 Project.

## 9. REFERENCES

[1] *Intel Integrated Performance Primitives for Intel Pentium Processors and Intel Itanium Architectures.* Intel Corporation.

[2] *TMS320C64x Technical Overview.* Texas Instruments, 2000.

[3] *Cortex-A8 Technical Reference Manual.* ARM, 2007.

[4] *Realview Compilation Tools: NEON Vectorizing Compiler Guide.* ARM, 2007.

[5] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software*, pages 62–71, 2007.

[6] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. Cray X-MP: The Birth of a Supercomputer. *IEEE Computer*, 22(1):45–52.

[7] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, 30(2):65–98.

[8] H. Chang, J. Cho, and W. Sung. Performance Evaluation of an SIMD Architecture with a Multi-Bank Vector Memory Unit. In *Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation*, 2006.

[9] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[10] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. *SIGPLAN Notices*, 39(6):82–93.

[11] E. J. Fluhr and S. B. Levenstein. Method and Apparatus for Efficiently Accessing Both Aligned and Unaligned Data from a Memory. US Patent 7302525, 2007.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1):1–13, 2001.

[13] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):243–260, 2004.

[14] A. Kudriavtsev and P. Kogge. Generation of Permutations for SIMD Processors. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, Chicago, Illinois, USA. ACM.

[15] S. Larsen, E. Witchel, and S. P. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, 1997.

[17] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber. Vectorization Techniques for the Blue Gene/L Double FPU. *IBM Journal of Research and Development*, 49(2/3):437–446, 2005.

[18] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP Architecture. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, California, USA. ACM.

[19] D. Nuzman and A. Zaks. Autovectorization in GCC - Two Years Later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–58, 2006.

[20] N. C. Paver, B. C. Aldrich, and M. H. Khan. Intel Wireless MMX Technology: A 64-Bit SIMD Architecture for Mobile Multimedia. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 2003.

[21] G. Ren, P. Wu, and D. Padua. Optimizing Data Permutations for SIMD Devices. *SIGPLAN Notices*, 41(6):118–131.

[22] D. Talla, L. K. John, and D. Burger. Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003.

[23] S. Udayakumaran and R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM.

[24] Z. Wang. Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32(4):803–816, 1984.

[25] P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization.*

[26] K. X. Zhang. Buffer for a Split Cache Line Access. US Patent 6862225, 2005.