Pack Instruction Generation for Media Processors Using Multi-valued Decision Diagram

Tanaka Hiroaki, Yoshinori Takeuchi, Keishi Sakanushi, Masaharu Imai
Graduate School of Information Science and Technology,
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{h-tanaka, takeuchi, sakanusi, imai}@ist.osaka-u.ac.jp

Yutaka Ota, Nobu Matsumoto, Masaki Nakagawa
Center for Semiconductor Research and Development,
Semiconductor Company, Toshiba Corporation
580-1 Horikawa-Cho, Saiwai-Ku, Kawasaki, 212-8520, Japan
{yutaka2.oota, nobu.matsumoto, masaki.nakagawa}@toshiba.co.jp

ABSTRACT

SIMD instructions are often implemented in modern multimedia oriented processors. Although SIMD instructions are useful for many digital signal processing applications, most compilers do not exploit SIMD instructions. The difficulty in the utilization of SIMD instructions stems from data parallelism in registers. In assembly code generation, the positions of data in registers must be noted. A technique of generating *pack* instructions which pack or reorder data in registers is essential for exploitation of SIMD instructions. This paper presents a code generation technique for SIMD instructions with pack instructions. SIMD instructions are generated by finding and grouping the same operations in programs. After the SIMD instruction generation, pack instructions are generated. In the pack instruction generation, Multi-valued Decision Diagram (MDD) is introduced to represent and to manipulate sets of packed data. Experimental results show that our code generation technique can generate assembly code with SIMD and pack instructions performing complex repacking of 8 packed data in registers for a commercial VLIW processor with 6 pack instructions and achieved speedup ratio of up to 7.7.

Categories and Subject Descriptors

D.3.4 [**Programming Language**]: processors—*compilers*

General Terms

Performance, Algorithm

Keywords

SIMD Instructions, Multi-valued Decision Diagram

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

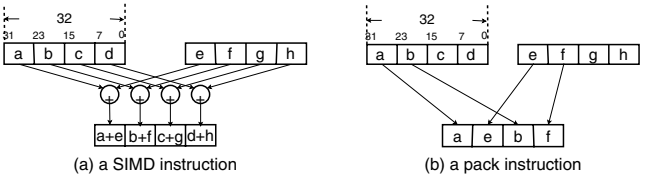

Figure 1: SIMD and pack instructions

1. INTRODUCTION

For systems for multimedia applications such as image processing and speech process, there is a great need for low-power processing offering high cost-performance. Recent micro processors are often customized to execute multimedia applications efficiently. The good nature in most of multimedia applications is data parallelism in applications. Therefore, many processors adopt SIMD (Single Instruction Multiple Data) instructions to exploit data parallelism. SIMD instructions perform operations using two source registers, and each register includes multiple data as shown in Fig.1(a). When a SIMD instruction is executed, the same operations are executed at the same time. Obviously, the processing efficiency of SIMD instructions is higher than that of conventional instructions which perform one operation at a time. Moreover, no special hardware is required to implement SIMD instructions. The identical functional units are embedded according to the number of operations in a SIMD instruction.

SIMD instructions are useful, but most compilers have limited ability to exploit SIMD instructions. In view of this limitation, in order to exploit SIMD instructions, programmers need to use compiler intrinsics, special functions in high level programming languages, which are mapped to specific instructions, or to write software in assembly languages. However, using compiler intrinsics or writing assembly programs is time-consuming tasks, and portability of programs is low. Therefore, a technique for automatically generating assembly programs including SIMD instructions is required.

The difficulty of code generation that exploits SIMD instructions stems from the data parallelism in registers. When using SIMD instructions, the positions of data in registers

must be noted. When a SIMD instruction which operates a binary operator is executed, operands of each operation performed by an identical SIMD instruction in registers must be in the same positions. If data and operations on the target application are well coordinated, SIMD instructions can be generated easily. If not, generation of additional *pack* instructions which reorder data in a register or repack data in different registers into one register are needed. Fig.1(b) shows a typical pack instruction which takes two elements from each source register and packs into the target register. Although such data repacking instructions take run-time execution cycles, the total execution cycles can decrease by the effect of SIMD instructions. Especially, the combination of SIMD and data repacking instructions with high level of parallelism emerging recent years can achieve high performance improvements compared with the case that SIMD instructions are unused.

There are many problems around code generation with SIMD instructions. One of the most essential topic is finding pack instruction sequence which generates required packed data from given packed data with given pack instructions. Almost processors have several pack instructions, but not all pack instructions. In addition, the number of all combination of data repacking is very large. Therefore, the pack instruction sequence which generates required packed data is not always found easily because of limitation of available pack instructions and large search space of data repacking. This is one of the most significant problem in the exploitation of SIMD instructions.

In this paper, a code generation technique for SIMD instructions including *pack* instructions is presented. This paper focuses on generation of pack instructions. In the generation of pack instructions, Multi-valued Decision Diagram(MDD) is utilized. Using MDDs, the sets of packed data are efficiently represented and manipulated. Exploiting the feature of MDDs, efficient and universal pack instruction generation algorithm is provided. As a result, high quality of code with SIMD and pack instructions can be obtained in reasonable compilation time.

The rest of this paper is organized as follows: Related works are summarized in section 2. The way to find SIMD operations in high level language program is described in section 3. Pack instruction generation based on MDDs is presented in section 4. Experimental results are shown in section 5. This paper is concluded in section 6.

2. RELATED WORK

Many publications have been released about automatic code generation of SIMD instructions[4, 5, 6, 7, 8].

In [8], pattern matching and covering problem with SIMD instructions are formulated to Integer Linear Programming (ILP). Solving the covering problem using ILP solver, highly optimized assembly codes with SIMD instructions are obtained. However, this approach takes too much time to solve ILP problems. For the latest SIMD instructions which handle 8 or 16 packed data, the time required to solve ILP problems may not be acceptable. Moreover, reorder or repacking data in registers is not handled in this method. In [7], SIMD instructions are generated by grouping statements in a basic block. Using data dependency and alignments information, statements executable in parallel are grouped into *Pack Set* to minimize data packing cost. Performance improvements are larger than traditional vectorization. However, the way to generate pack instructions and the related problem of packing are not mentioned. In [6], generation of SIMD and *permutation* instructions is presented. SIMD instructions are generated by grouping operations in basic block represented by Data Flow Graph(DFG). After the grouping, pack instructions are inserted between SIMD instructions. In this approach, in order to generate *permutations* which means packed data ordering in registers, permutation decomposition backward tree and forward tree are used. The backward tree is constructed from output permutation as root computing input permutations for any pack instructions. On the other hand, forward tree is constructed from an input permutation computing output permutations for any pack instructions. By matching the leaves of backward and forward trees, computation steps of output permutations are determined. Experiments shows significant performance improvements in some applications which require permutation reordering. All the approaches above target basic blocks. On the other hand, there are approaches targeting loops. In [4], using several analyses and loop transformations, loops are vectorized to generate SIMD instructions. Though this approach aims at exploitation of SIMD instructions, reordering packed data is not mentioned. [5] presents a vectorization technique in the presence of misaligned memory access in loop bodies. To exploit SIMD instructions, instructions which align elements in packed data registers are inserted.

Our approach is also a basic block approach and SIMD instructions are heuristically generated. Initial DFG is constructed by the same way presented in [6]. SIMD instruction generation is also performed by grouping operations and ordering operations in each grouped operations. The grouping and ordering operations are simpler approaches than [6], however, our approaches are useful for SIMD instruction exploitation. A *pack* instruction (called *permutation* instruction in [6]) generation technique based on MDD is originally presented in this paper. Though loop level approaches are also another approach, we consider that utilization of pack instructions is essential for exploitation of SIMD instructions. Therefore loop level algorithms are not mentioned in this paper.

3. GENERATION OF SIMD INSTRUCTIONS

Our code generation approach mainly consists of two parts. The first part is SIMD instruction generation. The second part is pack instruction generation. The first part is similar to [6]. Using tree matching [2] and [8], a data flow graph (DFG) whose nodes are elements of SIMD operations is constructed. After the DFG construction, the DFG is divided into data flow trees (DFT), then operations are grouped into SIMD instructions. Finally, a DFG whose nodes are SIMD instructions is constructed. In this section, grouping of SIMD operations is explained. Details of pattern matching, DFG construction and DFT construction are omitted because they are almost the same as in [6].

3.1 Grouping SIMD Operations

Groups of operations performed by SIMD instructions are determined as follows. First, leaves of DFTs which have the same operations are selected. Then, if the number of the selected nodes is less than the number of operations that one SIMD instruction can perform, the selected nodes are

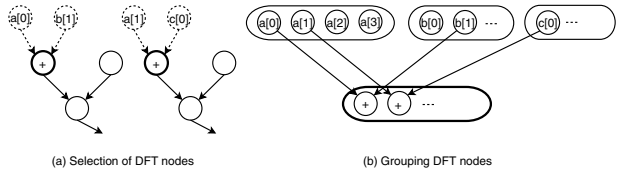

Figure 2: Operation grouping

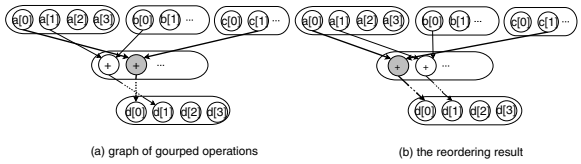

Figure 3: Operation ordering

grouped as a SIMD instruction. If not, the selected nodes are divided into smaller groups whose number of elements is less than the number of operations of a SIMD instruction. Finally, the nodes in the selected group are removed from the DFTs. This grouping is repeatedly processed until all nodes are removed from DFTs. In this process, for load and store operations, nodes that can be executable by one SIMD instruction is restricted by its memory address. Since misaligned memory access is unavailable or cause large penalty cycles, consecutive and aligned memory access operations are grouped as SIMD instructions. Fig.2 shows an example of operation grouping. The load operations have been removed from DFTs as shown in Fig.2(a). The add operations, nodes with a plus operator, are grouped and added to the DFT in Fig.2(b). Then, the add nodes will be removed from Fig.2(a). After this grouping, the DFT in Fig2(b) is constructed. This process continues until all nodes are removed from DFTs.

3.2 Ordering SIMD Operations in Registers

The order of operations in a register is determined as follows. The load and store operations are uniquely ordered according to the memory address accessed by operations, because the available group of memory access operation is limited by the memory address and alignment as mentioned in section3.1. The order of operations except for load and store is determined according to the order of load and store operations. The most frequently used position where sources and destinations are arranged is selected for each operation. Fig.3 shows the example of operation ordering. In Fig.3(a), a part of grouped graph is shown. The most left add operation has two sources `a[0]` and `b[1]`, one destination `d[1]`. The order in the grouped node of `a[0]` is the first element, `b[1]` is the second and `d[1]` is the second. Therefore, the most left add operation in the left hand side of Fig.3 is reordered to the second in the grouped node as shown in the right hand side of Fig.3. Similarly the second add operation is reordered to the most left in the grouped node.

4. GENERATION OF PACK INSTRUCTIONS

This section describes how the pack instructions are generated. After the grouping and ordering operations described in the section3, pack instructions which arrange the data in registers are generated to enable execution of SIMD instructions. Hereafter, the contents of registers are called *permutation* because they are naturally represented by permutations. The generation method consists of two steps. In the first step, it is examined whether the required permutation can be generated using given pack instructions. The basic concept of the first step is to generate all permutations from source permutations using available pack instructions. In the second step, the expression tree representing the generation of the required permutation from input permutations is built. The tree construction starts from the root, which is the node corresponding to the required permutations, and subtrees are built recursively by the tree construction procedure. In the packing instruction generation, Multi-valued Decision Diagram(MDD)[3] is utilized to represent and manipulate the sets of permutations. Using MDD, a pack operation can be manipulated not on a pair of permutations but on a pair of sets of permutations. This characteristic enables efficient generation of pack instructions. In the rest of this section, MDDs are introduced first. Then the pack instruction generation algorithm is described.

4.1 Introduction of MDDs for Representation of a Set of Permutations

In this section, representation of the set of permutations using Multi-valued Decision Diagram(MDD) is introduced.

Consider the register in which has n elements of packed data. Let $S = \{s_1, s_2, \dots, \}$ be the set of given sub-word data. Let $r = (r(1), \dots, r(n))$, $r(j) \in S$ for $j = 1, \dots, n$, be the permutation representing the content of a register. Let $R = \{r_1, r_2, \dots\}$ be the set of permutations. When a set of permutations R and a permutation r are given, a function $F_R : S^n \rightarrow \{0, 1\}$ is defined as follows :

$$F_R(r) = \begin{cases} 1, & \text{if } r \in R \\ 0, & \text{if } r \notin R \end{cases} \quad (1)$$

According to the definition, the function F_R implicitly represents the set of permutations R.

Here, discrete variables $x_1, .., x_n$ are introduced whose domain is S. Assume x_i to be the ith element of r which is the input of F_R. Using x_i, the equation(1) can be expressed as follows :

$$F_R(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \text{if } (x_1, \dots, x_n) \in R \\ 0, & \text{if } (x_1, \dots, x_n) \notin R \end{cases} \quad (2)$$

In the equation(2), F_R is defined as the multi-valued input, binary-valued output function. Such functions can be represented by Multi-valued Decision Diagram. Fig.4 shows two MDDs for {`abcd`} and {`abcd`,`abdc`}. In Fig.4(a), the only one path exists from the root to 1-terminal through the edges `a`,`b`, `c`,`d`. On the other hand, in Fig.4(b), there are two paths exist from the root to 1-terminal through `a`, `b`,`c`,`d` and `a`,`b`,`d`,`c`. Considering the sequences of the labeled symbols on edges as elements in a set, a set of permutations can be represented by a MDD. Moreover, some MDD manipulations correspond to operations on the sets of permutations. The logical-or operation on MDD corresponds to the union operation on the set of permutations. Similarly, logical-and operation on MDD corresponds to the intersection operation. For example, the MDD shown in Fig.4(b) is constructed by the logical-or of MDDs representing {`abcd`} and {`abdc`}, which corresponds to the union of {`abcd`} and {`abdc`}.

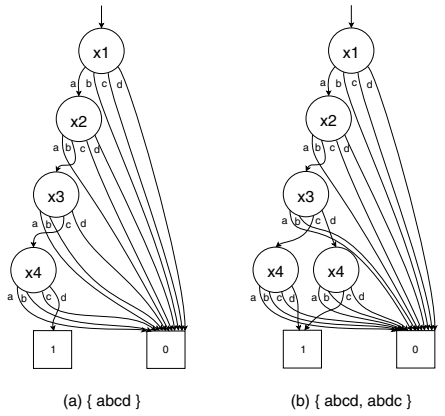

Figure 4: MDDs for { abcd } and { abcd,abdc }

4.2 Pack Operation Manipulation on MDDs

Using MDDs, basic operations such as union and intersection can be applied to permutations. Similar to such basic operations, pack operations can also be performed on MDDs.

Consider a pack operation p which takes two permutations r_1, r_2, and returns a permutation r_3. Let $\sigma(k)$ be a function defined by $\sigma(k) = (a_k, b_k)$, $a_k \in \{1, 2\}$, $b_k \in \{1, \ldots, n\}$ for $k = 1, \ldots, n$. Let $\sigma'(k)$ and $\sigma''(k)$ be the first and the second value of $\sigma(k)$ respectively. Let $q_{(i,j)}$ be the jth element of the ith input permutation of p. Given a function σ, a pack operation $p_\sigma(r_1, r_2)$ is defined as follows :

$$\begin{aligned} p_\sigma(r_1, r_2) &= (q_{\sigma(1)}, \ldots, q_{\sigma(n)}) \\ &= (r_{\sigma'(1)}(\sigma''(1)), \ldots, r_{\sigma'(n)}(\sigma''(n))) \end{aligned} \quad (3)$$

Let P_σ be the pack operation on sets of permutations R_1 and R_2. $P_\sigma(R_1, R_2)$ is defined as follows :

$$P_\sigma(R_1, R_2) = \bigcup_{(r_1, r_2): r_1 \in R_1, r_2 \in R_2} \{\ p_\sigma(r_1, r_2)\ \} \quad (4)$$

The result of $P_\sigma(R_1, R_2)$ is a set of permutations whose elements are results of p_σ on any pairs of elements of input sets.

The direct computation of equation(4) is hard when $|R_1|$ and $|R_2|$ are large. However, using MDDs, pack operations on sets of permutations are effectively manipulated. In other words, there is no need to execute the pack operation for each pair. The way to compute $P_\sigma(R_1, R_2)$ using MDDs consists of three primitive manipulations.

1. For each R_i, make R_i' from R_i by adding any permutations whose elements to be used by the pack operation are the same as one of the permutations in R_i.

   This computation on MDDs is simply implemented. Every node which corresponds to unused element is replaced with the union of its children as shown in Fig.5.

2. For each R_i', make R_i'' by reordering the elements of all permutations in R_i' to match with the order of the output of the pack operation.

   This computation on MDDs is almost same as the conventional variable ordering technique for decision diagrams. The difference between this reordering and conventional variable ordering is that the level of variable is not changed in this reordering whereas it is changed in the conventional variable ordering. Fig.6 shows reordering of elements on MDDs.

3. Finally, $P_\sigma(R_1, R_2)$ is obtained by computing intersection of R_1'' and R_2''. The intersection operation corresponds to the logical-and operation on MDDs.

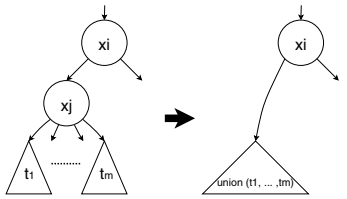

Figure 5: Adding permutations on MDDs

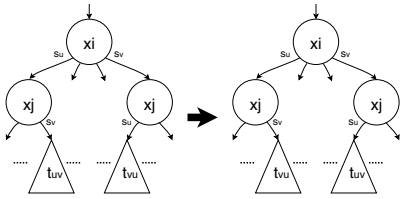

Figure 6: Reordering on MDDs

For the explanation of the pack operation manipulation, consider a pack operation p_σ shown in Fig.7. The couples of integers in the output register mean $\sigma(k)$. As shown in Fig.7, σ of the pack instruction is $\sigma(1) = (1, 1)$, $\sigma(2) = (2, 1)$, $\sigma(3) = (1, 2)$, $\sigma(4) = (2, 2)$. Assume the input sets of permutations are $R_1 = \{\texttt{abcd}\}$ and $R_2 = \{\texttt{dcba}\}$. The elements of R_1' are all permutations matching `ab**`. As a result, R_1' is obtained as follows :

$$\begin{aligned} R_1' = \{&\texttt{abaa}, \texttt{abba}, \texttt{abca}, \texttt{abda}, \texttt{abab}, \texttt{abbb}, \texttt{abcb}, \texttt{abdb}, \\ &\texttt{abac}, \texttt{abbc}, \texttt{abcc}, \texttt{abdc}, \texttt{abad}, \texttt{abbd}, \texttt{abcd}, \texttt{abdd}\} \end{aligned}$$

Similarly, the elements of R_2' are all permutations matching `dc**`. In the second step of the pack operation manipulation, elements in R_1' and R_2' are reordered according to the pack operation. The elements of R_1'' and R_2'' are all permutations matching `a*b*` and `*d*c` respectively. Finally, the intersection of R_1 and R_2 is computed. The result of the intersection is the set of permutations matching both `a*b*` and `*d*c`. As a result, `{ adbc }` is obtained for this example.

In the example shown in this section, the length of permutation is 4, and the input sets of permutations have only one element. However, it is clear that these are not restrictions, because such parameters are independent of those manipulations.

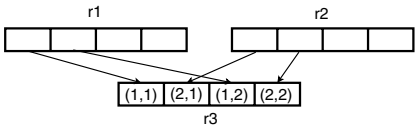

Figure 7: An example pack instruction

CanGeneratePermutation($R_0, \mathbf{P}, r_o$)

1: $i \leftarrow 0$
2: **while** $r_o \notin R_i$ **do**
3: $\quad \forall P_j \in \mathbf{P} : R_{i+1,j} \leftarrow P_j(R_i, R_i)$
4: $\quad R_{i+1} \leftarrow (\bigcup_j R_{i+1,j}) \cup R_i$
5: $\quad$ **if** $R_{i+1} = R_i$ **then**
6: $\quad\quad$ **return** false
7: $\quad$ **end if**
8: $\quad i \leftarrow i + 1$
9: **end while**
10: $n_{depth} \leftarrow i$
11: **return** true

Figure 8: Testing Target Permutation Generation

4.3 Pack Instruction Generation Algorithm

In this section, the pack instruction generation algorithm is explained.

The inputs of the algorithm are the set of permutations R_0, a required permutation r_o and a set of pack operations $\mathbf{P}$. The output is the expression tree whose operations are $P \in \mathbf{P}$ and leaves are $r \in R_0$. The result of evaluation of the tree have to be r_o. Note that the subscript of R is used to distinguish among the variants of the set of permutations generated in this algorithm though R_i means the ith input of the pack operation in the section3.

The pack instruction generation algorithm consists of two subprocedures.

1. Examine whether the required permutation can be generated using the given pack operations.
2. Build the expression tree whose intermediate nodes are the pack operations, leaves are the input permutations.

The first subprocedure *CanGeneratePermutation* is shown in Fig.8. The basic concept of *CanGeneratePermutation* is to generate all permutations from source permutations using available pack operations until r_o is generated. The main process is the while loop in the lines 2-9. The variable i is initialized to 0 and incremented for every iteration. R_i holds all permutations generated in $0, \ldots, i$ th iterations. In the lines 3-4, R_{i+1} is made from R_i by adding permutations generated by available pack operations. In the line 5, R_{i+1} and R_i are compared. If R_{i+1} is equal to R_i, this subprocedure will finish and return "false" since it means that no more permutations can be generated and the required permutation could not generated by available pack operations. Until the required permutation is generated or no other permutations can be generated, the while loop is executed repeatedly. When this subprocedure finished, the number of iterations is obtained as a constant n_{depth}. The constant n_{depth} and the sets of permutations $R_1, \ldots, R_{n_{depth}}$ generated in this subprocedure are also used in the second subprocedure.

The second subprocedure *GetExpressionTree* is shown in Fig.9. The inputs are a set of permutations $R^{require}$ and an integer i. An expression tree representing the expression to generate one of the elements in $R^{require}$ is returned. The second input i indicates the depth of tree to be built. The depth of obtained tree will be less than equal to i. This subprocedure constructs a expression tree recursively. At the start, *GetExpressionTree* is invoked with $i = n_{depth}$ and $R^{require} = \{r_o\}$. In Fig.9, the lines 1-3, a leaf of permutation in $R^{require} \cap R_0$ is returned if $R^{require}$ includes any

GetExpressionTree($R^{require}, i$)

1: **if** $R^{require} \cap R_0 \neq \phi$ **then**
2: $\quad$ **return** a leaf corresponds to $r \in R^{require} \cap R_0$
3: **end if**
4: **if** $R^{require} \cap R_i = \phi$ **then**
5: $\quad$ **return** *nil*
6: **end if**
7: **for all** $P_j \in \mathbf{P}$ **do**
8: $\quad (R_j^{require,l}, R_j^{require,r}) \leftarrow P_j^{-1}(R^{require})$
9: $\quad T_j^l \leftarrow$ *GetExpressionTree*$(i-1, R_j^{require,l})$
10: $\quad T_j^r \leftarrow$ *GetExpressionTree*$(i-1, R_j^{require,r})$
11: $\quad$ **if** $T_j^l \neq nil$ and $T_j^r \neq nil$ **then**
12: $\quad\quad T_j \leftarrow$ a tree with P_j as root, subtrees are T_j^l and T_j^r
13: $\quad$ **else**
14: $\quad\quad T_j \leftarrow nil$
15: $\quad$ **end if**
16: **end for**
17: **if** $\forall T_j : T_j \neq nil$ **then**
18: $\quad$ **return** T_j such that the number of nodes is minimal
19: **else**
20: $\quad$ **return** *nil*
21: **end if**

Figure 9: Expression Tree Construction

permutations in R_0. In the lines 4-6, *nil* is returned if the condition is satisfied since the condition indicates that no required permutation is in R_i. In the lines 7-15, for each pack operation P_j, a tree whose root is P_j is constructed. P_k^{-1} is the inverse pack operation. In the line 8, P_k^{-1} returns a pair of sets of permutations ($R_{i,j}^{require}$,$R_{i,j}^{require}$) which is the source of $R^{require}$. In the lines 9-10, *GetExpressionTree* is recursively invoked to build the left and right subtrees, T_k^l and T_k^r. In the lines 11-15, If both T_k^l and T_k^r are not *nil*, a tree T_k whose root is P_k, and the subtrees are T_k^l and T_k^r is built. Finally, in the lines 17-21, T_k which has minimal cost is returned. If any T_k is *nil*, *nil* is returned.

In *GetExpressionTree*, R_i generated in *CanGeneratePermutation* is used to prune redundant subtree construction. This gives greate reduction of computation time to search a desired expression tree. In *CanGeneratePermutation*, on the other hand, pack operations are performed $n_{depth} \cdot |\mathbf{P}|$ times. However, it is reasonable since it is polynomial in both the number of pack operations and the depth of the tree.

5. EXPERIMENTAL RESULTS

To confirm the effectiveness of SIMD and pack instruction generation algorithm, the algorithm was implemented. Media embedded Processor, MeP[1, 9] was used as the target processor. MeP supports several extension options, embedding user designed logics, adding coprocessors, etc. The extension option used in this experiments was coprocessor option. The additional coprocessor has a 64 bit register file, and supports 8-parallel byte, 4-parallel halfword, and 2-parallel word SIMD instructions. The additional coprocessor works with MeP core in parallel and behaves as a 3-way VLIW processor. For this experiments, following 3 programs, (1)binary threshold, (2)reversed reordering and

Table 1: The number of insns. and compilation time

program	no-SIMD # of insns. of kernel	SIMD # of insns. of kernel	compilation time of SIMD [sec]
binary threshold	64	8	0.42
reversed reordering	20	11	390
color conversion	72	33	1020

Table 2: Execution cycles

program	no-SIMD [cycles]	SIMD [cycles]	speedup ratio
binary threshold	716082	92228	7.76
reversed reordering	332064	110809	3.00
color conversion	221338	90459	2.44

(3)color conversion were coded. The binary threshold generates the binary vector from an integer vector. The reversed reordering takes a vector and outputs the reversed ordered input vector. The color conversion takes an input vector composed of three colors, red, green and blue, and computes an output vector whose elements are the results of $a[3*i] + (a[3*i+1] >> 1) + (a[3*i+2] >> 2)$. These programs were simple for a limited implementation of the algorithm, but suitable to confirm the ability to exploit SIMD and pack instructions. The size of processing data was 8 bits for all evaluated programs. The algorithm was implemented as the converter from plain C program to C program using intrinsic functions of coprocessor instructions. Converted programs were compiled by MeP C compiler, and then simulated by the instruction-set simulator (ISS). All processes in this experiments were executed with RedHat 7.1 on Pentium III 600 MHz. Table 1 shows the number of instructions of the kernel in compiled code and compilation time of the SIMD and pack instruction generation algorithm. Table 2 shows the execution cycles counted by ISS.

In the case of the binary threshold, speedup ratio of up to 7.7 was achieved. Because the binary threshold could be completely vectorizable, no pack instructions were needed.

In the case of the reversed reordering, speedup ratio of up to 3.0 was achieved. High performance could be achieved by not only the code reduction by SIMD instructions utilization, but also by instruction level parallelism from the core and coprocessors. Fig. 10 shows the packing performed in this program. In the kernel of the reversed reordering, a load instruction, a store instruction and 7 pack instructions were generated. Many pack instructions were generated and the total number of the instructions was reduced.

In case of the color conversion, speedup ratio of up to 2.5 was achieved. In this case, 3 load instructions, 3 store instructions, 4 SIMD arithmetic instructions and 24 pack instructions were generated. Fig. 11 shows the packing performed in this program. As shown in Fig.11, packing was very complex. Whereas it took 1020 seconds to compile, the algorithm could generate pack instructions and the number of instructions generated was half in the case of without SIMD and pack instructions.

Figure 10: Packing in the reversed reordering

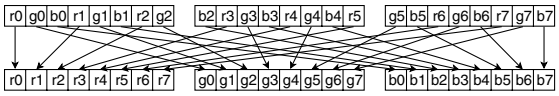

Figure 11: Packing in the color conversion

6. CONCLUSIONS

In this paper, a code generation technique for SIMD and pack instructions are presented. Utilization of pack instructions is essential for exploitation of SIMD instructions. In the presented algorithm, the packed data in registers are represented and manipulated by MDDs. Utilizing MDDs, pack instructions can be generated. The experimental results shows the pack instruction generation algorithm can generate SIMD and pack instructions and reduce code size even though the data repacking is very complex.

7. REFERENCES

[1] *Media embedded Processor, http://www.mepcore.com.* 2005.

[2] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491 – 516, October 1989.

[3] S. M. Arvind Srinivasan Timothy Kam and R. K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[4] 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, April 2002.

[5] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82 – 93, June 2004.

[6] 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*, pages 147 – 156, June 2005.

[7] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[8] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[9] T. Miyamori, J. Tanabe, Y. Taniguchi, K. Furukawa, T. Kozakaya, H. Nakai, Y. Miyamoto, K. Maeda, and M. Matsui. Development of image recognition processor based on configurable processor. *Journal of Robotics and Mechatronics*, 17(4):437–446, 2005.