Generation of Permutations for SIMD Processors

Alexei Kudriavtsev
University of Notre Dame
Notre Dame, IN 46556
akoudria@cse.nd.edu

Peter Kogge
University of Notre Dame
Notre Dame, IN 46556
kogge@cse.nd.edu

ABSTRACT

Short vector (SIMD) instructions are useful in signal processing, multimedia, and scientific applications. They offer higher performance, lower energy consumption, and better resource utilization. However, compilers still do not have good support for SIMD instructions, and often the code has to be written manually in assembly language or using compiler builtin functions. Also, in some applications, higher parallelism could be achieved if compilers inserted permutation instructions that reorder the data in registers. In this paper we describe how we create SIMD instructions from regular code, and determine ordering of individual operations in the SIMD instructions to minimize the number of permutation instructions. Individual memory operations are grouped into SIMD operations based on their effective addresses. The SIMD data flow graph is then constructed by following data dependences from SIMD memory operations. Then, the orderings of operations are propagated from SIMD memory operations into the graph. We also describe our approach to compute decomposition of a given permutation into the permutation instructions of the target architecture. Experiments with our prototype compiler show that this approach scales well with the number of operations in SIMD instructions (SIMD width) and can be used to compile a number of important kernels, achieving up to 35% speedup.

1. INTRODUCTION

As more transistors become available to conventional microprocessor hardware designers, a larger share of the transistors is devoted to discovering parallelism in sequential code, rather than to actual computation. On the other hand, virtually all processors today need to run multimedia applications that offer easy to use parallelism. However, to use this kind of parallelism efficiently, a special class of parallel functional units is employed. A number of identical or similar operations (**s**ingle **i**nstruction) are performed in parallel on consecutive data (**m**ultiple **d**ata). These individual units of data are kept in large SIMD registers spanning more than a hundred of bits. Register-level SIMD is a relatively inexpensive form of instruction level parallelism with low hardware overhead. The processor has to fetch and issue only one SIMD instruction to execute all operations in that instruction. The register file design is simpler, since all the operations in one SIMD instruction use the same register file ports. This results in more regular and energy-efficient hardware. Today, such SIMD extensions exist in virtually every processor on the market. Examples are in DSP processors (StarCore, TI's c60x, Trimedia, etc.), desktop and laptop processors (Pentium, Athlon, PowerPC), and high end server processors (Itanium, UltraSPARC).

To take full advantage of the processor's SIMD capabilities, many applications have had to be written by hand either in assembly language or in C, using compiler-known functions. Each such function directly corresponds to a specific SIMD instruction. By inserting such functions in a program, the programmer instructs the compiler which instructions need to be selected, and the compiler finishes the job by performing register allocation and code scheduling.

Currently, some compilers can automatically generate SIMD instructions in some cases. In a loop level vectorizer, for example, a loop is unrolled as many times as the number of operations in SIMD instructions. Thus, the code of the body of the loop is replicated many times, and each replica is placed into a slot in a SIMD instruction. But such compilers cannot handle the code with more complicated data dependences, such as in bit-reversal data reordering used in FFT. In general, modern compilers cannot efficiently reorder data for SIMD instructions. Here, the process of such data reordering is called a *permutation*, and it is assumed that specification of such reordering is a compiler-visible feature of the target instruction set architecture. Note that this kind of reordering is not strictly a permutation in the mathematical sense, because elements of the source vector may be replicated or omitted, and the size of the result may be different from the size of the source.

There is a relationship between generation of SIMD instructions and generation of permutations. If the data dependences are not straightforward, it is not always obvious which operations should be packed into SIMD instructions. Depending on how the operations are grouped and ordered within the groups, different permutation code needs to be produced. This, of course, affects the performance of the generated code.

If the SIMD instructions are only generated at the loop level (loop vectorization), SIMD code cannot be produced for linear code without loops. While SIMD instructions are extremely useful for improving loop performance, they can and should also be generated at basic block level (linear code) too. Some approaches to overcome this limitation are mentioned in the next section.

This paper presents a technique for generation of SIMD code for wide SIMD units (more than 4 independent operations in one SIMD instruction), including permutation code, from regular C programs, without use of intrinsics or compiler-known builtin functions. Also, presented in this paper is an approach to decompose a

Submitted to LCTES'05 Chicago, Illinois, June 15–17 2005

given permutation into permutation instructions available in the instruction set of the target architecture.

The paper is organized as follows. In the next section, an overview of existing compilers and other approaches to generating SIMD instructions is given. The general problem of generating SIMD instructions and our algorithm are described in section 3. Our approach to generating efficient permutations is explained in section 4. An interesting abstract problem of optimal permutation decomposition and our solution in the context of this work are presented in section 5. After that, some experimental results are given in section 6, and the paper is concluded in section 7.

2. RELATED WORK

When a compiler cannot efficiently use some special instructions, the programmer may still be able to use those instructions in a program written in a high level language by means of special functions that directly correspond to those instructions. Application development with such compiler-known builtin functions is easier than coding in assembly language because such tasks as code scheduling and register allocation are performed by the compiler. The programmer can write programs in C that are just as efficient as those written in assembly, but since different platforms offer different sets of builtin functions, such programs are not portable.

It is possible to define a set of common SIMD operations and automatically translate the applications written in this high level SIMD language to platform-specific C code, achieving code portability, as it was done with SWARC [3]. The drawback of this approach is that it effectively introduces a new C-like programming language, requiring the applications to be rewritten to achieve portable usage of SIMD capabilities of the target processors.

As an example of a compiler that makes use of builtin functions, the Intel C++ compiler [4] provides a set of intrinsics for MMX and SSE SIMD support, C++ SIMD vector classes, and a loop vectorizer that extracts SIMD parallelism from code automatically. For the automatic vectorization to work, code must adhere to a set of guidelines and sometimes contain `#pragma` directives in the loops where vectorization is desirable.

Another vectorizing compiler was developed at IBM for the eLite DSP processor [10]. The compiler is based on Chameleon, IBM's VLIW research compiler. It adds a vectorizer phase that selects SIMD instructions for the eLite DSP. The processor has a vector element file, in which each element of a short vector can be arbitrarily selected from the register file using a vector element pointer. The eLite DSP has a flexible vector pointer unit that makes explicit permutations unnecessary since most data reorganization can be achieved by manipulating the vector element pointers instead of the elements. However, the architecture of the eLite processor is distinct from all other SIMD architectures, and the techniques developed for its compiler may not be directly applicable to more common SIMD architectures.

An approach that can generate SIMD instructions at the basic block level rather than loop level was suggested by Leupers [8, 7]. This code selection is based on data flow tree (DFT) parsing, in which special nonterminals are assigned to subregisters of SIMD register, such as `reg_lo` and `reg_hi`. For each DFT, the parser generates a set of alternative DFT covers, including SIMD covers and regular non-SIMD covers. These separate SIMD covers of subregisters are combined into complete SIMD instructions by using integer linear programming (ILP).

Leupers' approach is very capable of finding SIMD instructions in the code, because it essentially considers all possibilities and selects one with the best cost. However, it does not scale well with the width of SIMD and the size of basic block. In his work, Leupers describes code selection for SIMD with only 2 and 4 subregisters. This may be sufficient for current processors, but new SIMD architectures are bound to offer higher levels of parallelism. Also, with this approach it is difficult to generate permutations.

In [5], Larsen suggested an algorithm for grouping operations into SIMD instructions by first identifying *seeds* of adjacent memory operations and packing them together into small groups, and then merging the groups until they reach the size of SIMD instructions. In [6], he suggested a number of techniques to increase the number of memory operations that can be successfully grouped into SIMD instructions. This approach scales better with the SIMD width, but the problem of generating permutations is not addressed in his work.

Eichenberger et al. describe their approach to the problem of SIMD vectorization ("*simdization*") with data alignment constraints in [2]. Their algorithm can generate permutation instructions to align data for misaligned data references, but they again do not address the general problem of generating permutations.

3. WIDE SIMD INSTRUCTIONS

Similarly to Larsen [5], the formation of SIMD instructions in this work begins from memory operations, based on their effective address, but the approach to grouping other operations is quite different.

We assume that memory operations for SIMD instructions must access consecutive data, i.e. memory operations can be sorted by their effective addresses, and the address difference of any two adjacent memory operations in the sorted sequence must be equal to the size of the data they access. These memory operations are combined into *SIMD groups*. A SIMD group is a set of operations that can potentially be converted into a SIMD instruction of the target architecture. Once the memory operations are grouped, the data flow graph of the basic block can be traversed from the memory operations in the SIMD groups to form SIMD groups of other operations.

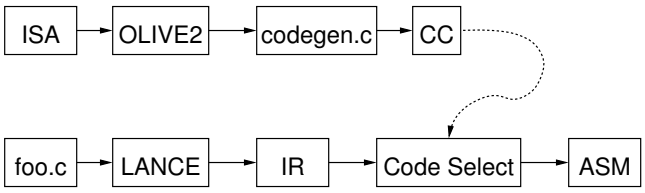

Figure 1: Code Selection with LANCE.

The compiler prototype developed in this work is based on the LANCE C frontend [7]. The structure of the code generator is shown in figure 1. The instruction set of the target architecture is represented as a set of rules. Each rule has a tree pattern that consists of terminals and nonterminals, and it also has cost part and action part. Terminals represent operations such as addition, multiplication, storing data to memory or reading data from memory. Nonterminals correspond to registers and variables. The cost part computes the cost of covering the data flow tree with this tree pattern, and the action part of the rule generates assembly code if the rule is selected. The cost part is very flexible. It is a piece of C/C++ code, and the cost value can be any data type such as `integer`, `double`, or an object. This flexibility allows code selection not only for minimal number of instructions, but also for energy consumption, etc.

From this tree grammar, the source code of the code generator is generated by the *OLIVE2* code generator generator from [8].

OLIVE2 is a modified version of the *OLIVE*, which is a derivative of the *twig* tool [1]. These tools use tree pattern matching and dynamic programming to compute an optimal tree cover in linear time.

The LANCE frontend analyzes a C program, converts it into intermediate representation (IR), performs a number of optimization on the IR and saves it. The optimized IR is then processed by the code generator. Code selection is performed at the level of a basic block. For each basic block of the program, a *data flow graph* (DFG) is constructed. If some nodes of DFG are read by more then one node, such nodes represent *common subexpressions* (CSEs), and the graph is not a tree, but the code generator works at the tree level, covering one DFT at a time. Thus, to make the graph suitable for OLIVE, it is split into *data flow trees* (DFTs) at the common subexpressions, so that each CSE becomes the root of a new DFT. The different operations within a SIMD instruction belong to different DFTs, therefore SIMD instructions cannot be discovered at the tree level. They are discovered in the next step, *simdization*, which is performed after a set of optimal covers is computed for each DFT of the DFG. The code generator attempts to combine covers of different trees into SIMD covers, thus further reducing the cost of the basic block cover.

Here is a general formulation of the code selection problem using SIMD instructions. Given a data flow graph of a basic block and costs of the target architecture instructions, compute a DFG cover that uses both SIMD and nonSIMD instructions and minimizes the total cost of the basic block.

Since memory address usually must be aligned, the memory operations (loads and stores) can be grouped into SIMD memory operations so that the resulting group accesses data at an aligned memory address, each operation accesses data of the same size, and the total size of accessed data is equal to the size of the SIMD register. The individual memory operations in a group are ordered by their effective address.

The effectiveness of using SIMD operations depends on the mapping of program objects to memory. The mapping can be optimized to improve performance of the SIMD code [6], but for this work the mapping is considered fixed.

Other operations in the basic block can be grouped arbitrarily, as long as the groups are *valid*. A partitioning of operations of a basic block into SIMD groups is valid if

- operations in each group are compatible with each other, i.e. the target ISA has an instruction that implements the operations of the whole group in one instruction,
- the operations in each group are independent, and
- the groups are schedulable, i.e. if some operation from group *A* depends on an operation from group *B*, no operation from group *B* can depend on an operation from group *A*. In other words, the DFG of the SIMD operations is a directed acyclic graph, DAG.

Depending on how the operations are grouped and ordered within the groups, it may be necessary to insert data reordering operations, "permutations" into the SIMD DFG. These operations add to the cost of the basic block and reduce the advantages of implementing the basic block with SIMD operations. It is possible that for some operation grouping and groups ordering assignment the resulting SIMD code will be less efficient than scalar code.

Thus, assigning operations to SIMD groups, determining the ordering of the operations, and inserting permutations into the SIMD DFG are related subproblems of the optimization problem. However, this problem as a whole is hard to solve, and thus it is divided into two subproblems. In the first part, the operations are assigned to SIMD groups. In the second, the ordering of the operations within SIMD groups is determined and the permutations are inserted as needed.

3.1 SIMD Groups

A SIMD group is a set of operations that can be combined into an assembly language SIMD instruction of the target ISA. The operations in a SIMD group must be performed on the same data type, be independent (not reachable from one another in the basic block's DFG), and, if memory operations, must have consecutive effective addresses accessing adjacent locations.

To specify tree patterns of the operations that can be combined into SIMD groups, *SIMD nonterminals* `reg8`, `reg16`, etc. are used in OLIVE rules. These SIMD nonterminals denote subregisters of a SIMD register. For example, the OLIVE rules for 16-bit and 32-bit additions performed on subregisters of a SIMD register are:

```
reg16: PLUS(reg16,reg16)
reg32: PLUS(reg32,reg32)
```

In these tree patterns, an addition operation reads two values from subregisters, and saves the result in another subregister. If the number of trees matching this pattern is sufficient to form a SIMD group, and all trees are selected to be covered by this pattern, then, in the simdization phase, such nonSIMD trees are merged into SIMD trees. The number of nonSIMD trees that can be merged into a SIMD tree is equal to the number of bits in the SIMD register divided by the number of bits in the SIMD subregister. For example, Intel SSE registers are 128-bit wide. To form a 32-bit SIMD addition, there must be four 32-bit addition trees. When the *action* corresponding to this pattern is invoked, it will generate an assembly instruction for SIMD addition, such as parallel addition of 16-bit values `paddw` or parallel addition of 32-bit values `paddd` in Intel SSE.

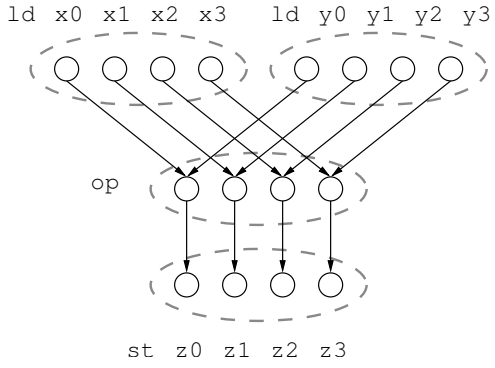

Figure 2: Grouping of SIMD operations

An operation is a *SIMD candidate* if it matches a pattern that produces a SIMD nonterminal or if it is a store operation and its data operand is a SIMD nonterminal. An example of how the DFTs can be combined into SIMD groups is shown in figure 2. In this example two vectors x and y are added to produce third vector z. Each element of the vectors is shown on the picture separately as x_i, y_i, and z_i. The operations that form a group are surrounded by a dotted oval. First, elements x_i and y_i are loaded from memory (`ld` groups), then they are added (`op` group) and the results are stored into z_i (`st` group). The memory operations that read x form one load SIMD group, and the memory operations that read y form the other load SIMD group. The memory operations that store z are combined

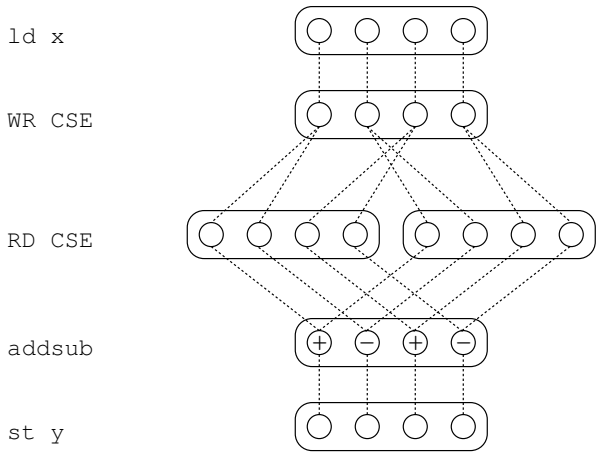

Figure 3: DFG with entangled SIMD dependences.

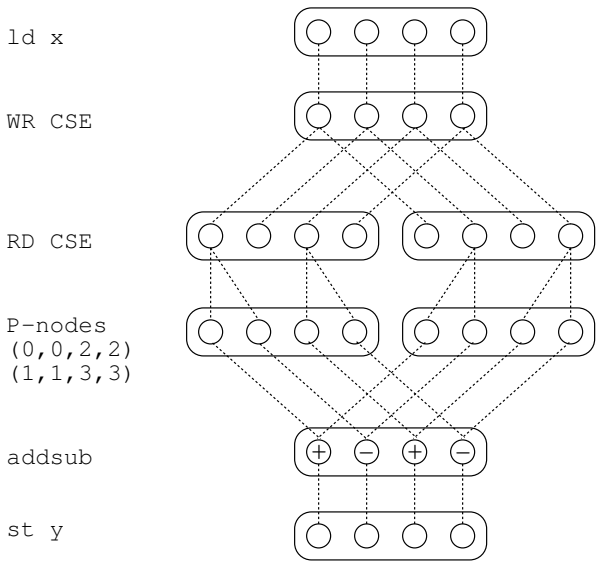

Figure 4: Resulting DFG with P-nodes.

into a store SIMD group. The remaining arithmetic operations are also combined into a SIMD group.

The following algorithm is used to form SIMD groups. To simplify grouping, all SIMD candidates are sorted by their operation type. First, the groups of memory operations are created. Memory operations with the same data type are sorted by their effective address. Then they are split into groups of memory operations that access adjacent locations, so that the total size of the data they access is equal to the size of SIMD register, and the address of the compound data they access is aligned. The memory operations are ordered within the groups by their effective addresses. This approach can only work if the compiler has sufficient information about effective addresses of memory operations.

To create groups of other operations, the first pass is made starting from the groups of store SIMD operations. This algorithm takes sets of children nodes of current SIMD group nodes, checks if they form a valid SIMD group, and continues recursively with newly formed children SIMD groups. The process continues until an existing SIMD group is reached, or a group of operations that read a common subexpression (`RD_CSE`) is formed.

The algorithm must stop at common subexpressions because they require additional analysis. A common subexpression means that the same subregister is read by multiple operations, and it may be necessary to insert permutations to satisfy orderings of all SIMD groups involved. Also, the common subexpression may be created in another basic block, in which case additional inter-basic block ordering analysis is required.

The second pass starts from the groups of load SIMD operations, and stops when existing groups are reached. This pass is different from the previous because the binary operations may be grouped differently, depending on which operand is used for group formation. For example, a group may be formed in such a way that it has its left operands coming from one SIMD group, but its right operands come from more than one SIMD group. This problem is not solved optimally, instead, a simple heuristic is used: the groups are formed based on their left dependences.

When all SIMD groups are identified, they are used to construct a SIMD DFG by considering data dependences of individual operations within the SIMD groups. The SIMD DFG is then analyzed for orderings of operations in the SIMD groups, and permutation nodes are inserted if necessary. This process is discussed in more detail in the following sections.

3.2 SIMD DFG

Each SIMD group of operations is represented by a single node in the SIMD DFG. The SIMD DFG is inserted into the DFG of the basic block and replaces the nodes that were combined into the SIMD groups. If the constructed SIMD DFG contains common subexpressions, it is not a tree, and to make it amenable to OLIVE-based tools it is split into DFTs at the common subexpressions. Then, the SIMD DFTs are parsed by the code generator produced by OLIVE, and the SIMD assembly code is generated.

Some SIMD instructions may perform operations that do not directly correspond to scalar instructions. For such operations, no existing type of DFG node can be used, and new DFG node types are needed. Examples of such DFG node types are permutations and mixed add/subtract operations that are common in DSP processors. If a SIMD instruction performs operations of different types, not all orderings of the operations can be valid. For example, the SIMD add/subtract instruction may require the additions to be in even positions and the subtractions in odd positions.

It may happen that a SIMD group is not using all results of operations from another SIMD group, replicating them, or taking results of operations from more than one SIMD group. In this case, abstract permutation nodes, *P-nodes*, are inserted into the DFG. These P-nodes represent locations where SIMD permutation instruction(s) will definitely have to be inserted, but the specific instructions will be determined later. The P-nodes represent irregularities in the data flow from one SIMD register to another. Consider the following C code:

```
y[0] = x[0] + x[1];
y[1] = x[0] - x[1];
y[2] = x[2] + x[3];
y[3] = x[2] - x[3];
```

In this code, the dependences do not flow straight from source SIMD registers to operations. Different trees use the same values and the dependences get entangled. The DFG of this basic block is shown in figure 3. In this figure, the values loaded from memory (`ld x` group) become common subexpressions (CSEs). The operations that create CSEs are denoted `WR CSE`, and the operations that read CSEs are denoted `RD CSE`. These operations do not produce

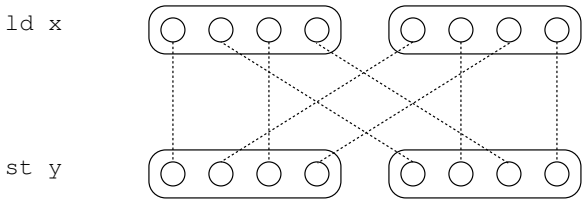

Figure 5: SIMD DFG of 8-point bit-reversal.

code, but serve to track the data dependences of common subexpressions. If a tree ends at a CSE, the result of the tree can be stored in a register, so that operations that read the CSE can use that register as one of their operands. These operands are read by a group of operations (`addsub` group) and their results stored to memory (`st y` group).

SIMD groups writing and reading common subexpressions must have the same orderings of their dependences, because they represent writing to a SIMD register and subsequent reads of the register. In the example above, the ordering of the dependences is different between the groups that create and read the CSE. The SIMD group writes the CSE in the order `(0,1,2,3)`. The SIMD group that reads the left operands, accesses results of the load SIMD group in the order `(0,0,2,2)`, and the right operands are read in the order `(1,1,3,3)`. But the data can be read from a register only in the order it was written to the register. Thus, a P-node is inserted after a node reading the common subexpression, as shown in figure 4. After insertion of the P-nodes, orderings of the SIMD groups that read CSEs correspond directly to orderings of the SIMD groups that write CSEs. The reordering patterns of the P-nodes are shown in the figure 4 as `(0,0,2,2)` and `(1,1,3,3)`.

Consider another example. This is a code for bit-reversed reordering of an array of length 8.

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

The SIMD DFG of this code is shown in figure 5. Here, each store SIMD group depends on two load SIMD groups. Thus, a P-node is inserted for each store group, and each load SIMD group creates a common subexpression read by the two P-nodes. The SIMD DFG after insertion of the needed P-nodes and common subexpression nodes is shown in figure 6.

In the previous example, the P-nodes had one or two source SIMD groups and produced results for only one SIMD group. A more general P-node can have N source SIMD groups and can also produce results for K SIMD groups. Such P-nodes are designated as $P_{N\to K}$. They are discovered while grouping the operations. When for some SIMD group its operands come from more than one SIMD group, a P-node is inserted that reads all relevant SIMD groups and feeds its output into the current SIMD group. If some of the source SIMD registers of two P-nodes are the same, the P-nodes are combined, because one $P_{N\to K}$ data reordering can be implemented more efficiently than K $P_{M\to 1}$ data reorderings.

When a P-node is factored into permutation instructions from the target ISA (see section 5), the DFG of the solution is inserted into the SIMD DFG. If the DFG of the P-node is not a tree, it is split into trees, and each tree is inserted into the SIMD DFG. This may happen if the P-node decomposition consists of more than one permutation instruction and more than one of those instructions read some P-node source.

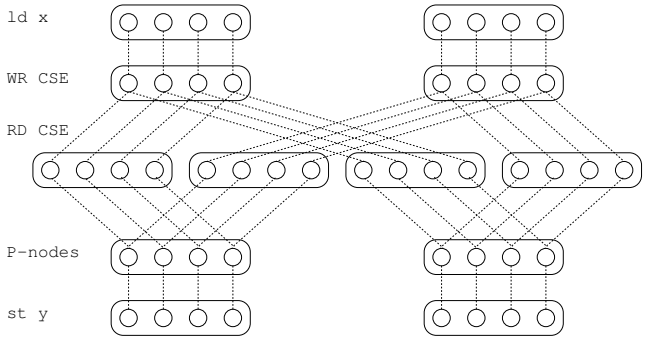

Figure 6: SIMD DFG of 8-point bit-reversal.

4. SIMD PERMUTATIONS

To generate a SIMD instruction, the operations in the SIMD group need to be assigned to specific positions in the SIMD instruction. This assignment is called an *ordering*. In the example in figure 2, the operations are easily ordered so that for all operations the results are read in the same order they are produced. If a different ordering were chosen for the SIMD group of arithmetic operations, such ordering assignment would require data reordering, and permutation instructions would have to be inserted into the SIMD DFG. For some SIMD DFGs, it may be impossible to order the operations in the SIMD groups so that for all edges in the SIMD DFG the results are produced in the same order that they are used. In such cases, it is necessary to insert permutation instructions between the groups with different orderings.

Figure 7 shows four SIMD groups. Elements of each SIMD group i are somehow ordered. Let's designate that ordering assignment as A_i. The edge between two groups corresponds to a bunch of data dependences, and individual dependences are not shown. The permutation that needs to be performed along edge (i, j) from SIMD group i to SIMD group j is shown as $P_{i\to j}$.

Initially, only SIMD groups of memory operations have orderings, which are determined by their effective addresses. These orderings are fixed and cannot be changed. For other nodes in the SIMD DFG, their orderings can be selected arbitrarily with the goal of finding an ordering assignment that minimizes the total cost of permutations, where the cost of a permutation can be, for example, the number of instructions needed to perform the permutation, or the estimated energy consumed by these instructions.

Without constraints on the orderings to be considered in search of the best ordering assignment, each SIMD group without a fixed ordering assignment (SIMD groups of operations other than mem-

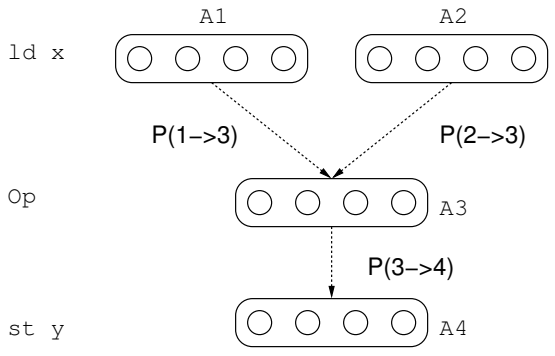

Figure 7: SIMD Group Ordering.

ory operations) can have $W!$ different orderings, where W is the number of operations in the SIMD instructions, or the width of SIMD operations. Thus, for the entire SIMD DFG there exist $W!^{N_{free}}$ choices of ordering assignments, where N_{free} is the number of SIMD nodes without constraints on ordering selection. Such a problem can only be solved directly for very small widths of SIMD operations and small SIMD trees.

4.1 Orderings Propagation

To allow the problem to scale better with the SIMD width and DFG size, some constraints must be imposed on the choices of the orderings of operations in the SIMD groups. It seems reasonable to assume that for an ordering assignment to the SIMD groups that minimizes total permutation costs, *each node in the SIMD DFG will have the same ordering as at least one of its neighbors in the graph*. Thus, the idea is to take known orderings of SIMD groups of memory operations and propagate them through the SIMD DFG. The set of orderings that a SIMD DFG node has after orderings propagation can be called the *reaching orderings* of that node. For each node with more than one reaching ordering, an ordering is selected to minimize the total cost of permutations in the basic block.

To explain how orderings are propagated from node to node, consider an edge in a SIMD DFG. One of its nodes, let's say the source node, has a number of orderings that will be propagated to the other node, the target node. For each ordering of operations at the source node, we compute an ordering of the operations at the target node such that individual data dependences arrive to the target in the same order that they start from source. If some operation in the target node reads data from an operation that was assigned position i in the source node, it is also assigned position i, so that no data reordering is required between the two nodes.

Let's estimate the total number of choices under these constraints on ordering selection. If the SIMD DFG is a perfect binary tree of height H with one more node for storing the result, the number of nodes in the tree is $N = 1+\sum_{i=0}^{H-1} 2^i = 2^H$. Each node in the SIMD DFG has one reaching ordering from the root of the tree, the store SIMD group, and one ordering from each load SIMD group from which it is reachable.

The first "leaf" level of the tree consists of 2^{H-1} load SIMD groups, whose orderings are given by the effective addresses of their operations. Let's assume that all 2^{H-1} have different orderings. On the next level, there are 2^{H-2} nodes that use results of two loads, and each is therefore reachable from only two leaf nodes and one root node. Thus each node on the second level has 3 reaching orderings. At the next level, there are 2^{H-3} nodes, with 5 possible orderings each, because 4 orderings come from 4 leaf (load) nodes from which the node is reachable and one from the store node. If the tree levels are numbered from 0 to $H-1$, at level i there are 2^{H-i-1} nodes, with $1+2^i$ choices of orderings, assuming that this number of different orderings is possible, that is $W! \geq 1+2^{H-1}$. Thus, the number of choices at level i is $(1+2^i)^{2^{H-i-1}}$. The number of different ordering assignments, N_A, for the whole tree is

$$N_A = \prod_{i=1}^{H-1} (1+2^i)^{2^{H-i-1}} = \left(\prod_{i=1}^{H-1} (1+2^i)^{\frac{1}{2^i}}\right)^{2^{H-1}} = \left(\prod_{i=1}^{\log_2 N-1} (1+2^i)^{\frac{1}{2^i}}\right)^{\frac{N}{2}}. \quad (1)$$

It can be shown that $\prod_{i=1}^{\log_2 N-1} (1+2^i)^{\frac{1}{2^i}}$ converges to a constant value C: $\lim_{N\to\infty} \prod_{i=1}^{\log_2 N-1} (1+2^i)^{\frac{1}{2^i}} = C$, and numerical estimates show that $C \approx 5.28$. Thus, the total number of different ordering assignments N_A depends on the total number of the nodes N in the DFG as

$$N_A(N) = O(C^{\frac{N}{2}}) \approx O(5.28^{\frac{N}{2}}), \quad (2)$$

compared to the $W!^{N_{free}} = W!^{\frac{N}{2}-1}$, because the number of nodes without fixed orderings in such a graph is $N_{free} = N - N_{leaf} - N_{root} = 2^H - 2^(H-1) - 1 = N/2 - 1$. The number of alternative ordering assignments still grows quickly with the size of the tree, but now it does not depend on the width of the SIMD register.

It should be noted that the above estimate is for the worst case, when orderings of all leaf nodes are different. In practice, many orderings are likely to be identical. For example, if only one of the loads has different ordering, only $H-1$ nodes on the path from that load to the store have 2 choices. In this case, $N_A = 2^{H-1} = 2^{\log_2 N-1} = N/2$.

4.2 Isolated Nodes

When orderings are being propagated through the SIMD DFG, propagation along a path stops when the path ends with a memory operation, or when it reaches a P-node. Orderings cannot be propagated through a P-node because the node itself represents a data reordering, and specific orderings of the inputs and outputs of the P-node can be selected to improve permutation costs.

It is possible that after P-nodes insertion, some nodes in the SIMD DFG become isolated, that is no ordering can reach them. This happens if every path from load or store SIMD groups to these nodes contains a P-node. However, to generate a SIMD instruction from a SIMD group, the operations in the group must be assigned to specific positions in the SIMD instructions in some order. Some ordering must be assigned to each SIMD group.

As an easy solution to this problem, positions could be assigned to the operations in the isolated nodes in an arbitrary order. For example, the operations are assigned to SIMD instruction positions in the order they are encountered in the program, which is usually useful and can be easily controlled by changing the order of the statements in the source code.

On the other hand, the total cost of the basic block can be improved by determining which orderings can be propagated through the P-nodes. Since these P-nodes are on the border of isolated nodes, they have some reaching orderings on one side, either inputs or outputs. Given a set of orderings of inputs or outputs of a P-node, it may be possible to find a set of orderings of the other side, such that the cost of implementing the P-node is minimized. These orderings can be propagated into the isolated nodes behind the P-node. The rationale for doing this is that if an ordering assignment achieves minimum permutation cost for the SIMD DFG, it is likely that at least one of the P-nodes on the border of isolated nodes will have such orderings of its inputs and outputs that minimize the cost of the P-node.

4.3 Permutation Selection with ILP

To determine the assignment of orderings to SIMD DFG nodes that achieves minimum total permutations cost, consider a SIMD DFG $G = (V,E)$, where V is the set of DFG nodes (i.e. SIMD instructions), and E is the set of edges that represent data dependences of the instructions. Using the algorithm described in section 4.1, a set of alternative orderings $\{O_{ik}\}$ is computed for each node $i \in V$, and $1 \leq k \leq s_i$, where s_i is the number of alternative orderings of SIMD node i. Each ordering O_{ik} specifies one of s_i unique orderings of operations at SIMD node i. If the cost of permutation from ordering O_{ik} to ordering O_{jl} is $permCost(O_{ik} \rightarrow O_{jl})$, and A_{ik} is a binary variable that is set to 1 when ordering O_{ik} is chosen for node i and otherwise it is set to 0, then the cost C_{ij} of

permutations for edge (i, j) is:

$$C_{ij} = \sum_{k=1}^{s_i} \sum_{l=1}^{s_j} permCost(O_{ik} \rightarrow O_{jl}) \times (A_{ik} \wedge A_{jl}). \quad (3)$$

The total cost C of permutations for the DFG G is thus

$$C = \sum_{(i,j) \in E} C_{ij}. \quad (4)$$

The constraints for the ordering assignment variables are that for each node i, exactly one ordering is selected:

$$\forall i \in V : \sum_{k=1}^{s_i} A_{ik} = 1. \quad (5)$$

The P-nodes are handled slightly differently. They themselves are only place holders for the DFGs of the permutations they represent and cannot have orderings. The cost of P-node has to be computed for all possible orderings of its neighbors. If the cost of P-node P_i with N_i neighbors, where neighbor k uses orderings j_k, is denoted as $permCost(P_i(j_1, \ldots, j_{N_i}))$, the expression for the total cost as a function of orderings assignments becomes

$$CP_i = \sum_{j_1=1}^{s_1} \cdots \sum_{j_{N_i}=1}^{s_{N_i}} permCost(P_i(j_1, \ldots, j_{N_i})) \prod_{k=1}^{N_i} A_{kj_k} \quad (6)$$

If P is the set of all P-nodes in G, the expression for total cost becomes

$$C = \sum_{(i,j) \in E} C_{ij} + \sum_{k \in P} CP_k. \quad (7)$$

The solution of this ILP model is a set of values of variables A_{ij} for which the total cost C is minimal. This selects one ordering for each node in the SIMD DFG. All orderings that were not selected are removed from the SIMD DFG, so that each node has exactly one ordering. Next, for each permutation, its representation with the target ISA instruction(s) is inserted into the SIMD DFG. Then, the SIMD DFG is split into DFTs at the CSEs, if necessary, and the SIMD DFG is ready for final code selection.

The problem of permutation selection should be slightly modified if permutation instructions from the target ISA need to load the permutation pattern, the mask, from memory into a SIMD register. The cost of loading permutation masks should be added to the total cost of permutations. Some masks may be used by more than one permutation, but loaded from memory only once.

5. PERMUTATION DECOMPOSITION

Some instruction sets allow an arbitrary permutation to be performed in one instruction, such as in AltiVec, while others have a set of permutation instructions with limited capabilities. Most SIMD instruction sets are designed to efficiently perform common permutations, but the problem of general permutation decomposition into instructions available in the target ISA is an interesting mathematical problem. This section describes how a general permutation can be decomposed into a sequence of basic permutation instructions that are available in the target ISA.

The problem formulation is as follows: given a permutation P_{ij} that reorders a source vector A_i into a goal vector A_j, determine the cost Cij of implementing such permutation with the permutation instructions from the target ISA. In general, cost can be the number of primitive permutation instructions needed to perform P_{ij}, the energy consumed by those instructions, a combination of both, etc. For simplicity, let's consider permutation cost to be the number of permutation instructions needed to achieve it.

A related problem is that given a set of permutations that need to be performed most frequently, what instructions should be available in the target architecture. Yet another problem is given a set of basic permutations in the target ISA, determine whether a given permutation can be implemented using the basic permutations. Although these problems are interesting, they are beyond the scope of this paper. The set of basic permutations in the target architecture is considered given, and if the algorithm cannot find a decomposition for some permutation, it simply returns infinite cost, thus signaling to the code generator to consider other permutation choices. If no ordering assignment to SIMD groups results in implementable permutations, the code has to be implemented without SIMD using regular scalar instructions.

One approach to determine which combination of primitive permutations results in the given permutation is to compute all possible permutations as a sequence of primitive permutations in steps, each time increasing the sequence length by one, and stop when the required permutation is reached. If B is the number of basic permutation instructions in the target architecture and if n is the number of basic permutations needed to implement a given permutation, this algorithm would need to take n steps and compute $(B^{n+1} - 1)/(B - 1)$ possible sequences of basic permutations. Not all of these sequences result in unique permutations. This approach is usable for simple permutations that require a small number of basic permutations, but for the permutations that cannot be efficiently performed with the given basic permutations the number of permutations on further steps of the algorithm quickly becomes too big.

The idea of a more efficient permutation decomposition search is to build two permutation trees. A forward tree is built from the source vectors, and a backward tree is built backward from the goal vectors. An example of a backward tree is shown in figure 8. Both trees are built in steps. The trees are checked for matching vectors after each step, and the trees are grown until there is a path from each goal vector to source vectors. With this approach, if n steps are needed to perform a given permutation, the height of each of the two trees would be approximately $n/2$, thus reducing the amount of computation by approximately a factor of $\frac{1}{2}B^{n/2}$.

The data reordering instructions fall into two classes: those that read one source SIMD register, and those that read two source SIMD registers. To build the back tree, for each permutation with two operands the pair of vectors is computed from which a given vector could be produced with this permutation. Because such instructions read twice more data than write, only half of the computed sources are actually known. Other elements of the source vectors could be anything. This property is essential for matching backward vectors to forward vectors.

Let's denote the ordering of the source vector as `(0,1,2,3)`, where each number corresponds to the initial position of an element. In the goal vector, the same numbers will be in a different order. For example, if vector `(3,2,1,0)` is traced back through `unpack_hi` instruction from Intel's SSE, it can result from `(3,1,x,x)` and `(2,0,x,x)`, where `x` means "any". Further, vector `(3,1,x,x)` could be obtained from a permutation of `(3,x,x,x)` and `(1,x,x,x)`. Thus, after $\log_2 W$ steps backward, there will be only one known element in the vector. Such a vector will match any vector with the known element in the correct position.

On the other hand, going forward from the source vectors, each element can be broadcast to an entire register in $\log_2 W$ steps. For example, if element `2` from vector `(0,1,2,3)` should fill all positions, it can be achieved with two unpack instructions.

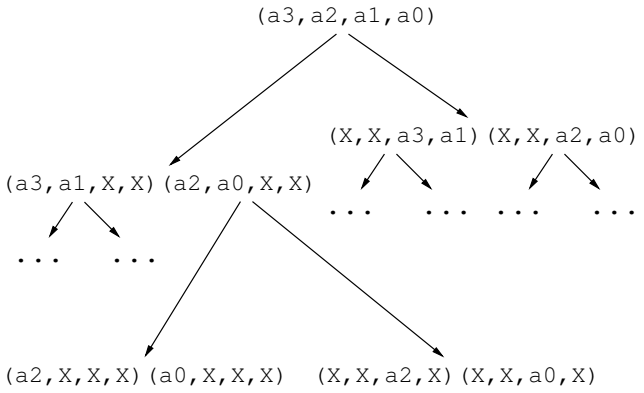

Figure 8: Permutation decomposition back tree.

```
unpack_lo(0,1,2,3)(0,1,2,3) --> (2,2,3,3)
unpack_hi(2,2,3,3)(2,2,3,3) --> (2,2,2,2)
```

Thus, to get an upper bound on the number of permutations to perform a given permutation, let's assume an algorithm that computes a forward tree by broadcasting each of the source elements to an entire SIMD register, and matches it to the back tree. The number of basic permutations needed to implement any permutation cannot be more than the minimum number of permutations to broadcast each element of the source vectors to an entire register, and the minimum number of permutations to put all of them together in the back tree. Since they form binary trees of height $\log_2 W$, the number of nodes in each tree is $2W-1$, and the total number of operations cannot be more than $4W-2$. In fact, this upper bound is not tight, and the number of permutations required to implement a given permutation is usually much smaller.

As mentioned before, it is not necessary to build the entire trees at once. The trees can be grown in steps. After each step, the vectors in a backward tree are matched to the vectors in forward tree, and new steps are computed until there exists a path from the goal vector to the source vectors. A backward vector matches a forward vector if all of its known elements are equal to the corresponding elements in the forward vector. A backward vector can match more than one forward vectors. If in the ISA there are permutations performed on a single source operand, it may be possible to reorder a forward vector so that it matches a backward vector with such an instruction. Therefore, during the matching, not only direct matches are considered, but also those resulting from applying permutation instructions that operate on one register.

Once there is a path from goal vector to the source vectors, it is likely that the path is not unique. The number of alternative paths can be rather large. Each source of a binary node in the back tree can have multiple matches in the forward tree. If one source has n matches and the other m matches, the number of different paths is $m \times n$ just at one node.

Two algorithms for searching the shortest path were implemented. The first one, optimal, finds all paths and selects one of the shortest. This is clearly expensive and in some cases takes excessive amount of time. The second algorithm uses dynamic programming. At each node it computes the cost of the node as the sum of the costs of its source nodes with minimal cost, plus the cost of the node. For this problem, the dynamic algorithm does not always find the optimal solution because using optimal solutions to subproblems does not guarantee optimality of combined solution. This algorithm often finds the same path, or path of the same cost as the optimal, but sometimes it finds a longer path. The reason is that it does not take into account that some vectors can be used more than once, and their cost is counted twice.

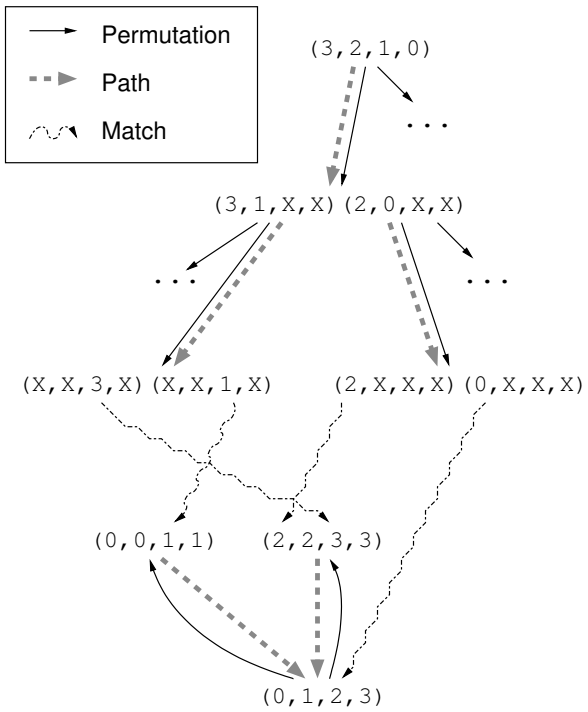

Figure 9: An example of a path from back tree to forward tree.

The following example demonstrates how this simple algorithm can overestimate the real cost of the permutation. In this example, the goal vector is `(3,2,1,0)`, and the source vector is `(0,1,2,3)`. Basic permutations are `unpack_hi` and `unpack_lo` from Intel's SSE instruction set. The cost of the source vector is 0. Costs of vectors are shown in square brackets.

```
#  src1      src2      op    res
-------------------------------------
1 (0123)[0](0123)[0] -hi- (0011)[1]
2 (0123)[0](0123)[0] -lo- (2233)[1]
3 (2233)[1](0011)[1] -lo- (3131)[3]
4 (2233)[1](0123)[0] -hi- (2021)[2]
5 (3131)[3](2021)[2] -hi- (3210)[6]
```

The cost of vector `(2,2,3,3)`, which is 1, is counted twice, and computed cost of the goal vector is 6, while it is achieved by only 5 permutations. Since the cost of this path appears to be 6, the dynamic programming algorithm can return another path of cost 6.

6. RESULTS

The following examples demonstrate performance improvements achievable with permutations on Intel SSE architecture. All the benchmarks are coded in C and compiled with both Intel `icc` compiler version 8.0 and our compiler prototype. Since our prototype does not perform register allocation, it is performed by hand. Next, our code for the benchmark kernel is inserted into the assembly file produced by `icc` to replace the original code. Then each version is run several times and the run times are averaged. This approaches focuses on the potential of using permutation instructions in SIMD code. All experiments are performed on a Pentium 4, 3.2GHz, 16kB L1, 1MB L2.

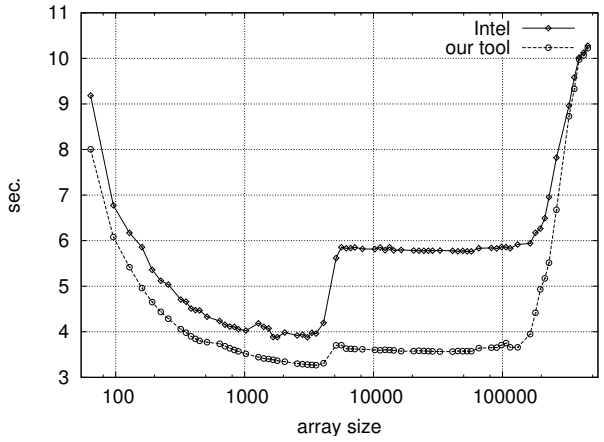

Figure 10: FIR filter run times.

6.1 FIR Filter

The Finite Impulse Response (FIR) filter is given by

$$y[n] = \sum_{k=0}^{N-1} h[k] \times x[n-k], \tag{8}$$

where $h[k]$ is the impulse response of the filter. The input sequence $x[i]$ is shifted by one element for each output sample $y[n]$. If the FIR filter is vectorized, some vectors will not be aligned in memory. The ISA may provide unaligned loads for such situations, or each vector can be loaded by performing two aligned loads and shifting the data. When more than one vector is processed, the aligned vector that is used by portions of two unaligned vectors can be loaded only once and reused, thus reducing memory traffic by half.

The Intel compiler generates pairs of unaligned loads, while our tool creates sequences of aligned loads and permutation instructions. The experiments demonstrate in figure 10 that unaligned loads are somewhat less efficient than aligned loads with permutations. Also, it should be noted that while aligned loads with permutations are equivalent to unaligned loads, they offer more opportunity for out-of-order execution in the processor, which allows the processor to effectively hide misses in L1 cache. This, however, is not enough to tolerate misses in L2 cache. The performance improvement due to permutations is shown in figure 11.

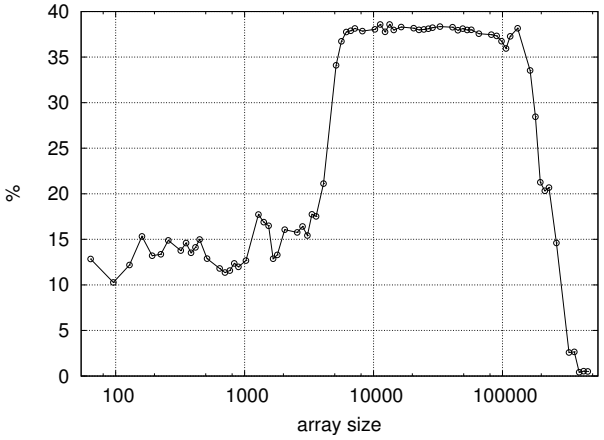

Figure 11: FIR filter improvement.

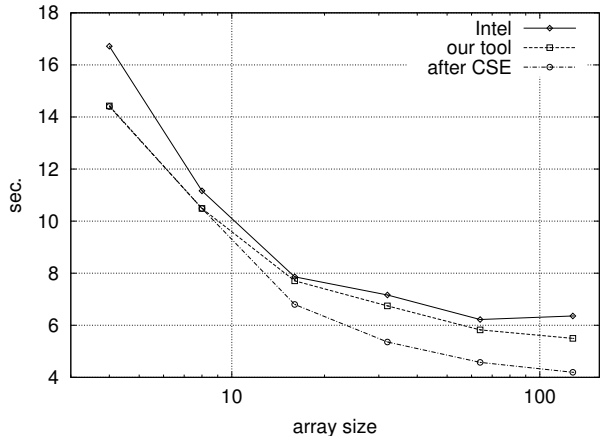

Figure 12: Bit-reversed reordering run times.

6.2 Bit-reversed Data Reordering

In this benchmark, the elements of an array are reordered in such a way that the new index of an element is produced by reversing the order of bits in the old index. This kind of data reordering is performed in an FFT after the butterfly is computed. The data dependences in this benchmark is rather complex, and traditionally, it is not vectorized, but as shown in figure 12, it has a potential for vectorization.

Our prototype computes factorization ot each permutation separately. In some cases, the sequences of basic permutation instructions needed to achieve given permutations may have identical portions, as shown by the code example below. The first four instructions load the array into four SIMD registers. The permutations marked by (1) and (2) are identical, and the second instance can be safely removed. They both compute vector `(8,12,9,13)` in register `%xmm4` and vector `(0,4,1,5)` in register `%xmm5`. These vectors are then permuted to obtain vector `(0,8,4,12)` and store it to `y`, and vector `(1,9,5,15)`, which is stored to `y+32`. Such redundancy can be eliminated with a standard compiler technique, common-subexpression elimination [9].

```
    movdqa x,%xmm0
    movdqa x+16,%xmm1
    movdqa x+32,%xmm2
    movdqa x+48,%xmm3

(1) MOVDQA %xmm2,%xmm4      (2) MOVDQA %xmm2,%xmm4
(1) PUNPCKLDQ %xmm3,%xmm4   (2) PUNPCKLDQ %xmm3,%xmm4
(1) MOVDQA %xmm0,%xmm5      (2) MOVDQA %xmm0,%xmm5
(1) PUNPCKLDQ %xmm1,%xmm5   (2) PUNPCKLDQ %xmm1,%xmm5
    MOVDQA %xmm5,%xmm6          MOVDQA %xmm5,%xmm6
    PUNPCKLDQ %xmm4,%xmm6       PUNPCKHDQ %xmm4,%xmm6
    movdqa %xmm6,y              movdqa %xmm6,y+32
```

6.3 Matrix Transposition

In this benchmark, the $N \times N$ matrix is split into blocks. If the size of the SIMD register is 128 bits and the size of the elements is 32 bits, then the block size is 4×4. Each row of the block is read into a SIMD register, they are permuted to achieve block transposition and the rows of the transposed block are stored. Similarly to the bit-reversed data reordering (section 6.2), our tool produces some redundant permutations that can be removed by common-subexpression elimination. The benchmark is run for different matrix sizes N, and the run times and improvement due to the use of permutations are shown in figures 13 and 14 respectively.

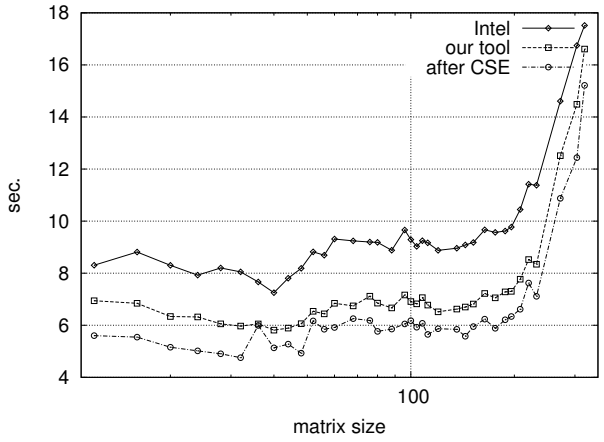

Figure 13: Matrix transposition run times.

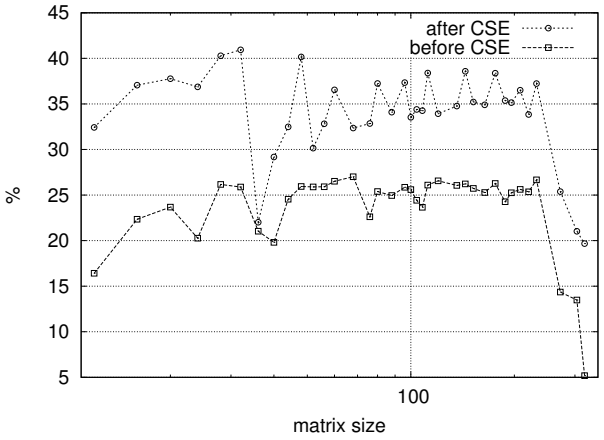

Figure 14: Improvement in matrix transposition.

7. CONCLUSION

In recent years, SIMD extensions have become ubiquitous. Most processors on the market today have them in some form. However, commercial compilers still cannot always use the opportunity to improve performance of an application on a SIMD processor by automatically generating SIMD instructions and permutations. Thus, when code efficiency is required, it has to be written in assembly language or with compiler builtin functions. Applications with potential for performance improvement with SIMD code are very common. They span from embedded systems to scientific computations. For all of these applications, their performance or energy efficiency can be improved by using SIMD code and permutations. In this paper, we present an approach to solve an important problem of generating permutation instructions for SIMD processors. It is shown that the relationship between operations grouping and permutations generation is important. They are two parts of one problem, rather than two independent problems. Also, an approach to permutation decomposition is described. The SIMD code generation is performed on the basic block level, and the SIMD code is generated from plain C code. To form SIMD instructions, memory operations are grouped into SIMD instructions based on their effective address. Other operations are grouped starting from the memory operations groups. Selection of permutations is optimized with ILP. The proposed method of creating SIMD instructions and permutation selection has better scalability with the SIMD width than existing approaches. Permutations decomposition for generic permutation instruction set is performed by building forward and backward trees of permutation instructions. This approach is not tied to any particular architecture and can be relatively easily ported to any SIMD instruction set. The potential of this approach is demonstrated with Intel SSE, because the architecture is wide spread and well known.

8. ACKNOWLEDGMENTS

The material in this paper is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003. We are very grateful to R. Leupers for giving us access to his code selector, which served as a basis for our tool.

9. REFERENCES

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Prog. Lang. Syst.*, 11(4):491 – 516, Oct. 1989.

[2] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI*, pages 82–93, June 2004.

[3] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. In *Workshop on Languages and Compilers for Parallel Computing*, pages 290–304, Aug. 1998.

[4] Intel Corporation. *Intel© C++ Compiler for Linux* Systems User's Guide*, 2003.

[5] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 145–156, Vancouver, British Columbia, Canada, June 2000.

[6] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Sept. 2002.

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

[8] R. Leupers. Code selection for media processors with SIMD instructions. In *Design, Automation and Test in Europe*, pages 4–8, Mar. 2000.

[9] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[10] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES*, pages 2–11, San Jose, CA, Oct. 2003.