Introducing Control Flow into Vectorized Code

Jaewook Shin
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA
jaewook@mcs.anl.gov

Abstract

Single instruction multiple data (SIMD) functional units are ubiquitous in modern microprocessors. Effective use of these SIMD functional units is essential in achieving the highest possible performance. Automatic generation of SIMD instructions in the presence of control flow is challenging, however, not only because SIMD code is hard to generate in the presence of arbitrarily complex control flow, but also because the SIMD code executing the instructions in all control paths may slow compared to the scalar original, which may bypass a large portion of the code. One promising technique introduced recently involves inserting branches-on-superword-condition-codes *(BOSCCs) to bypass vector instructions. In this paper, we describe two techniques that improve on the previous approach. First, BOSCCs are generated in a nested fashion so that even BOSCCs themselves can be bypassed by other BOSCCs. Second, we generate all* `vec_any_*` *instructions to bypass even some predicate-defining instructions. We implemented these techniques in a vectorizing compiler. On 14 kernels, the compiler achieves distinct speedups, including 1.99X over the previous technique that generates single-level BOSCCs and* `vec_any_ne` *only.*

1 Introduction

As the clock frequency of microprocessors flattens, more attention is being paid to exploiting parallelism to satisfy the ever-increasing need for high performance. While parallelism can be categorized into multiple levels depending on the size of the program units being executed in parallel, for the best performance it is important to exploit parallelism at all levels, including *single instruction multiple data* (SIMD) parallelism.

To support SIMD parallelism, manufacturers have extended their processors with support for short vectors [16, 20, 23]. If the short vectors are larger than the size of a machine word, then they are called *superwords* [15]. The rationale behind this extension is that adding such an extension does not cost much in terms of die area and design time but sometimes gives a large benefit when the SIMD units can be utilized [16, 29]. However, writing parallel programs manually is not easy, is error prone, and sometimes undermines the portability of the code. We believe that parallelization has to be performed automatically by software tools such as the compiler. There are two approaches to generate SIMD instructions automatically. The first is to adapt the vectorization technique developed for the conventional vector machines. More recently, Larsen and Amarasinghe developed a technique to exploit SIMD parallelism from basic blocks [15]. This fine-grained SIMD parallelism in a superword is called *superword-level parallelism* (SLP).

Currently, several compilers can generate SIMD instructions automatically [15, 5, 21, 11]. They can generate efficient code for well-formed input programs, but their SIMD code generation is limited by several factors. One such factor is control flow. For conventional vector computers, vectorization in the presence of control flow involves *if-conversion* followed by generating vector instructions guarded by vector predicates [13]. In 1991, Park and Schlansker developed an if-conversion technique that is optimal in the number of predicates used and in the number of predicate-defining instructions [22]. Our earlier work used this technique to generate SIMD instructions for modern microprocessors in the presence of arbitrarily complex acyclic control flow [26]. However, the generated code still has to execute instructions in all control paths to merge values. Consequently, when branches in the scalar baseline are frequently taken to bypass a large number of instructions, the vector code may slow compared to the baseline.

To overcome this overhead of executing all control paths, we introduced a technique to generate *branches-on-superword-condition-codes* (BOSCCs) automatically [25]. A sequence of consecutive vector instructions guarded by the same vector predicate can be bypassed by a BOSCC if all fields of the guarding vector predicate are false. If

viewed in conjunction with the scalar baseline, a BOSCC is taken when all corresponding scalar branches are taken. For the loops with simple control flow where if-statements are not nested, this technique improves the performance of the generated SIMD code. For the loops with *complex* control flow where if-statements are nested, however, the technique generates a long sequence of non-nested if-statements. As a result, although the vector instructions might be bypassed, all vector predicates have to be checked to set the superword condition code, and all BOSCCs have to be executed.

In this paper, we describe a technique to nest BOSCC instructions so that multiple BOSCCs and the vector instructions enclosed within them can be bypassed by a single BOSCC instruction. A key idea of this technique is to utilize the relation $\neg VP1 \implies \neg VP2$ between the two vector predicates $VP1$ and $VP2$. If this relation holds, whenever the vector instructions guarded by $VP1$ can be bypassed, the vector instructions guarded by $VP2$ can also be bypassed, including the BOSCC for $VP2$. In addition, we describe a new code generation technique that can generate all `vec_any_*` instructions to bypass even some predicate-defining instructions.

The contributions of this paper are summarized below.

- Discovery of a key relation between vector predicates that can be used to generate nested control flow in vector code
- A code generation technique that allows a BOSCC to bypass more instructions by generating all `vec_any_*` instructions
- Development of an algorithm that exploits the relation and code generation technique described above
- Experimental evaluation of the algorithm and the interaction of two predicate-obtaining approaches with redundancy elimination on 14 kernels

In the next section, we describe the terms and concepts necessary to understand the later sections. Section 3 introduces the two techniques using examples to motivate this research. In Section 4, we describe the key relation between vector predicates. The algorithms are described in Section 5. In Section 6, we present our implementation and the experimental results on a set of 14 benchmarks. Section 7 discusses related work, and Section 8 summarizes our research.

2 Background

We begin by presenting background information about vectorization and predication. This information will be used in the subsequent sections.

2.1 If-Conversion

The SLP compiler applies the RK-algorithm of Park and Schlansker [22] to generate a large basic block of predicated instructions. We denote an instruction guarded by a predicate `pred` as follows.

```
dst = operation; <pred>
```

The semantics of this notation is that if `pred` is true, `dst` is updated with the operation's result; otherwise, `dst` remains unchanged. Thus, we can introduce an if-statement as follows and still preserve the original semantics of the predicated instruction.

if (pred) { `dst = operation; <pred>` }

We use `pset` as a predicate-defining instruction whose syntax is as follows.

```
pT, pF = pset(cond) <pred>
```

The instruction `pset` takes one source operand and two destination operands. The source operand is the result of a previous comparison instruction, and the two destination operands are predicate variables that can be used to guard the subsequent instructions. A `pset` instruction itself can also be guarded by another predicate just like any other instruction. The semantics of `pset` is that `pT = cond` and `pF = !cond` when `pred` is true. If `pred` is false, both `pT` and `pF` remain unchanged.

2.2 Predicate Hierarchy Graph

After the if-conversion is applied, some instructions in a basic block may have predicates. To identify relations among predicates, we use Mahlke's analyses based on a *predicate hierarchy graph*. Some relevant concepts are defined as follows [18].

Definition 1 *A predicate hierarchy graph (PHG) is a directed acyclic graph representing the Boolean equations used to compute all predicates in a basic block (after the if-conversion has been applied).*

A PHG comprises two types of nodes, *predicate nodes* and *condition nodes*. Initially, a PHG consists of a single predicate node for the NULL predicate, which represents the constant *true*. In order to construct a PHG, instructions are examined in textual order. For each instruction that defines predicates, at most two predicate nodes are created, representing the *true* and *false* values of the comparison. For example, for a predicate-defining instruction such as `pT, pF = pset(comp) <pParent>`, one condition node for `comp` is created as a child of the predicate node for `pParent`. The two predicate nodes representing `pT` and `pF` are also created as children of the condition code for `comp`, if they do not already exist. This process is repeated for each predicate-defining instruction.

2.3 Relations between Predicates

From the PHG built, we can extract some useful relations between predicates, such as *ancestor*-relation and *implies*-relation [18].

Definition 2 *A predicate P1 is an **ancestor** of another predicate P2 if all conditions used to compute P2 are derived directly or indirectly from P1.*

To determine whether P1 is an ancestor of P2, we traverse the PHG backward from the node for P2 toward the root node, examining all possible paths. For P1 to be an ancestor of P2, all paths from the root node to the node for P2 should contain the node for P1. If P1 is an ancestor of P2, $P2 \implies P1$.

Definition 3 *A predicate **implies** another predicate when the conditions for the first predicate being true guarantee that the second predicate is also true.*

A situation can arise where a predicate P1 implies another predicate P2 but P2 is not necessarily an ancestor of P1. One such situation occurs if, for each condition leading to the predicate node for P1, there is the same condition leading to the predicate node for P2, in which case there can be a condition node C that leads to P2 but does not necessarily lead to P1 [1].

2.4 Vectorization in the Presence of Control Flow

Based on if-conversion and PHG, code with any arbitrarily complex acyclic control flow can be vectorized. For multimedia extension architectures that do not support predicated execution, however, vector predicates must be converted to something semantically equivalent. `Select` instructions can be used to remove vector predicates [26]. The `select` instruction

```
dst = select(src1, src2, mask)
```

assigns `src2` to `dst` for the fields where the corresponding `mask` bit is 1. Otherwise, `src1` is assigned to `dst`.

2.5 Branch-on-Superword-Condition-Code (BOSCC)

BOSCC is a branch instruction that can be conditionally taken based on the comparison result of two vector variables. AltiVec supports BOSCC instructions with *AltiVec predicates* [19]. For example, the predicated vector instruction

```
Vdst = vec_operation; <Vpred>
```

[1] Figure 3(c) shows an example. P5 implies P6, but P6 is not an ancestor of P5.

can be bypassed by introducing a BOSCC instruction as follows.

```
NotTaken = vec_any_ne(Vpred, ZeroVector)
if (NotTaken) { Vdst = vec_operation; <Vpred> }
```

The `vec_any_ne` instruction returns *true* if *any* field of `Vpred` does not match the corresponding field of `ZeroVector`. Assuming `ZeroVector` contains *false* values in all fields, `NotTaken` will be set to *false* only when all fields of `Vpred` are false. We use *predicate region* to refer to a sequence of consecutive instructions guarded by the same predicate and *BOSCC region* to refer to the sequence of instructions enclosed by a BOSCC. By enclosing the vector instructions guarded by vector predicate `Vpred` inside an if-statement, we bypass them whenever the guarding vector predicate has all false values, which is the only case when `NotTaken` is set to false.

By inserting a BOSCC instruction, we increase the number of instructions statically in the hope of reducing the number of instructions executed during run time. Thus, it is important to know how frequently the BOSCC being inserted will be taken. According to the above code generation style, a BOSCC is taken only when the values in all fields of the given vector predicate are false. Thus it is useful to know the *percentage of all false superwords*, defined as follows.

Definition 4 *The percentage of all false superwords (PAFS) of a vector predicate VP is the percentage of the run-time values of VP where all fields are false.*

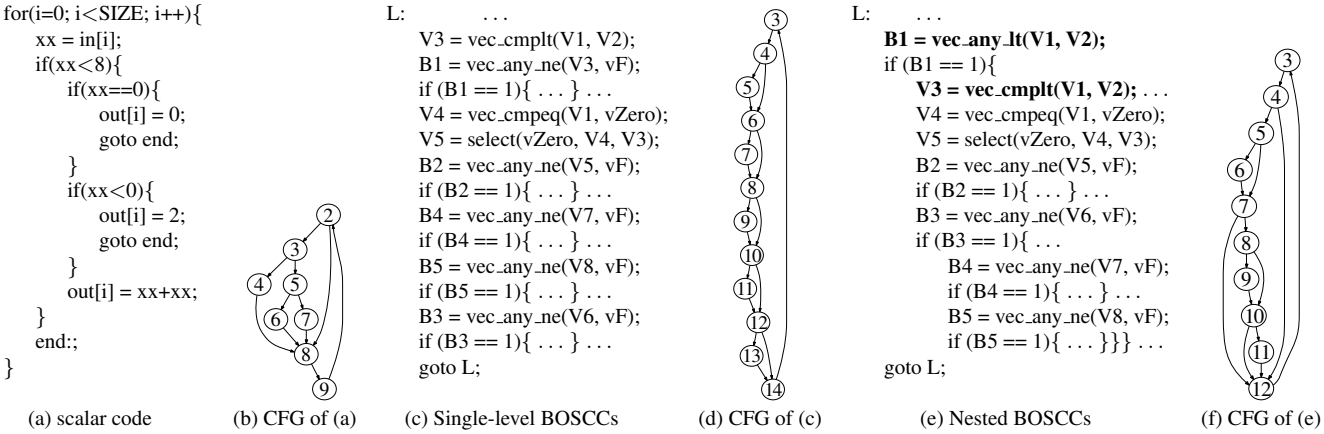
$$PAFS = \frac{\textit{\# run-time values of VP, whose fields are all false}}{\textit{\# all run-time values of VP}}$$

3 Two New Techniques

In this section, we use simple examples to introduce two new techniques: nesting BOSCCs and bypassing more instructions by generating all `vec_any_*`.

3.1 Nesting BOSCCs

Figure 1(b) shows the control flow graph of the example code shown in Figure 1(a), where the loop body has four control paths. When vectorized without using BOSCCs, the vector instructions along all four control paths have to be executed to merge the values computed along the control paths. As a result, if edge $2 \rightarrow 8$ is taken most of the times during scalar execution, the vector code may slow. In order to address this problem, BOSCCs can be inserted to bypass vector instructions [25]. Figures 1(c) and 1(d) show the BOSCC inserted code and the corresponding control flow graph, respectively. Note that all BOSCCs are single-level if-statements. In this code, vector instructions are executed only if the corresponding scalar instruction is



(a) scalar code (b) CFG of (a) (c) Single-level BOSCCs (d) CFG of (c) (e) Nested BOSCCs (f) CFG of (e)

Figure 1. Motivating example for nesting BOSCCs and generating all `vec_any_*`

executed at least once during the corresponding scalar iterations. Although this approach is efficient when the control flow is simple, the drawback is evident for more complex control flow; since the algorithm does not nest BOSCCs, all BOSCCs must be executed always.

We suggest a technique to nest BOSCCs, allowing BOSCCs to bypass other BOSCCs and the instructions enclosed within them. Figures 1(e) and 1(f) show the code generated by our approach and the CFG. Nesting the BOSCCs can reduce the number of BOSCCs that are executed. Since we do not generate `else` part of BOSCCs, the statements in the `else` part of the scalar code are guarded by a separate BOSCC. Although we have used a simple example for clarity, the strength of this approach, compared to the previous approach, becomes clear for programs with more complex control flow.

3.2 Bypassing More Instructions by Generating All `vec_any_*`

The other technique we introduce in this paper is a code generation optimization that can be used to bypass more instructions than the previous approach. To illustrate the differences between the two approaches, we use the example in Figure 1 again.

The previous BOSCC generation approach checks each vector predicate for any true values. In the code generated by the previous approach shown in Figure 1(c), all vector predicates are computed before being used in `vec_any_ne`'s. We suggest bypassing even predicate-defining instructions shown in bold in Figure 1(e). Here, the instruction defining predicate `V3` is enclosed by the BOSCC for the same predicate; however, the comparison instruction defining `B1` has changed from `vec_any_ne` to `vec_any_lt`, together with the source operands. As a result of this change, `B1` is defined by the same value as before, but the predicate-defining instruction can also be bypassed if it would define predicate `V3` with all false values.

Although this optimization can also be applied to the BOSCCs nested within other BOSCCs, doing so can degrade performance. Note that the instructions defining `V4` and `V5` are not enclosed by the inner BOSCC for `B2` in Figure 1(e). In fact, while abbreviated, all instructions defining the vector predicates `V6`, `V7`, and `V8` are not enclosed within the BOSCC regions for them. Applying this code generation optimization to the BOSCC defining instructions at all nested levels will result in a code that is still correct. For example, if we did, the two statements defining `V4` and `V5` would be enclosed in the BOSCC region for `B2` by changing the instruction defining `B2` into "`B2 = vec_any_eq(V1, vZero)`". However, this aggressive bypassing often entails an increase in the frequency of execution for the nested BOSCC regions because the BOSCC regions will be executed based on a single comparison rather than on the conjunction of the vector predicates of all outer BOSCCs. For example, the value of `B2` would be a result of single comparison, `vec_any_eq(V1, vZero)` rather than the conjunction of two comparisons, one for `V3` and the other for `V4`. Thus, we apply this optimization to the outermost BOSCCs only. Although this technique reduces only one instruction per BOSCC, when a loop contains many non-nested BOSCC regions with a small number of instructions, reducing even one instruction per BOSCC can result in a large gain in performance.

4 Inverse-Implies Relation

We start by defining the *inverse-implies* relation, which is a core concept in nesting one region of predicated instructions inside another.

Definition 5 *Given two predicates P1 and P2, P1* inverse-implies *P2 iff* $\neg P1 \implies \neg P2$.

The inverse-implies relation can also be viewed as the contraposition of the *implies* relation. Given $P2 \implies P1$, the contraposition is $\neg P1 \implies \neg P2$, which means that whenever P1 is false, P2 is also false. Figure 2(a) shows an example of two predicated vector instructions to illustrate how we use this relation to nest BOSCC regions. As shown in Figure 2(b), we can introduce an if-statement around each predicated instruction. When p1 *inverse-implies* p2 ($\neg p1 \implies \neg p2$), if all fields of p1 are false, all fields of p2 are false as well. Hence, we can bypass the instruction guarded by p2 just by checking whether p1 are all false, as shown in Figure 2(c).

```
dst1 = operation1; <p1>
dst2 = operation2; <p2>
```

(a) predicated instructions

```
if (p1)
    dst1 = operation1; <p1>
if (p2)
    dst2 = operation2; <p2>
```

(b) Single-level BOSCCs

```
if (p1) {
    dst1 = operation1; <p1>
    if (p2) {
        dst2 = operation2; <p2>
    }
}
```

(c) Nested BOSCCs

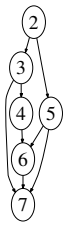
Figure 2. Nesting if-statements using an *inverse-implies* relation.

We use an *inverse-implies graph* to represent the inverse-implies relations between predicates. An inverse-implies graph is a tuple (V, E), where a node $n \in V$ represents a predicate in a given basic block and there is an edge $(n_1, n_2) \in E$ if the predicate of node n_1 inverse-implies the predicate of node n_2. Inverse-implies graphs are acyclic but not necessarily trees. Figure 3(a) is an example designed to illustrate this point. Figure 3(b) shows the CFG of the code in (a), and Figure 3(f) shows the mappings between the statements in (a) and the nodes in (b). Figure 3(c) is the PHG of the code in (a) after it is predicated. Predicate nodes are shown as boxes, and condition nodes have circular shapes. The root node of PHG, labeled P0, always represents the constant *true* predicate. The condition nodes C1, C4, and C7 represent the statements 1, 3, and 8 of the code in Figure 3(a), respectively.

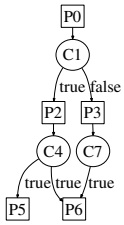
Now let us turn to the inverse-implies graph shown in Figure 3(d). We use edge $P2 \rightarrow P5$ of this graph as an example to show how the inverse-implies graph is used to nest regions of predicated instructions and BOSCCs. The edge means that whenever vector predicate P2 has all false values, the vector predicate P5 is guaranteed to have false values in all fields as well and that it is safe to bypass the instructions guarded by P5 whenever we can bypass the instructions guarded by P2. This is exactly what we did for the example code in Figure 2(c) with an inverse-implies edge $p1 \rightarrow p2$. In terms of the source code in Figure 3(a), the

```
1. if (b[i] < 0){ // C1
2.     b[i] = 0;
3.     if (a[i] > 10){ // C4
4.         c[i] = a[i] + b[i];
5.         goto L1;}}
6. else{
7.     b[i] = b[i] + 1;
8.     if (a[i] > 0){ // C7
9.         L1:
10.        c[i] = a[i] - b[i];}}
```

(a) C program



(b) CFG of (a)



(c) PHG of (b)

(e) CFG of the vectorized code

(d) inverse-implies graph of (c)

Stmts	Nodes		
(a)	(b)	(c)	(e)
1	2,7	P0	4,7,9,11
2,3	3	P2	5
4,5	4	P5	6
6,7,8	5	P3	8
9,10	6	P6	10

(f) Mapping between the statements and nodes

Figure 3. Example showing that the inverse-implies graph is not a tree.

inverse-implies edge $P2 \rightarrow P5$ informs us that whenever we can bypass the vector codes for the statements 2 and 3, we can also bypass the vector codes for the statements 4 and 5. Likewise, the inverse-implies edge $P6 \rightarrow P5$ means that the vector codes for statements 4 and 5 can be bypassed whenever the vector codes for statements 9 and 10 can be bypassed. The three outgoing edges from node P0 are not useful because, by definition, P0 will never be false and no BOSCC regions can be nested based on these edges. Figure 3(e) shows the CFG of the vectorized code, where BOSCCs are generated in a nested fashion. Figure 3(f) shows the mappings between the nodes of the PHG and the CFG nodes in Figure 3(e). In general, the two CFGs for the scalar and vector codes are not isomorphic.

Inverse-implies relation between predicates is closely related to *dominator* and *postdominator* relations between basic blocks of the corresponding CFG. If node A dominates or postdominates node B in a CFG, the predicate of node A is equal to or inverse-implies the predicate of node B.

5 Algorithm

In this section, we describe an algorithm to insert BOSCCs in a nested fashion while enclosing predicate-defining instructions when profitable. This algorithm is based on our previous work to generate single-level BOSCCs [25].

Figure 4 shows a high-level view of the algorithm. The lower four boxes are either newly added or modified for this

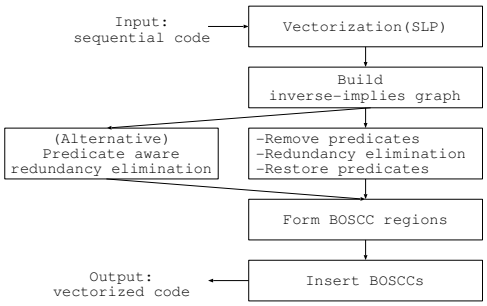


Figure 4. Overview of the algorithm.

work. After vectorization, the basic block contains vector instructions possibly guarded by vector predicates. From this predicated vector code, we build an *inverse-implies* graph. Next, we apply a set of compiler passes, which includes removing both vector and scalar predicates, redundancy elimination, and restoring vector predicates for the subsequent BOSCC insertion. For this step, we also explore an alternative approach, where vector predicates are preserved rather than removed and restored back. When this alternative approach is used, the passes for both removing and restoring vector predicates are skipped, and the redundancy elimination is performed in a way that honors predicates. Next, the instructions that are guarded by the same predicate are collected so that they form a region of adjacent instructions guarded by the same predicate. As the last step, BOSCCs are inserted so that they can bypass regions of contiguous instructions. The five boxes of Figure 4 except the one labeled "Vectorization(SLP)" are the subjects of the next five subsections.

```
Algorithm BuildInverseImpliesGraph (basic block bb):
    phg ← build a predicate hierarchy graph (bb)
    inverseImpliesGraph ← new Graph
    for each predicate p1 ∈ phg
        for each predicate p2 ∈ phg where p1 ≠ p2
            if ( p1 implies p2 ∨ p2 is an ancestor of p1 )
                inverseImpliesGraph ∪ ← edge(p2 → p1)
```

Figure 5. Algorithm to build inverse-implies graph.

5.1 Building an Inverse-Implies Graph

Figure 5 shows an algorithm to build an inverse-implies graph for a basic block of predicated vector instructions. After building a PHG, all possible pairs of vector predicates are examined to check whether one inverse-implies the other. To find an inverse-implies relation between two predicates p1 and p2, we use the algorithms described in [18]. If either p1 *implies* p2 or p2 is an *ancestor* of p1, we add an edge to the inverse-implies graph from the node for p2 to the node for p1. The result of the algorithm in Figure 5 is a directed acyclic graph but may contain redundant information by having all inverse-implied predicate nodes as immediate successors. We minimize the graph by removing the edges whose head is reachable from the tail without going through the edge.

```
Algorithm RestoreVP (basic block bb, inverse-implies graph iiGr):
    dGraph ← build a dependence graph (bb);
    vCmpTB ← build a vector compare table (bb);
    for each predicate p ∈ iiGr in postorder
        for each select I:"dst = select(src1, src2, p)" ∈ bb \
            where dst == src1
            I.predicate ← p;
            PredicateDefs(src2, I, p, iiGr, dGraph, vCmpTB);
            PredicateMemAcc(src1, dst, p, iiGr, dGraph);
            if (vCmpTB[p])
                PredicateDefs(p, I, p, iiGr, dGraph, vCmpTB);
    Schedule(dGraph);
```

(a) Restoring vector predicates

```
Algorithm PredicateDefs(operand opr, statement s, predicate p, \
    inverse-implies graph iiGr, dependence graph dGr, \
    vector comparison table vCmpTB):
    rd ← reaching definition of source operand opr in s;
    if (rd is outside the basic block ∨ (opr == p ∧ !vCmpTB[p]) ∨ \
        rd is used by another statement whose predicate is neither p \
        nor its descendant ∨ p is not a descendant of the predicate \
        of rd in iiGr) return;
    rd.predicate ← p; update dGr;
    if (rd == vCmpTB[p]) return;
    for each source operand src of rd
        PredicateDefs(src, rd, p, iiGr, dGr, vCmpTB);
```

(b) Restoring predicates for reaching definitions

Figure 6. Algorithm to restore vector predicates.

5.2 Preserving Predicates While Inserting `select`s

In order to insert BOSCCs, vector code must have predicate information. The previous approach used an algorithm to restore vector predicates that had been removed [25]. But wouldn't it be easier just to keep the vector predicates and provide a direct mapping between the vector predicates in the inverse-implies graph and the ones encountered in the code by the later steps of the algorithm? It turns out that for the set of benchmarks used in this paper, preserving vector predicates is not a good idea because all intervening compiler optimizations such as redundancy elimination and loop invariant code motion must be aware of predicates and this predicate awareness limits their applicability, significantly in some cases. In Section 6, we present the experimental comparisons between the two approaches. In this subsection, we describe how we preserve vector predicates for the experiments.

Since `select` instructions are inserted to remove vector predicates [26], one might think naively that we can just

not insert `selects` to preserve predicates. `Select` instructions still have to be inserted, however, because predicates should be removed after BOSCCs are inserted for the output code to run on the architectures that do not support predicated execution. The statement guarded by a predicate `pred` shown below

```
dst = operation; <pred>
```

is transformed to the two statements

```
temp = operation; <pred>
dst = select(dst, temp, pred); <pred>
```

which are semantically identical to the original statement both with and without the predicate.

5.3 Restoring Predicates from `selects`

We restore vector predicates starting from `select` instructions similar to our previous approach. We extend the previous algorithm so that even predicate-defining instructions are guarded by the predicate they are defining. After a vector predicate is restored to guard the vector compare instruction defining itself, the instruction appears as if it uses the predicate before defining it. However, this is only to convey the predicate information to the later passes so that the predicated instructions can be bypassed by a BOSCC generated from the same condition. The extended algorithm is shown in Figure 6. Initially, a table of vector compare instructions is generated to store mappings from vector variables to the vector compare instructions defining them. This table provides the vector compare instructions as a barrier to stop predicating reaching definitions beyond it. In other words, we want to bypass the vector compare instructions but not the ones defining the source operands of them. The second if-statement of Figure 6(b) checks for this situation. The second call to `PredicateDefs` in Figure 6(a) predicates the reaching definition of the third source operand of `select` instructions only when the definition is a vector compare instruction. Since the vector predicates that are nested within another predicate are defined by `select` instructions to merge predicate values, the table entries for them are NULL and this checking of `vCmpTB` filters the new code generation technique for the nested vector predicates. `PredicateMemAcc` is not shown but is the same as `IdentifyMemoryAccesses` of [25].

5.4 Forming BOSCC Regions

The next step is to collect the instructions guarded by the same predicate so that they are textually contiguous. Figure 7 shows the algorithm that performs this operation. For each predicate P in postorder of the given inverse-implies graph, we perform two operations. First, for each instruction guarded by P in the textual order, we follow the def-use

```
Algorithm FormBosccRegions (basic block bb, inverse-implies graph \
    iiGr, vector comparison table vCmpTB):
    depGr ← build a dependence graph (bb)
    duGr ← build a def-use graph (bb, phg)
    for each predicate p ∈ iiGr in postorder
        // expand the instruction region guarded by p
        for each statement s guarded by p in the textual order
            duNode ← duGr.getNode(s)
            for each predecessor preNode of duNode
                if ( p is a descendant of the predicate of preNode in iiGr ∧\
                    (preNode.statement does not define p ∨ vCmpTB[p]) ∧ \
                    all successors of preNode are guarded by p or the \
                    descendants of p ∧ preNode is not a predecessor of the \
                    node for vCmpTB[p]) preNode.predicate ← p
        // merge dependence graph nodes guarded by p
        tempNode ← ∅
        for each node n ∈ depGr in reverse postorder
            if (depGr.canBeScheduledAfter(n, tempNode) ∧\
                n.predicate is tempNode.predicate or its descendant in iiGr)
                tempNode.append(the statement of n)
            else tempNode ← n
    schedule instructions in depGr
```

Figure 7. Algorithm to form BOSCC regions.

graph backward to see whether the predecessor instructions can also be guarded by P. Predecessors of an instruction in the def-use graph are the instructions whose definition reaches the instruction. The predecessor's predicate P' is replaced by P if P is a descendant of P' in the inverse-implies graph and all uses of the predecessor statement are guarded by either P or the descendants of P in the inverse-implies graph. By doing so, we are trying to guard each instruction with as restrictive a predicate as possible, so that the instruction can have a larger chance of being bypassed.

The second operation is to schedule the instructions guarded by the same predicate or its descendant predicates as contiguously as possible. For example, consider two instructions guarded by the same predicate P. If there is an intervening instruction guarded by P' that is not a descendant of P in the inverse-implies graph, two BOSCCs have to be generated to bypass each instruction separately. The two instructions can be scheduled contiguously unless one or more instructions intervene between them in the dependence graph. Note that with this operation, the instructions guarded by two different predicate P1 and P2 are scheduled contiguously if the inverse-implies relation holds between them.

5.5 Inserting BOSCCs

At this point of the algorithm, the instructions that can be bypassed by a single BOSCC have been scheduled so that they are textually contiguous. The remaining task is to find the largest region of contiguous instructions that can be bypassed by a single BOSCC, check whether inserting the BOSCC is profitable, and insert a BOSCC if so. Here, we make an observation that can reduce the overhead of

Table 1. Benchmark programs.

Name	Description	Type size (bits)	# lines	PAFS of Figure 9	PAFS of Figure 10
Chroma	Chroma keying of two images	8(char)	128	98	97
Sobel	Sobel edge detection	16(int)	135	0,4,19,99	17,4,2,82
TM	Template matching	32(int)	84	89	87
Max	Max value search	32(float)	93	88,85	100,99
TR	Shortest path search	32(int)	98	0	0
dist1	`dist1` of MPEG2 encoder	8(char),32(int)	163	0∼31	20∼38
unquantize	`unquantize_image` of unEPIC	16(int),32(int)	92	6∼40	0,4,5,82,83,83,83
Calculation	`Calculation_of_the_LTP_parameters` of GSM	16,32(int)	207	1∼7	2∼10
logf	Single-precision `log` of glibc-2.4	32(float)	184	(5*) 0, (17*) 100	24∼98, (12*) 100
sinf	__kernel_sinf of glibc-2.4	32(float)	145	0,6,7,100,100	0,5,6,100,100
LARp	LARp_to_rp of GSM encoder	16(int)	114	0,75,75,(23*)100	0,70,70,97,(22*)100
ceilf	Single-precision `ceil` of glibc-2.4	32(float)	124	(6*)0,(4*)96,(3*)100	(6*)0,(4*)96,(3*)100
roundf	Single-precision `round` of glibc-2.4	32(float)	136	(4*)0,(2*)96,97,(2*)100	(5*)0,(2*)96,98,(2*)100
truncf	Single-precision `trunc` of glibc-2.4	32(float)	120	0,0,06,100,100	0,0,06,100,100

Algorithm InsertBOSCCs (basic block bb, inverse-implies graph iiGr):
  Regions ← ∅
  for each predicate p ∈ iiGr in reverse postorder
    for each region r of consecutive instructions ∈ bb such that \
      instruction i ∈ r is guarded by p or its descendant in iiGr
      Regions ∪ ← {r}
      Profitable[r] ← computeProfitability(r, p, iiGr, Profitable)
  for each predicate p ∈ iiGr in postorder
    for each region r ∈ Regions
      if (Profitable[r]) insert a BOSCC around r

Figure 8. Algorithm to insert BOSCCs.

BOSCCs. When a BOSCC is to be generated within another BOSCC region, it is more accurate to use the conditional probability of executing the inner BOSCC region, given that the outer BOSCC region is executed. To illustrate this point, we revisit the example of Figure 2 where we have two predicates $p1$, $p2$ and a relation $\neg p1 \implies \neg p2$ between them. $p2$ can be true only when $p1$ is true. If we define *percentage of any true superwords* (PATS) as follows,

$$PATS = 1 - PAFS,$$

then

$$\begin{aligned} PATS(p2|p1) &= PATS(p1 \cap p2)/PATS(p1) \\ &= PATS(p2)/PATS(p1), \end{aligned}$$

and we adjust the PAFS of p2 as follows.

$$\begin{aligned} PAFS'(p2) &= 1 - PATS(p2|p1) \\ &= 1 - \frac{1 - PAFS(p2)}{1 - PAFS(p1)} \end{aligned}$$

Intuitively, when we compute the profitability of inserting a BOSCC for p2 within a BOSCC region for p1, we need to use the percentage of p2 being false given that p1 is true. For example, if both $PAFS(p2)$ and $PAFS(p1)$ are 90%, $PAFS'(p2)$ will be zero, and most likely we don't want to generate a BOSCC for p2 although its original PAFS is high. This adjustment of PAFS is performed only if any outer BOSCC for an ancestor predicate in the inverse-implies graph is determined to be profitable. Because of this, profitability is computed in the reverse postorder of predicates in the inverse-implies graph. However, we generate BOSCCs in the postorder of the nodes in the graph for implementation convenience.

The algorithm shown in Figure 8 consists of two loopnests. The first loopnest looks for the largest predicate region r that can be bypassed based on a predicate p and computes the profitability. The predicate region r may contain not only the instructions guarded by p but also the ones guarded by the descendant predicates of p in the inverse-implies graph. Hence a BOSCC generated by this algorithm can bypass a larger number of instructions than the ones generated by the algorithm in the previous approach that does not exploit the inverse-implies relations between predicates. When we count the number of instructions guarded by p for use in computing the profitability, however, we do not count the instructions guarded by the descendant predicates of p to not generate the empty outer BOSCCs when only the innermost BOSCC has nonzero instructions to bypass. For profitability, we use the same equation as in [25] except that we use 2 for the cost of a BOSCC. The second loopnest of Figure 8 performs the code transformation.

6 Experiments

In this section, we apply our compiler to 14 kernels. We describe our implementation, the experimental methodology, the kernels used in the experiments, and the experimental results.

6.1 Implementation

The algorithm described in Section 5 is implemented in a vectorizing compiler that exploits SLP. This compiler was developed by Larsen and Amarasinghe [15] based on the SUIF compiler infrastructure [10] and subsequently extended by Shin et al. [24, 26, 25]. The boxes representing the component algorithms in Figure 4 also closely match the individual passes in our implementation. The input to

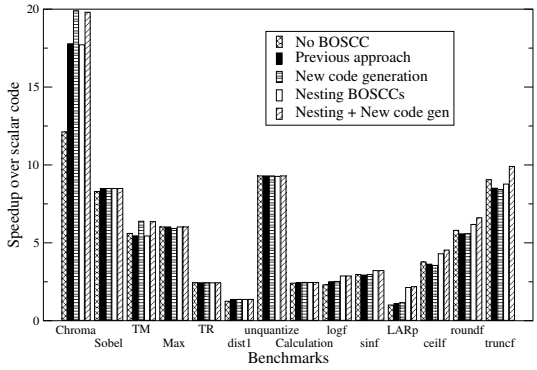


Figure 9. Speedups over scalar code with small data sizes.

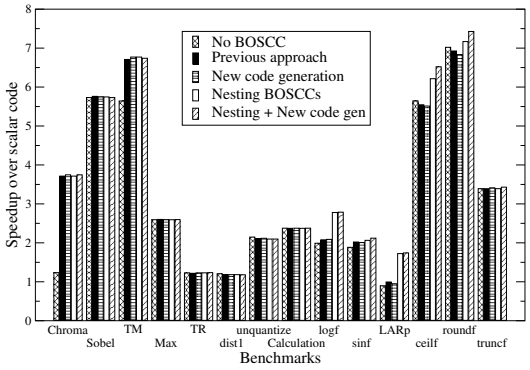


Figure 10. Speedups over scalar code with large data sizes.

the compiler is a sequential code written in C or Fortran, and the output is a C code augmented with vector intrinsics [19].

6.2 Methodology

In order to evaluate our implementation, the 14 kernels shown in Table 1 were automatically vectorized by our compiler and run on a Power Mac G5. The first eight kernels are identical to the ones used in our previous work [26]. In addition to the eight kernels, six others were adopted to study the effects on nested control flow. Five math library functions `logf`, `sinf`, `ceilf`, `roundf`, and `truncf` were taken from GNU math library implementation [9]. To make benchmarks, the library functions were inlined in a loop that computes the function values of uniform random numbers except for `logf`, whose input data is taken from an application implementing the link discovery algorithm [1]. The last two columns of Table 1 show PAFS values or value ranges for the vector predicates in each benchmark. The numbers in parentheses represent the counts of the PAFS values coming next. For each benchmark, we used two data sizes to observe the interaction of the suggested techniques with the memory hierarchy. Figures 9 and 10 show the speedups over the scalar baseline for data sizes that fit in L1 data cache and for data sizes that are larger than the L2 cache, respectively. For each benchmark, speedups of five different versions are measured. The "No BOSCC" version is generated by vectorization but without inserting any BOSCCs. The "Previous approach" version represents the previous approach; uses only single level BOSCCs and does not exploit inverse-implies relations to include the predicate regions associated with the descendant predicates. The "New code generation" version exploits the new code generation technique of Section 3.2 in addition to the "Previous approach" version to isolate its impact on performance. The "Nesting BOSCCs" version uses BOSCCs in a nested fashion and shows the benefit of nesting BOSCCs. The last version, labeled "Nesting + New code gen," combines the two new techniques to show their combined impacts on performance. For all five versions in Figures 9 and 10, vector predicates were restored as described in Section 5.3 after redundancy elimination was performed in a predicate-oblivious fashion.

These kernels are used to show the effectiveness and wide applicability of our techniques and are not intended to be representatives of applications in general. Since the focus of this paper is on efficient vectorization of code with complex control flow, all our benchmarks are vectorizable and have control flow. Furthermore, `unquantize`, `logf`, `sinf`, `LARp`, `ceilf`, `roundf`, and `truncf` have nested if-statements in the loopbody. To compile and run the vectorized output codes, we used gcc 4.0.1 with the -O2 option on a Mac OS X 10.4.7 and a machine that has 32 KB L1 data cache, 512 KB L2 integrated cache, and 8 GB of memory.

6.3 Results

In Figures 9 and 10, the speedups of the first eight benchmarks roughly match or exceed those of [25]. Speedups drop, however, when the data sizes increase. Since vectorization reduces more of the instruction execution time than the memory access time, it is not as beneficial when memory accesses dominate the execution time. For both large and small data sizes of `sobel`, `Max`, `TR`, `dist1`, and `Calculation`, and for the small data of `unquantize`, no BOSCCs are generated because of the low PAFS values (see Table 1) and the small number of instructions to bypass, leading to the similar speedups across all five versions.

Figure 11 shows speedups of the three versions with the new techniques over the "Previous approach" version for small data sizes. `Chroma` and `TM` show distinct speedups when the new code generation technique is added for the "New code generation" version. For the other benchmarks,

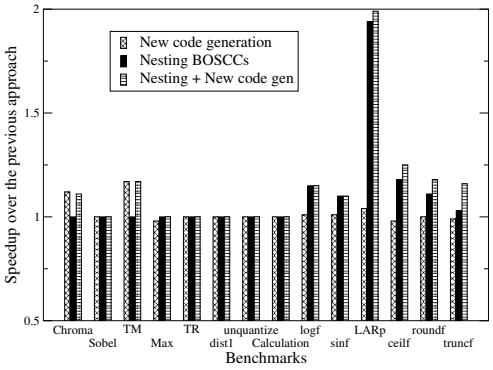


Figure 11. Speedups over "Previous approach" for small data sizes.

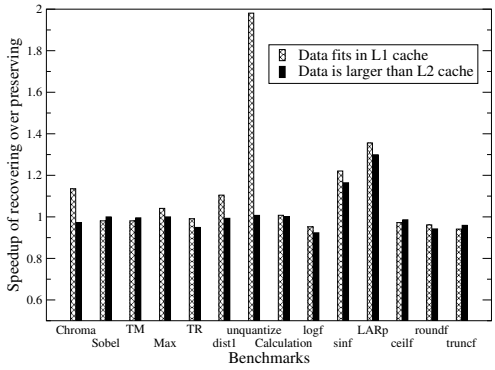


Figure 12. Speedups of predicate restoring over predicate preserving for the "Nesting + New code gen" version.

the "New code generation" versions show almost no performance improvement over the 'Previous approach" versions. For the six benchmarks with no BOSCCs, adding one more instruction to the predicate regions by applying the new code generation technique was not enough to make the BOSCCs profitable. For the other benchmarks, the predicate defining instructions for the `then` part are not included within any BOSCC region because the result of the vector comparison is also used by the instructions guarded by the other predicate, which is associated with the `else` part of the vector comparison. For `ceilf`, `roundf` and `truncf`, applying the new code generation technique made one more BOSCC profitable but the performance benefit was not significant.

For the six rightmost benchmarks from `logf` to `truncf`, the speedups jump when BOSCC nesting is applied. However, only `logf` and `LARp` have nested BOSCCs in the "Nesting BOSCCs" and "Nesting + New code gen" versions. For both benchmarks, the "Nesting + New code gen" versions bypass about 55% of the BOSCCs evaluated by the "Previous approach" version, leading to speedups of 1.15 and 1.99. Although no nested BOSCCs are generated for the other four benchmarks (`sinf`, `ceilf`, `roundf`, and `truncf`), each BOSCC in the "Nesting BOSCCs" version bypasses a larger number of instructions than those in the "Previous approach" versions because the BOSCC insertion pass uses the inverse-implies relation to bypass the additional instructions guarded by the descendant predicates in the inverse-implies graph. For `ceilf`, `roundf`, and `truncf`, the additional speedups are achieved by the "Nesting + New code gen" versions over the "Nesting BOSCCs" versions. Applying the new code generation technique increased the number of instructions guarded by a predicate, which in turn caused the profitability model to generate a BOSCC that is not in the "Nesting BOSCCs" versions. The performance improvement is notable when the new BOSCC is added for the "Nesting + New code gen" versions while it is insignificant when the new BOSCC is added for the "New code generation" versions. This is because the newly added BOSCC could bypass a larger number of instructions when the inverse-implies relations can be exploited. While the "Nesting BOSCCs" versions could have bypassed the similarly large number of instructions, the BOSCC was not generated because we do not count the instructions guarded by the descendant predicates when computing profitability.

In general, the "Nesting + New code gen" version is the best or close to the best across the 14 benchmarks. In particular, eight out of 14 kernels achieve speedups when our two techniques are used. Furthermore, for six out of the seven benchmarks with complex control flow, nesting BOSCCs achieves distinct performance improvement.

6.4 Restoring vs. Preserving Predicates

In Section 5, two approaches to obtain predicates were discussed: restoring the predicates after removing them vs. preserving them. The two approaches interact differently with the intervening optimization passes such as redundancy elimination. If the redundancy elimination pass is applied to the code with predicates, it must take into account the predicates. Since one approach is not always better than the other, in this subsection we experimentally compare the two approaches. Figure 12 shows speedups of the predicate-oblivious redundancy elimination (i.e., restoring predicates) over the predicate-aware one (i.e., preserving predicates) for the "Nesting + New code gen" version for both small and large data sizes. While some benchmarks slowed somewhat, `chroma`, `dist1`, `unquantize`, `sinf`, and `LARp` achieve distinct performance improvement when predicate-oblivious redundancy elimination is used. This experiment suggests that the choice between the two approaches has a

certain impact on performance and that restoring predicates has distinct benefits over preserving predicates for the set of benchmarks we used.

7 Related Work

Since the advent of the conventional vector machines, vectorizing loops with control flow has been a major research subject. One approach is to apply if-conversion, followed by loop distribution for the subsequent vectorization [13].

Recently, many vectorizing compilers have been built for multimedia extension architectures [15, 14, 28, 6, 16, 5, 8, 17]. Two distinct approaches are used to generate SIMD instructions automatically: one based on unrolling to expose *superword-level parallelism* (SLP) [15, 14, 17, 11] and the other based on the vectorization techniques used for the conventional vector machines [5, 28, 8]. Bik et al. use a technique called *bit masking* to combine the definitions of a variable generated from if-statements [5, 4]. Our earlier work describes a technique that can be used to vectorize codes in the presence of arbitrarily complex control flow [26].

A comprehensive survey of vector instructions to support conditional operations is described in [27]. Branch-on-superword-condition-codes (BOSCCs) are supported in AltiVec [19], DIVA [7], SSE [12], and other architectures [3, 2]. The research most relevant to this work is our previous work on generating BOSCCs automatically [25], which generates only single-level BOSCCs. This work improves on our previous work in two ways: first, we generate BOSCCs in a nested fashion, and second, we bypass some predicate-defining instructions by generating all `vec_any_*` instructions. To the best of our knowledge, this paper describes the first technique that can introduce nested control flow into vectorized codes.

8 Conclusion

Vectorization in the presence of control flow has been an important research subject for the past decade, not only because of its benefit, but also because of the difficulty. This paper describes a fundamental technique that can be used to generate efficient vector codes in the presence of complex control flow. Compared to our previous work that generates single-level *branch-on-superword-condition-codes* (BOSCCs), the technique suggested in this paper is capable of generating BOSCCs in a nested fashion so that even BOSCCs can be bypassed by other BOSCCs. In addition, we introduce a technique that can generate all `vec_any_*` BOSCCs to bypass more instructions. We implemented the techniques in a vectorizing compiler. In our experiments with 14 kernels that have control flow, the codes generated by our compiler achieve speedups up to 19.80 over the scalar baseline. Eight out of 14 kernels achieve distinct speedups when the two new techniques are employed. For six out of seven benchmarks with nested if-statements, we observe distinct speedups when BOSCCs are nested and the inverse-implies relations are used to bypass the instructions guarded by the descendant predicates, including 1.99 over the codes generated by the previous approach.

We also compared two approaches to obtain predicates. Restoring predicates achieves distinct speedups over preserving predicates for 5 out of 14 kernels because the redundancy elimination pass can eliminate more redundancies when predicates do not have to be taken into account.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We thank Gail Pieper for proofreading several revisions.

References

[1] Jafar Adibi. Link discovery via a mutual information model: From graphs to ordered lists. In *DIMACS Workshop on Applications of Order Theory to Homeland Defense and Computer Security*, DIMACS Center, CoRE Building, Rutgers University, September 2004.

[2] Krste Asanovic, James Beck, Tim Callahan, Jerry Feldman, Bertrand Irissou, Brian Kingsbury, Phil Kohn, John Lazzaro, Nelson Morgan, David Stoutamire, and John Wawrzynek. CNS-1 architecture specification: A connectionist network supercomputer. Technical Report TR-93-021, International Computer Science Institute, April 1993.

[3] Mladen Berekovic, Hans-Joachim Stolberg, and Peter Pirsch. Implementing the MPEG-4 AS profile for streaming video on a SOC multimedia processor. In *3rd Workshop on Media and Streaming Processors*, Austin, Texas, December 2001.

[4] Aart J. C. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.

[5] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[6] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *The Second*

SUIF Compiler Workshop, Stanford University, August 1997.

[7] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steel, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 26–37, June 2002.

[8] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.

[9] GNU. http://ftp.gnu.org/gnu/glibc/.

[10] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.

[11] IBM. *Exploiting the Dual Floating Point Units in Blue Gene/L*, March 2006. http://www-1.ibm.com/support/docview.wss?uid=swg27007511.

[12] Intel. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.

[13] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *ACM/IEEE Conference on Supercomputing*, pages 407–416, New York, November 1990.

[14] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[15] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, Canada, June 2000.

[16] Ruby Lee. Subword parallelism with MAX-2. *ACM/IEEE International Symposium on Microarchitecture*, 16(4):51–59, August 1996.

[17] Rainer Leupers. Code selection for media processors with SIMD instructions. In *ACM/IEEE Conference on Design Automation and Test in Europe*, pages 4–8, 2000.

[18] Scott A. Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana, IL, September 1996.

[19] Motorola. *AltiVec™ Technology Programming Interface Manual*, June 1999. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

[20] Motorola. *AltiVec™ Technology Programming Environments Manual, Rev. 3*, April 2006. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf.

[21] Dorit Naishlos. Autovectorization in GCC. In *The 2004 GCC Developers' Summit*, pages 105–118, 2004.

[22] Joseph C. H. Park and Mike Schlansker. On predicated execution, May 1991. Software and Systems Laboratory, HPL-91-58.

[23] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[24] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[25] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. In *6th Workshop on Media and Streaming Processors*, December 2004.

[26] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *International Symposium on Code Generation and Optimization*, March 2005.

[27] James E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *International Symposium on Computer Architecture*. ACM, 2000.

[28] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, 2000.

[29] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, August 1996.