

Evaluating Compiler Technology for Control-Flow Optimizations for Multimedia Extension Architectures

Jaewook Shin, Mary Hall and Jacqueline Chame

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{jaewook,mhall,jchame}@isi.edu

ABSTRACT

This paper addresses how to automatically generate code for multimedia extension architectures in the presence of conditionals. We evaluate the costs and benefits of exploiting branches on the *aggregate* condition codes associated with the fields of a *superword* (an aggregate object larger than a machine word) such as the branch-on-any instruction of the AltiVec. Branch-on-superword-condition-codes (BOSCC) instructions allow fast detection of aggregate conditions, an optimization opportunity often found in multimedia applications. This paper presents compiler analyses and techniques for generating efficient parallel code using BOSCC instructions. We evaluate our approach, which has been implemented in the SUIF compiler, through a set of experiments with multimedia benchmarks, and compare it with the default approach previously implemented in our compiler. Our experimental results show that using BOSCC instructions can result in better performance for applications where the aggregate condition codes of a superword often evaluate to the same value.

1. INTRODUCTION

Many modern microprocessors include an expanded instruction set specifically targeting multimedia applications, with a functional unit that operates on aggregate objects to perform SIMD parallel operations on variable-sized fields in the object (*e.g.*, 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than a machine word, they are called *superwords* [12].

Recently proposed parallelization techniques for multimedia extension architectures are based on two distinct approaches. One approach relies on exploiting classical vectorization technology [20, 4, 5, 3, 7]. Another approach, *superword-level parallelization* (SLP), involves packing *isomorphic* instructions and their associated data into superwords, possibly performing loop unrolling to expose parallelism [12, 11, 14]. SLP relies on the observation that multimedia extension architectures support short “vectors”, and that unrolling

to expose parallel operations on objects in a superword can lead to more effective parallelization than the more complex code transformations associated with vectorization. Whether SLP or vectorization is used, parallelization in the presence of control flow is still an open issue for multimedia extensions. Specifically, while SLP is simple and effective, it only identifies parallelism within a basic block. The following inherently parallel loop would not be parallelized:

```
for (i=0; i<16; i++)  
  if (a[i] != 0)  
    b[i]++;
```

The limited instruction-set support for control-flow constructs leads to unique challenges in generating efficient parallel code for multimedia architectures.

Recently, we have developed compiler technology to support SLP in the presence of control flow for both the PowerPC AltiVec [17] and a research architecture called DIVA [8, 6]. In [19], we describe a general approach, which uses superword *select* operations to combine the fields of data computed on multiple control flow paths, as discussed in the next section. In this paper, we examine an optimization which can be used in some cases to improve the performance of our general approach using *branch-on-superword-condition-codes* (BOSCC) instructions. BOSCC instructions allow the fast detection of aggregate conditions often found in multimedia applications. The benefits of BOSCC instructions depend on properties such as the *density* of true or false branches, the number of instructions within a branching construct and the data set size. This paper presents a compiler algorithm and optimizations for parallelization in the presence of control flow using BOSCC instructions, and experimental results that illustrate the tradeoffs associated with these optimizations.

The remainder of the paper is organized as follows. Section 2 illustrates the problem using two alternative versions of SLP code for the PowerPC AltiVec. Section 3 describes experiments on synthetic data that provide insight into the tradeoff space associated with using BOSCC instructions. Section 4 presents the compiler analysis and the algorithm for code generation. Section 5 describes our compiler implementation and a set of performance measurements on seven multimedia computations. Section 6 discusses related work and Section 7 concludes the paper.

2. BACKGROUND

In this section we discuss how to exploit SLP in the presence of control flow using special instructions supported by multimedia extension architectures. First, we briefly show how the SLP compiler parallelizes a simple loop *without* conditional statements, using the example C code in Figure 1(a). Figure 1(b) shows the first step which is unrolling the loop to expose superword-level parallelism in the loop body [12]. An unroll factor of four is selected based on the assumption that the superword register width is sixteen bytes and the array type sizes are four bytes. In this case, the unroll factor is the same as the *superword size*(SWS), which we define as the number of data elements in a machine superword. Next, the parallelizer packs together isomorphic instructions and the resulting code is shown in Figure 1(c).

```
for(i=0; i<1024; i++){
    C[i] = A[i] + B[i];
}
(a) Original
```

```
for(i=0; i<1024; i+=4){
    C[i] = A[i] + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
    C[i+3] = A[i+3] + B[i+3];
}
(b) Unrolled
```

```
for(i=0; i<1024; i+=4)
    C[i:i+3] = A[i:i+3] + B[i:i+3];
(c) Parallelized
```

Figure 1: Example: Superword level parallelization

```
for(i=0; i<1024; i++){
    if(fore[i] != 255)
        back[i] = fore[i];
}
(a) Original
```

```
for(i=0; i<1024; i+=4){
    v255 = (255,255,255,255);
    v_pT = fore[i:i+3] != v255;
    back[i:i+3] = select(back[i:i+3], fore[i:i+3], v_pT);
}
(b) Parallelization using select
```

```
for(i=0; i<1024; i+=4){
    v255 = (255,255,255,255);
    v_pT = fore[i:i+3] != v255;
    branch-on-none(v_pT) L1;
    back[i:i+3] = select(back[i:i+3], fore[i:i+3], v_pT);
    L1:
}
(c) Parallelization using BOSCC
```

Figure 2: Example: Two approaches for SLP in the presence of control flow

We now consider superword-level parallelization in the presence of control flow. We base our discussion on the instruction set archi-

```
3232 = SELECT( 2222, 3333, 1010 );
```

Figure 3: Merging two superwords using a select instruction

ture of the PowerPC AltiVec; other multimedia extension architectures typically support similar instructions. Figure 2(b) shows a parallel version of the code in Figure 2(a), where both control flow paths are executed with SLP exploited in each path. The values from each path are selected and merged to form the superword result. This approach is based on a superword `select` operation that selects individual fields from two superword definitions according to the value of a superword predicate variable. Concretely, the effect of the `select` operation “`dst = select(src1, src2, mask)`”, is to assign `src2` to `dst` for the bit-fields where the corresponding mask bit is 1. Otherwise, `src1` is assigned to `dst`(Figure 3). In the example, the superword predicate variable is represented as `v_pT`, and is generated based on the result of a superword compare instruction.

The code in Figure 2(c) takes advantage of a common instruction supported by multimedia extensions, *branch on superword condition codes*(BOSCC), which checks the aggregate value of the condition codes associated with each field of a superword predicate. For example, a branch-on-none instruction can be thought of as an *AND* of the condition codes of all fields of a superword, that is, a branch is taken if none of these condition codes is true. The superword predicate is derived from the superword condition codes resulting from the previous superword compare operation.

The code in Figure 2(b) suffers from the cost of always executing both control flow paths and the extra `select` instruction, which may offset the benefits of parallelism. In Figure 2(c) these overheads may be significantly reduced if the expression associated with the BOSCC is false most of the time.

In the remainder of this paper we describe the tradeoff space in selecting between these two approaches for SLP in the presence of control flow. The next section shows results of a synthetic benchmark to illustrate this tradeoff space, followed by an algorithm and experimental results from our compiler implementation.

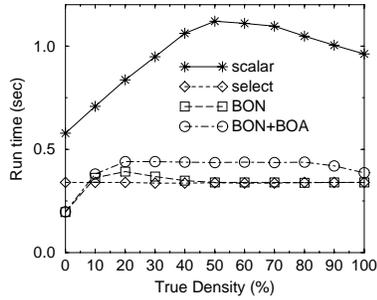
3. THE CHARACTERISTICS OF BOSCC

To gain insight into the factors influencing the profitability of BOSCC instructions, we performed a series of experiments using the following synthetic benchmark.

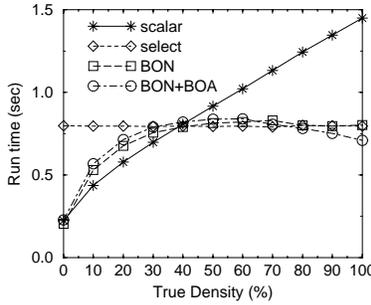
```
for(i=0; i<datasize; i++){
    temp = A[i];
    if (temp == B[i])
        C[i] = temp + D[i];
}
```

In this code, whenever the condition `(temp == B[i])` evaluates to false, the code following the conditional is bypassed. Thus, a BOSCC branch is most profitable when the condition evaluates to false. Profitability therefore depends on the *true density* of the predicate, the frequency of true values for the branch test. We expect that low true densities should correspond to more benefit from BOSCC instructions.

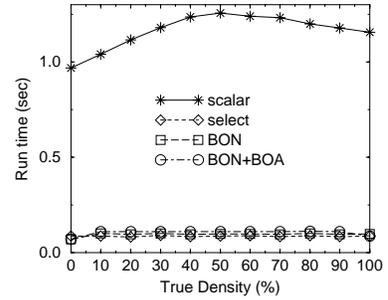
We present the results of a set of experiments in the three graphs from Figure 4. In each graph, the horizontal axis corresponds to



(a) SWS is 4 and data set size is 16 KB.



(b) SWS is 4 and data set size is 128 MB.



(c) SWS is 16 and data set size is 4 KB.

Figure 4: Run Time of Synthetic Kernels

the true density of the input data set. We used a random number generator to create data sets with true densities from 0% to 100%.

Each graph shows the execution time of four versions of the code, as a function of true density. The *scalar* curve represents the execution time of the original scalar code. The other three versions were hand-coded in C extended with the Motorola-Altivec programming model. The *select* version corresponds to what would be generated by the default approach in our compiler, as shown in Figure 2(b) and described in [19]. The *BON* version was derived by adding a *branch-on-none*(*BON*) instruction to the assembly code of the *select* version to bypass the code guarded by the conditional when the test on all fields evaluates to false, similar to the example in Figure 2(c). Finally, the *BON+BOA* version was derived by adding a *branch-on-all*(*BOA*) to the *BON* version. The branch-on-all permits an additional optimization which avoids the select operation; if all fields are known to evaluate to true, then the value of all fields of the corresponding superword of C are the result of the operation guarded by the conditional.

In Figure 4(a) and (b), the superword size (SWS) is four, that is, each superword can hold four integer array elements. Therefore the amount of available parallelism in a superword operation is four. Figures 4(a) and (b) show the run times of the benchmark for two data sizes: in (a) the data size fits in the L1 cache and in (b) the data size is larger than the L2 cache.

First, we consider the results of Figure 4(a). The *scalar* curve is consistently slower than the various parallel versions. It performs best when the true density is either very low or very high. This is because the G4’s branch prediction is most effective when the branching behavior is consistent. In the *select* version, the branch is eliminated and replaced with a merge of fields across the different control flow paths. For this reason, the execution time is the same regardless of the true density. It has the best performance among the four versions for true densities at or above 20%. The performance of the *BON* version is best for true densities near 0% and is the same as the *select* version for true densities above 40%. Interestingly, we see that the slowest performance is at a true density of 16%, also related to branch prediction accuracy. It is lower than 50% because the branch-on-none is taken only when the conditions for all four consecutive scalar comparisons are false. For a superword size of four and true density of D , the probability for all four conditions to be false is $(1 - D)^4$. When two BOSCC instructions are used for the *BON+BOA* version, the overhead of an additional branch overcomes any benefit.

The results of Figure 4(b) show how the tradeoff space is affected when the data footprint exceeds the L2 cache size. As the computation becomes memory bound, the benefits of parallelization become less significant. Thus, the performance gap between the *scalar* and parallel versions is reduced. For true densities below 40%, the *scalar* version is actually the fastest. The *BON* version behaves similarly to the *scalar* version for low true densities, while it behaves similarly to the *select* version for higher true densities. The *BON+BOA* version has the best performance for very high true densities.

To evaluate the effects of increasing the amount of available parallelism, in Figure 4(c) shows the impact of modifying the data type to *char*, thus increasing the superword size to 16. This change increases the performance gap between the scalar version and the other parallel versions for all values of true densities. The various parallel versions exhibit very similar behavior.

From the experiments shown in Figure 4, we can summarize the following conclusions. The BOSCC versions incur an overhead due to the addition of branches as compared to the *select* version, and sometimes this overhead makes them unprofitable. For this reason, we have decided in our compiler to use just one BOSCC instruction, comparable to the *BON* version. We have also determined that low true density can be used as one predictor of profitability. In addition, the profitability of the *BON* version over the *select* version increases as the cost of the instructions in the branch body increases. Also, as parallelism increases, the profitable true density range of the *BON* version actually decreases. While not shown in these experiments, a related profitability criteria is how many instructions appear in the code bypassed by the branch; more instructions lead to greater benefit. Finally, the cost of memory access instructions can dwarf the benefits of parallelizing the computation, but the *BON* version performs comparably to the best version for all true densities. In general, while not always the best performing version, the *BON* version has behavior that is comparable to the best version for all of the experiments, whereas both the *scalar* and *select* versions sometimes are much slower than the others. Based on the insights presented in this section, we build a model which can be used to guide the generation of BOSCC instructions only when profitable.

4. ALGORITHM

In this section we present the compiler analysis and code generation techniques used in our approach. We assume that parallelization has been performed and *select* instructions are inserted where

control flow paths merge, and focus on using BOSCC to reduce the overheads introduced by parallelization of multiple control flow paths. The main components of the algorithm are: a profitability model for BOSCC instructions; a profiling phase for collecting data for the BOSCC model; identifying regions of code and predicates associated with a BOSCC instruction; and code generation for inserting BOSCC instructions.

4.1 BOSCC model

The BOSCC model determines the profitability of using a BOSCC instruction to bypass code, allowing the compiler to decide whether or not to generate a BOSCC instruction. The model uses two key properties of the code to determine profitability. The first, *PAFS* (*percentage of all false superwords*), is the percentage of superword predicates where all fields are false, and indicates how frequently a BOSCC branch is taken. Determining the *PAFS* value associated with a particular superword predicate must be done dynamically, and is computed in a separate profiling phase as discussed in Section 4.2. The second, *NBI* (*number of bypassed instructions*), is the number of instructions bypassed when a BOSCC branch is taken, which represents the number of instructions for a single execution of the parallelized code. The *NBI* can be computed statically by the compiler.

The number of instructions of the *select* and BOSCC versions are Equation 1 and 2 respectively, and a BOSCC instruction is profitable whenever $NI(\text{select}) > NI(\text{BOSCC})$.

$$NI(\text{select}) = NBI \quad (1)$$

$$NI(\text{BOSCC}) = NBI + 1 - PAFS \times NBI \quad (2)$$

In Equation 2, we add an additional instruction for the BOSCC branch, and subtract the number of instructions skipped by the BOSCC branch ($PAFS \times NBI$). In reality, the cost of executing a BOSCC instruction may be higher or lower than that of other instructions depending on how the branch predictor performs. The additional weight of executing BOSCC instructions can be varied to improve the precision of the model, but since it is machine-specific, we omit it here.

Note that this model takes into account the effects discussed in the previous section of the data type size and associated parallelism, as well as the amount of computation bypassed by the BOSCC instruction. However, it ignores locality effects, which must be addressed separately.

To provide intuition as to why parallelization using BOSCC is more profitable than scalar execution of the equivalent code, let us assume that a scalar instruction is mapped to a single equivalent superword instruction and that the run time is computed as the number of executed instructions. In this specific situation, we can have a parallelized code using a BOSCC where each instruction is the superword counterpart of the scalar instruction in the original. The BOSCC can be thought of as the counterpart of the original scalar branch. If the branch body is executed in the scalar version more than once out of SWS iterations, the branch body in the BOSCC version will be executed exactly once for SWS scalar iterations. In this case, the version using BOSCC will run faster than the scalar version because of less loop overhead. If the branch body is not executed in the scalar version for SWS iterations, the branch body in the BOSCC version also will not be executed and will run faster because of less loop overhead.

4.2 Profiling Support to Compute PAFS

Algorithm INSTRUMENT

Given a basic block B

```
P ← find superword predicates(B)
if (P == ∅) return
Insert a basic block counter to B
for each superword predicate pred ∈ P
    Insert a counter for pred
```

(a) Algorithm

```
vec = vec_ld(i_0, ptr);
*(_basicblock + 0) = *(_basicblock + 0) + 1;
vec118 = vec_ld(i_0, ptr133);
vec119 = vec_ld(i_0, ptr134);
vec121 = vec_cmpeq(vec, vec120);
vec123 = vec_cmpeq(vec118, vec120);
vec125 = vec_cmpeq(vec119, vec124);
vec126 = vec_and(vec121, vec123);
vec127 = vec_and(vec126, vec125);
vec129 = vec_cmpeq((vector unsigned char)vec127, vec120);
vec130 = vec129;
sel = vec_ld(i_0, ptr135);
vec138 = (vector bool char)vec_splat_u8(0);
instrument = vec_all_eq(vec130, vec138);
if (instrument == 1)
{
  *(_superword_predicates + 0) = *(_superword_predicates + 0) + 1;
}
sel = vec_sel(sel, vec, vec130);
```

(b) Example

Figure 5: Automatic instrumentation to compute PAFS in profiling phase.

The *PAFS* value in the previous model is determined using automatic instrumentation in a separate profiling phase¹. Figure 5 (a) shows the simple algorithm for inserting instrumentation code. First, for each basic block, all superword predicates are identified. Next, for each basic block that contains superword *select* instructions, we measure the total number of times the block is executed and, for each predicate, the number of BOSCC's taken. To increment the counter only when the superword predicate contains false values in all the fields, we also use a BOSCC instruction. Use of BOSCC expedites the profile run as compared to checking the individual fields in a sequential loop. An example of instrumented code is shown in Figure 5 (b). The instructions in bold font are added for profiling.

4.3 Identifying BOSCC predicates

Prior to code generation, the compiler locates predicates associated with *select* instructions and identifies the set of instructions guarded by each predicate. The third operand of each superword *select* instruction, as defined in Section 2, represents a predicate.

The algorithm to extend these predicates to other instructions is shown in Figure 6. Initially, a null predicate is associated with all instructions. The algorithm in Figure 6(a) scans the code to locate *select* instructions. For each *select* instruction whose

¹While profiling has limitations in deriving dynamic information, particularly when a different input data set is used than was used in the profiling stage, we forgo more elaborate approaches for deriving dynamic information on-the-fly, since issues of deriving dynamic information are orthogonal to the focus of this work. Other approaches could also be used to derive the value of PAFS.

Algorithm ISP(B): Given a basic block B

```
// Initially, all instructions are associated with null predicates
for each select instruction I: “dst = select(src1, src2, pred)” ∈ B
  where dst == src1
// src1 is associated with ‘true’ value of pred
// src2 is associated with ‘false’ value of pred
predicate(I) ← pred;
IdentifyBranchBody(src2, I, pred);
IdentifyMemoryAccesses(src1, dst, pred);
```

(a) Identifying superword predicates

Algorithm IdentifyBranchBody(src, I, pred):

```
Given an operand src, an instruction I and a predicate pred
rd ← reaching definitions of src;
if (rd is not a single reaching definition ∨
    I is not the only use of rd) return;
predicate(rd) ← pred;
for each source operand src of rd
  IdentifyBranchBody(src, rd, pred);
```

(b) Identifying branch body

Algorithm IdentifyMemoryAccesses(src, dst, pred):

```
Given operands src, dst and a predicate pred

rd ← reaching definitions of src
u ← uses of dst
if (rd is single reaching definition ∧ rd is a load ∧
    u is the only use ∧ u is a store ∧
    rd and u access the same address)
  predicate(rd) ← pred
  predicate(u) ← pred
```

(c) Identifying unnecessary memory accesses

Figure 6: Algorithm to identify a predicate for instructions

first source operand and the destination operand are the same, it associates the predicate found in the third source operand with the select instruction, and then follows use-def and def-use chains to locate other instructions to which this predicate can be associated. Two sets of instructions are considered, as shown in Figures 6(b) and (c).

The goal of the algorithm in Figure 6(b) is to identify the set of instructions that are executed only when the predicate evaluates to true. The result of a superword select instruction is the first operand (`src1`) when the predicate `pred` contains all false values. We can therefore bypass any instructions that define the value of the second operand `src2` if all the fields of `pred` are false. This set of instructions can be thought of as the branch body from the original program, although it could include an even larger set of instructions. The algorithm `IdentifyBranchBody` then recursively follows the definitions of the variables contributing to the value of `src2`. Those that have a single definition reaching a single use can be guarded by the predicate `pred`, and can be bypassed by the BOSCC instruction. The goal of the algorithm in Figure 6(c) is to eliminate unnecessary memory accesses occurring when all fields of `pred` evaluate to false. If a load to `src1` and a store of `dst` occur in the code, the value is not modified between the load and store, and no other instructions depend on this load and store, both memory accesses can be predicated with `pred`. The algorithm in Figure 6 guarantees that at most one predicate is associated with each superword instruction.

4.4 Code Generation

Figure 7(b) shows the main algorithm to insert BOSCC instructions. After the predicate for each instruction is identified, instructions with the same predicate are combined into a BOSCC region if there are no intervening dependences. In the algorithm shown in Figure 7(a), the initial BOSCC regions are formed by finding consecutive instructions guarded by the same predicate. Then the BOSCC regions associated with the same non-null predicate are merged if no data dependences with the intervening instructions prevent the code motion. The algorithm first checks if the later region can be moved to the end of the earlier region. If this is not possible because of the data dependences with the intervening instructions, the algorithm checks if the earlier region can be moved before the first instruction of the later region. The goal is to form the largest possible region guarded by a single BOSCC predicate. The number of adjacent instructions guarded by the same predicate provides the value of *NBI* for the BOSCC model, while the value of *PAFS* is derived from profiling. If profitable, a BOSCC instruction is inserted just prior to the instructions that form a BOSCC region, and it branches to the instruction immediately following the last instruction of the BOSCC region.

5. IMPLEMENTATION AND EXPERIMENTS

This section describes our SUIF implementation of the algorithm presented in Section 4, and presents an experimental evaluation of our approach. The experimental performance data was obtained by using our implementation to automatically perform superword-level parallelization on a set of kernels from multimedia applications.

5.1 SLP Implementation

We have implemented the algorithms of Section 4 in the SUIF compiler [9]. The implementation, shown in Figure 8, is based on our extension of Larsen and Amarasinghe’s SLP compiler [12] to exploit SLP in the presence of conditionals, denoted ISI-SLP [19].

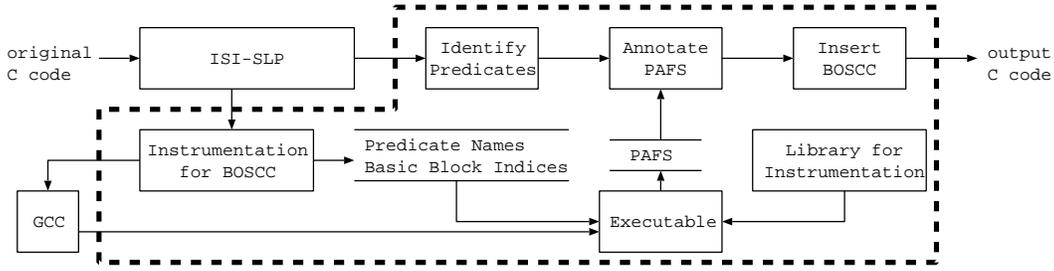


Figure 8: An SLP-based compiler that supports BOSCC.

Algorithm FBR(B): Given a basic block B

```

n ← 0
Region[0] ← new region(NULL)
current ← NULL
prev ← NULL
for each instruction I ∈ B
  pred ← predicate(I)
  if (current ≠ pred)
    Region[n].end ← prev
    n++
    Region[n] ← new region(pred)
    Region[n].moved ← false
    Region[n].begin ← I
    current ← pred
  prev ← I
Region[n++].end ← I

for (i=1; i<n; i++)
  for (j=i+1; j<n; j++)
    if (Region[i].predicate ≠ NULL ∧
        Region[i].moved == false ∧
        Region[i].predicate == Region[j].predicate)
      if (Region[j] can be moved after Region[i].end)
        move instructions in Region[j] after Region[i].end
        Region[j].moved ← true
      else if (Region[i] can be moved before Region[j].begin)
        move instructions in Region[i] before Region[j].begin
        Region[i].moved ← true

return Region, n

```

(a) Form BOSCC regions

Algorithm Insert-BOSCC

Given a basic block B

```

B' ← ISP(B)
R, n ← FBR(B')
for (i=1; i<n; i++)
  if (R[i].moved == false ∧ R[i].predicate ≠ NULL)
    NI_select ← # instructions(R[i])
    NI_boscc ← NI_select + 1 - PAFS(R[i]) × NI_select
    if (NI_boscc < NI_select)
      Insert boscc(R[i])

```

(b) BOSCC insertion algorithm main

Figure 7: BOSCC insertion algorithm

The boxes inside the thick dashed line represent the algorithms described in this paper. The input to our compiler is sequential C code and the output is parallelized C code that may contain BOSCC instructions. The compiler runs in two phases. In the first run, it generates instrumented code which is then compiled by an AltiVec-extended GCC [17] and linked to a library that supports the generation of a PAFS file. The *instrumentation for BOSCC* corresponds to the profiling algorithm of Figure 5, and generates a file that relates basic block indices to predicate names. This file is read by the instrumented executable for computing the PAFS values. In the second run, the predicates in the source code are annotated with PAFS values produced in the profiling run. *Identify Predicates* implements the algorithm shown in Figure 6 and *Insert BOSCC* implements the algorithm of Figure 7.

5.2 Experimental evaluation

We performed a set of experiments to evaluate the effectiveness of the two approaches (*select* and *boscc*) presented in Section 4. For the experiments presented in this section, we used seven kernels selected from multimedia applications.

The experiments were performed on a Motorola PowerPC G4 with a 533 MHz MPC7410 processor, an 8-way set-associative 32KB L1 cache and an 2-way set-associative 1MB L2 cache. To compile both scalar and superword codes, we used an AltiVec-extended GCC with the `-O3` option.

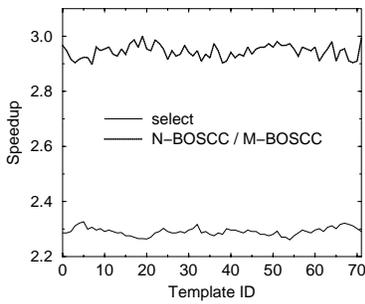
Table 1 shows the seven kernels used in the experiments, each containing at least one conditional. The data widths of the primary objects of each kernel, ranging from 8-bit to 32-bit fields, are shown in the third column. The last two columns describe the two input data sets used in the experiments: the original input data, and a synthetic input designed to yield a particular PAFS value.

Figure 9 shows speedup curves for the kernels in Table 1. Each graph shows the speedups of three parallel versions of a kernel, *select*, *N-BOSCC* and *M-BOSCC*, with respect to the sequential version of the kernel. The *N-BOSCC* (*Naive BOSCC*) version is derived by inserting a BOSCC instruction in all possible BOSCC regions. In the *M-BOSCC* (*Model-based BOSCC*) version, the model described in Section 4 is used to evaluate the profitability of inserting BOSCC instructions. All three versions of each kernel were derived automatically using our SUIF-based implementation.

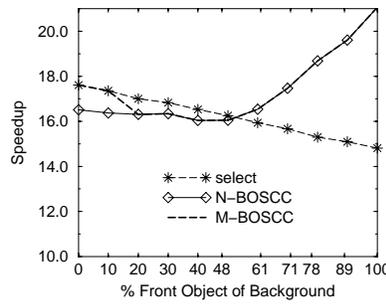
Figure 9(a) shows the speedups of TM for each of the 72 templates of the kernel's input data set, for versions *select*, *N-BOSCC* and *M-BOSCC*. The speedup of *N-BOSCC* varies with the input data sets, since the true density varies from template to template. The *M-BOSCC* version also has a BOSCC instruction for all templates,

Name	Description	Data Width	Input Size (original)	Input Size (synthetic)
Chroma	Chroma keying	8-bit character	48 × 48 color image (12 KB)	400 × 5 color image(12 KB)
Sobel	Sobel edge detection	16-bit integer	1024 × 768 gray scale image (3 MB)	N/A
TM	Image correlation	32-bit integer	64 × 64, 72 32 × 32 (1.4 MB)	16 × 64, 1 16 × 32(10 KB)
Max	Max value search	32-bit float	2 100 × 256 × 256 (52 MB)	2 8 × 256 (16 KB)
transitive	Shortest path search	32-bit integer	2 1024 × 1024 nodes (8 MB)	N/A
MPEG-dist1	dist1 of MPEG2 encoder	8-bit character	data for the first 1000 calls (11 MB)	N/A
EPIC-unquantize	unquantize_image of unepic	16/32-bit integer	reference input (393 KB)	N/A

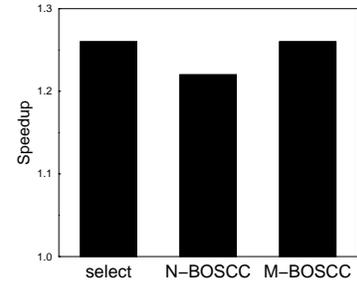
Table 1: Benchmark programs



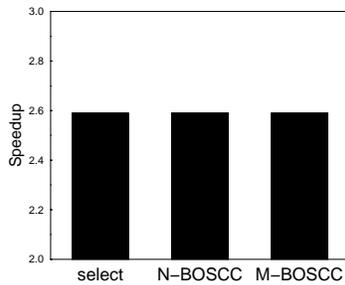
(a) TM



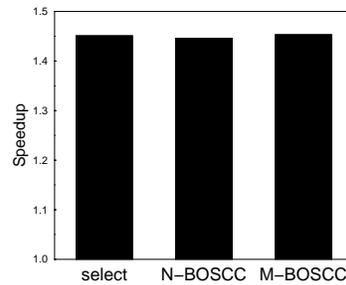
(b) Chroma



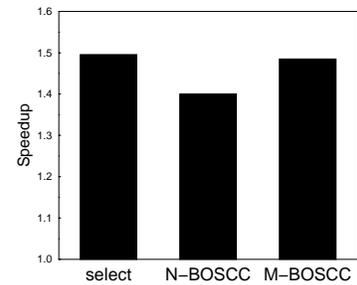
(c) Max



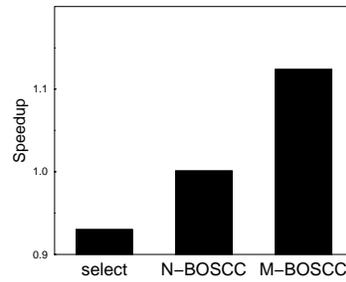
(d) Sobel



(e) transitive



(f) MPEG-dist1



(g) EPIC-unquantize

Figure 9: Speedups over scalar version for real data

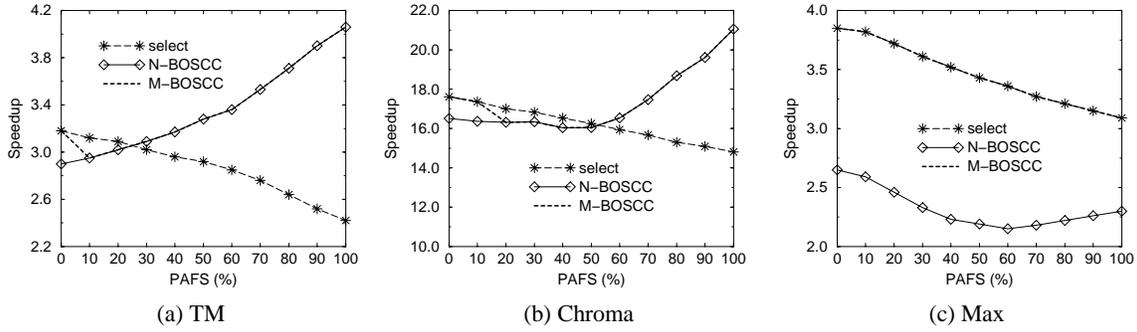


Figure 10: Speedups over scalar version for randomly generated data

and therefore the speedups are the same as those of *N-BOSCC*. Figure 11 shows that the speedup curve of the BOSCC versions closely matches the percentage of taken BOSCC branches of each template. Although not shown in the figure, the speedups of *select* follow the inverse of the percentage of taken BOSCC branches, because the run time of the sequential baseline is affected by the PAFS while that of *select* is not.

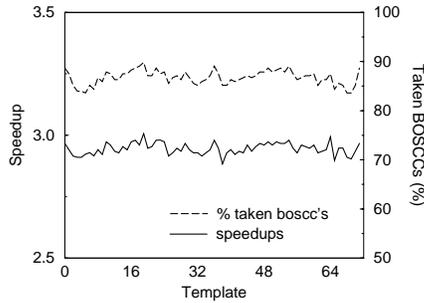


Figure 11: TM: % taken BOSCCs

The speedups of the parallel versions of Chroma-keying are shown in Figure 9(b). The horizontal axis corresponds to the ratio between the sizes of the foreground object and the background image in the input data set (both the size and shape of the foreground object affect the true density of the input data). Since in Chroma-keying a BOSCC branch is taken when all pixels in a superword are outside the foreground object, the speedups corresponding to smaller foreground objects are larger, as expected. In *select*, the runtime does not vary with the true densities, but there is a small speedup due to the fact that in the sequential version the body of the conditional is executed more often as the true density increases. *M-BOSCC* follows the better of the *select* and *N-BOSCC* speedups for most input data sets. The few exceptions are caused by a simplification in our model, where we assume that the cost of executing a BOSCC instruction is the same as any other instruction. In general, branch instructions cost more than arithmetic and logical instructions as the percentage of the taken BOSCCs approaches 50%. The BOSCC model makes the right decisions around 0% and 100% but it tends to make wrong decisions in between the two ends when the margin is small.

For MAX, the input data set was derived by running the TOM-CATV benchmark (from which the kernel is extracted) and collect-

ing the input data to the MAX kernel. The speedups of MAX are 1.26 for *select* and 1.22 for *N-BOSCC*, as shown in Figure 9(c). In *N-BOSCC*, each BOSCC body contains a single instruction, the *select* instruction shown below.

```
max = select(max, new_value, compare);
```

We expected GCC to generate a BOSCC instruction for the region associated with the *select* instruction. However, the GCC version we use generates code such that the *select* instruction is always executed and a new *copy* instruction is added after the BOSCC, possibly because the destination variable (*max*) is live across the iterations of the innermost loop. Thus *N-BOSCC* has two extra instructions, a BOSCC instruction and an extra copy instruction, resulting in a slow down with respect to *select*. When this problem is corrected manually at the assembly level by removing the copy instruction and moving the BOSCC ahead of the *select* instruction, the new *N-BOSCC* performs better than *select*.

For the *N-BOSCC* version of *Sobel*, a BOSCC instruction is generated for four BOSCC regions containing 2, 2, 1, and 1 instructions, respectively, yielding the same performance as the *select* version. The PAFS for each BOSCC region are 17%, 4%, 2% and 82% respectively. The high and low values of PAFS have reduced the cost of BOSCC instructions. Also, large latency of memory references have played a role in this result by overlapping the BOSCC latency. When memory latency is reduced by reducing the data size, *N-BOSCC* slows down by 10% with respect to *select*. No BOSCC instructions are generated for the *M-BOSCC* version. The speedups of the parallel versions with respect to the sequential baseline are 2.59 for all three versions, as shown in Figure 9(d).

For *Transitive*, *N-BOSCC* performs slightly worse than *select*, as shown Figure 9(e), again because the only BOSCC region in the kernel contains a single instruction. In addition, since the BOSCC instruction is never taken, the hardware branch predictor performs well.

The *N-BOSCC* version of *MPEG-dist1* has 16 BOSCC instructions, generated for 4 basic blocks. Each BOSCC region consists of two instructions, and the PAFS ranges from 30 to 40% for all BOSCCs increasing their costs. Thus the *M-BOSCC* version does not have BOSCC instructions.

EPIC-unquantize, shown in Figure 9(g) is interesting because the *M-BOSCC* version outperforms both *select* and *N-BOSCC*. While

N-BOSCC has seven BOSCC instructions, *M-BOSCC* has only four BOSCCs, associated to the four BOSCC regions with the highest number of instructions and PAFS. As a result, while *select* performs worse than the baseline and *N-BOSCC* achieves a negligible improvement, the *M-BOSCC* version speeds up by 12 %.

To further investigate how the performance of the *M-BOSCC* versions varies with the input data set, we used a random number generator to derive synthetic data sets with PAFS from 0% to 100% for TM, Chroma-keying and MAX. Figure 10 shows the speedups of the *select*, *N-BOSCC* and *M-BOSCC* parallel versions of these three kernels. For all three kernels, the speedup of *select* decreases as the PAFS increases, because the sequential version performs better when the scalar branches are taken more often. In general, *N-BOSCC* runs increasingly faster than the sequential version as the PAFS increases. This is because the *N-BOSCC* versions skip superword instructions, each of which corresponds to SWS scalar instructions. Mild slopes in the lower half of the PAFS range are due to the branch prediction mechanism of the machine. Finally, *M-BOSCC* usually performs as well as the better of the two other versions except for a small range of PAFS values, again due to our model's simple assumption for the cost of a branch.

6. RELATED WORK

There are several prior works on automatic parallelization for multimedia extensions [12, 11, 20, 4, 13, 3, 7, 14]. Two distinct approaches are used, that is, SLP [12, 11, 14] and an adaptation of vectorization technique [3, 20, 7]. Conventional parallelization technique for conditionals has been documented in [3, 20]. Bik and et. al. use a technique called *bit masking* to combine two definitions. However, their method is limited to singly nested conditional statements [3]. Our previous work describes the SLP techniques in the presence of control flows [19]. To exploit SLP for conditionals, we borrow many techniques developed for instruction level parallelism [18, 15].

Branch on superword condition code (BOSCC) is supported in AltiVec G4 [17], DIVA [6], and other architectures [2, 1]. *movemask* instruction in Pentium can also be used for a similar purpose to BOSCC [10]. However, no prior work describes generating BOSCC instructions automatically to reduce parallelization overhead of conditionals. Vector flag population count instruction [16] can be used to change the control flow similar to BOSCC instructions in vectorized programs. However, the probability of taken BOSCCs decreases exponentially to the vector length and the long vector length of vector machines reduces the chances for the profitability of BOSCC instructions dramatically.

7. CONCLUSION

This paper has described key concepts in optimizing control flow constructs for multimedia extension architectures. In many multimedia ISAs, including the PowerPC AltiVec, parallel code in the presence of control flow can utilize *select* instructions to combine multiple definitions along different control flow paths. We discussed how to optimize this type of code using a special instruction that examines superword condition codes to bypass unnecessary computation when the entire superword associated with a control flow test has all false values. We have described an implementation and presented a set of results that pinpoints the tradeoff space associated with these two alternate versions of superword code in the presence of control flow.

8. ACKNOWLEDGEMENTS

Authors would like to thank Samuel Larsen and Saman Amarasinghe at MIT for providing their SLP implementation. Especially, Samuel Larsen deserves a special thanks for his tremendous support. Also, we wish to thank Mark Stephenson at MIT for providing the initial implementation of Park and Schlansker's RK-algorithm. This material is based on research sponsored by AFRL and NSA under agreement number FA8750-04-1-0265. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and NSA or the U.S. Government.

9. REFERENCES

- [1] 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.
- [2] 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 Stream Processors*, December 2001.
- [3] 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.
- [4] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.
- [5] Derek J. DeVries. A vectorizing suif compiler: Implementation and performance. Master's thesis, University of Toronto, 1997.
- [6] 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.
- [7] 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, USA, June 2004.
- [8] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.
- [9] Mary W. Hall, Jennifer M. Anderson, Saman 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, 1996.
- [10] Intel. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.
- [11] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.
- [12] Samuel Larsen and Saman Amarasinghe. Exploiting

- superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.
- [13] Ruby Lee. Subword parallelism with MAX2. *ACM/IEEE international symposium on Microarchitecture*, 16(4):51–59, August 1996.
- [14] 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.
- [15] Scott A. Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana IL, September 1996.
- [16] David Martin. Vector extensions to the MIPS-IV instruction set architecture (The V-IRAM Architecture Manual), March 2000. Revision 3.7.5, <http://iram.cs.berkeley.edu/isa.ps>.
- [17] Motorola. *AltiVec Technology Programming Interface Manual*, June 1999.
- [18] Joseph C. H. Park and Mike Schlansker. On predicated execution, May 1991. Software and Systems Laboratory, HPL-91-58.
- [19] 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.
- [20] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.