Boosting the Performance of Multimedia Applications Using SIMD Instructions*

Weihua Jiang[1,2], Chao Mei[1], Bo Huang[2], Jianhui Li[2], Jiahua Zhu[1], Binyu Zang[1], and Chuanqi Zhu[1]

[1] Parallel Processing Institute, Fudan University,
220 Handan Rd, Shanghai, China, 200433
{021021073, 0022704, jhzhu, byzang, cqzhu}@fudan.edu.cn
[2] Intel China Software Center, Intel China Ltd,
22nd Floor, No. 2299 Yan'an Road (West), Shanghai, China, 200336
{weihua.jiang, bo.huang, jian.hui.li}@intel.com

Abstract. Modern processors' multimedia extensions (MME) provide SIMD ISAs to boost the performance of typical operations in multimedia applications. However, automatic vectorization support for them is not very mature. The key difficulty is how to vectorize those SIMD-ISA-supported idioms in source code in an efficient and general way. In this paper, we introduce a powerful and extendable recognition engine to solve this problem, which only needs a small amount of rules to recognize many such idioms and generate efficient SIMD instructions. We integrated this engine into the classic vectorization framework and obtained very good performance speedup for some real-life applications.

1 Introduction

Multimedia extensions (MME), e.g. Intel MMX/SSE/SSE2 [13][14], Motorola AltiVec [21] etc, have become an integral part of modern processors. They enable the exploitation of SIMD parallelism in multimedia applications. These SIMD ISA include not only simple SIMD arithmetic instructions (addition, subtraction etc) but also many domain-specific SIMD instructions to accelerate multimedia typical operations, e.g. saturated arithmetic, which are widely used in multimedia applications.

However, these MMEs have been underutilized so far due to the immaturity of compiler automatic vectorization support. Programmers are largely restricted to time-consuming methods such as inline assembly or intrinsic functions [16].

Many researches have been conducted in automatic vectorization for MMEs. Most of them have regarded this utilization as a similar problem with the vectorization for

* This research supported by: NSF of China (60273046), Science and Technology Committee of Shanghai, China (02JC14013) and Intel-University Cooperation Project (Optimizing Compiler for Intel NetBurst Microarchitecture).

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 59 – 75, 2005.

vector machines [7][11][15][16]. However, these two problems have different key points [3]. Traditional vectorization [1][2] focuses on how to transform source code into vector form correctly, while the utilization of MMEs shall concentrate on how to automatically recognize and then vectorize MME supported idioms in multimedia applications [3]. There are so many idioms needed to be recognized that an efficient and general way is of critical importance. Some researchers exerted efforts in this direction, the typical work is Bik *et al* [12]'s use of tree-rewriting technique to recognize two kinds of operations, saturation and MAX/MIN. Simple rigid pattern match methods [6] and specific languages (e.g. SWARC [17]) have also been used.

In this paper, we solve the key problem by introducing a powerful and extendable recognition engine and integrating it into the classic vectorization algorithm [1] as an extra stage. In this stage, we first normalize the program representation. Then we use an extended Bottom-Up Rewriting System (BURS) [9] to decide possible vectorization plans (VP) for each statement. Based on these single-statement VPs, we find out multi-statement vectorizable idioms and their VPs. Finally, we determine the best VP set for the loop. Experimental results show that we can vectorize many operations in real-life multimedia applications and the performances are quite satisfactory. We achieved a 10.86% average speedup for Accelerating Suite of Berkeley Multimedia Workload [4][5]. Compared with the vectorization ability of Intel C Compiler (ICC) version 8 [10], our compiler outperforms it by about 8% on average.

In short, this paper offers: (1) a uniform and flexible engine to recognize many vectorizable idioms, (2) a mechanism to generate efficient code for vectorized idioms and (3) very good performance for several real-life multimedia applications.

The rest of paper is organized as follows: In section 2, we show the key points in utilizing MMEs. After briefly introducing related techniques in section 3, we present our algorithm and discuss it in detail in section 4. In section 5, experiment results are presented. Then comparisons between our research and previous works are made in section 6. Finally, we end this paper by drawing conclusions in section 7.

2 Key Points in Fully Utilizing MMEs

MMEs include more domain-specific instructions than vector processors [3]. Table 1 lists those in Intel MMX/SSE/SSE2. The corresponding operations are heavily used in real-life multimedia applications. And after manual vectorization, they contribute to almost all speedups [4][5]. This fact shows that the recognition of these ISA-supported multimedia typical operations should be the focus of compiler support for MMEs.

As a result of more domain-specific instructions on MMEs, more idioms need to be recognized than traditional vectorization. Furthermore, because of lacking direct support in high-level languages, programmers often have to use multiple statements to express a multimedia typical operation. In such case, statements composing it are often connected by complex control structure. Sometimes, the statements may even not be adjacent. This fact greatly increases the number of idioms needed to be recognized. For example, Fig. 1 gives three typical idioms to express the signed short saturated operation in C language. Therefore, it is impractical to use the traditional 1:1 special treat

Table 1. Multimedia typical arithmetic instructions provided by MMX/SSE/SSE2

Function	Instruction
Saturated add	PADDSB/PADDSW/PADDUSB/PADDUSW
Saturated subtract	PSUBSB/PSUBSW/PSUBUSB/PSUBUSW
Saturated pack	PACKSSWB/PACKSSDW/PACKUSWB
Sum of absolute difference	PSADBW
Min/max	PMINUB/…/PMINSW/PMAXUB/…/PMAXSW
Average	PAVGB/PAVGW
Multiply and add	PMADDWD
Logical	PAND/PANDN/PXOR/POR
Compare	PCMPEQB/PCMPGTW/…/PCMPGTB

```
/* short a, b; int ltmp; */
#define GSM_SUB(a, b) \
   ((ltmp=(int)a-(int)b)\
   > MAX_WORD ? MAX_WORD:\
   ltmp < MIN_WORD ? \
   MIN_WORD: ltmp

t = GSM_SUB(a, b)
```

(a)

```
/*short *a,*b,*c;
 int t;*/
t = a[i] - b[i];
if(t>32767||t<-32768){
   if(t>32767)
      c[i] = 32767;
   else
      c[i] = -32768;
} else  c[i] = t;
```

(b)

```
/*float sum; int clip;
short *sample; */
if(sum>32767.0) {
   *samples = 32767;
   clip++;
}elseif(sum<-32768.0)
{ *samples = -32768;
   clip++;
}else  *samples = sum;
```

(c)

Fig. 1. Three variations of signed short saturated operation

ment to recognize each idiom as the number of idioms is now largely increased. It follows that a uniform and flexible way to recognize them is much preferred.

3 Background

3.1 Classic Vectorization Algorithm

The classic algorithm for automatic vectorization [1] is illustrated in Fig. 2. Its main idea is to reorder and vectorize statements in the loop according to data dependence.

```
for each loop in source code {
  construct its data dependence graph.
  condense each maximal strongly connected component in the graph.
  topological sort the condensed graph and number the nodes
  (1..m).
  for i = 1 to m {          // code generation
    distribute node_i into a loop.
    if node_i is not strongly-connected   //not in a dep cycle
      or can be recognized as vectorizable idiom then
        vectorize node_i.
  }
}
```

Fig. 2. Classic vectorization algorithm

Because its objects mostly are simple arithmetic operations, it pays little attention to code pattern and ISA support. For other few important idioms in numerical applications, e.g. MAX/MIN, it uses special treatment to recognize them.

3.2 Bottom-Up Rewriting System (BURS)

Bottom-Up Rewriting System (BURS) [9] is a code generator's generator, which is widely used in compiler to help generate code from IR tree. For a certain IR tree and a set of tree patterns, there may be more than one match (covering) of the tree. BURS uses dynamic programming to choose the lowest cost one. It accepts rules in the form of **non-terminal→pattern (cost) [action]** and produces tree matchers that make two passes over each subject tree. The first bottom-up pass finds a set of patterns that cover the tree with minimum cost. The second top-down pass executes the actions associated with minimum-cost patterns at the nodes they matched, which is driven by the goal non-terminal at tree root (similar with the *start* symbol in LR parsing).

According to the grammar in Fig. 3(a), tree *FETCH(PLUS(REG,INT))* has two coverings, namely, rule tree 1(4(6(5(2,3)))) and 1(4(8(2))) with costs 5 and 2, respectively.

In the first traversal of the BURS tree matcher, the tree is labeled as Fig. 3(b), in which each node is associated with minimum cost matching rule set for this subtree and corresponding costs. The best covering 1(4(8(2))) is indicated by the goal non-terminal *goal* at tree root.

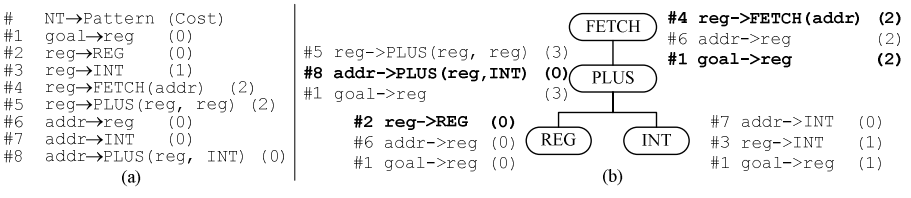

Fig. 3. Example BURS Matches. Action of each rule is omitted

4 Compiler Support for MMEs

To solve the idiom recognition problem in compiler support for MMEs, we design a powerful recognition engine and add it as an extra stage at the beginning of the classic algorithm in Fig. 2. The enhanced algorithm, as a whole, works as follows: our engine deals with the recognition of vectorizable language constructs (simple statements and idioms) in each loop and the dependence relations within each construct. Besides, the engine decides vectorization plan (VP) for each construct, i.e. the way to generate its SIMD code. Then it comes to the classic algorithm part that handles other issues, e.g. dependence relations between these constructs and other statements. During dependence graph construction, the statement(s) in each construct share one node

in the graph. After graph condensing, we discard those recognized constructs that are strongly connected. For those survived, SIMD code is generated according to their VPs.

4.1 Basic Ideas in Vectorizable Construct Recognition

A vectorizable construct is a code block in source code to express one or several related vectorizable operations as a whole. It can be transformed into efficient SIMD code. Recognizing such constructs and finding how to vectorize them i.e. their VPs are the two tasks the recognition engine needs to accomplish. These two tasks are closely related. To recognize such construct, we have to know whether it is supported by MME's ISA and whether its vectorized form is profitable. Only during deciding its VPs can such information be obtained. To find how to vectorize a construct, i.e. its VP, we have to use a series of patterns (rules) to match the construct. Thus, a VP actually is a rule set covering the construct and VP finding is a process of recognition. Therefore, we prefer to perform these two tasks together.

During the recognition of single-statement vectorizable construct, VP selection is needed since a construct may have more than one VP. E.g. on Intel SSE/SSE2, construct *c[i]=(a[i]+b[i]+1)>>1;* have two VPs: one is *add/add/shift/store* instruction sequence and the other is instructions *average/store*. If we express each statement as a tree, then BURS is a tool available to find the best one of all the coverings (VPs).

As to the recognition of multi-statement construct, the first key problem is to uniform variations of multimedia operations. As mentioned in section 2, such variations result from the complex control structure. If relations between statements were simplified, the number of variations and the difficulty of recognition would be lowered. Therefore, we first normalize the program representation. IF-conversion [18] is the technique we used to convert control dependence to data dependence. After conversion, statements are flattened and only related by data dependence. Besides, each statement is composed of a statement body and several guard conditions. In this way, it does not matter what statements look like in original source code since they are now all related by data dependence.

Then, we have to find which statements constitute such a construct. Each statement's semantic information and their relations are thus needed. The latter can be obtained from variables' DU chains. The former can be obtained from VPs of statements since certain VP is only linked with certain statement structure. Thus, we reuse the result of single-statement construct recognition (represented by goal non-terminals of VP) here to avoid redundant computation. E.g., we can know that the first statement in Fig. 1(b) can be part of a vectorizable saturated sub construct since its VP shows that it is a signed short subtract operation.

However, not every single statement in a vectorizable multi-statement construct can be vectorized. E.g. though statements in Fig. 1(b) as a whole are vectorizable on MMEs, the second statement itself is not vectorizable. To solve this problem, we regard each statement in a vectorizable multi-statement operation as partially vectorizable and design a goal non-terminal and a VP rule to represent it. After multi-statement construct recognition, these partial VPs will be discarded.

After multi-statement construct recognition, selection is also needed since constructs may have conflicts, i.e. a statement may belong to more than one construct.

To ease the introduction of our recognition algorithm, we define several concepts.

Definition 1. A vectorization plan (VP) for tree t defines a possible way to vectorize the tree. It consists of a series of 6-tuples <*r, vl, vc, sc, def, use*> associated with tree nodes, which means using rule *r* to vectorize the subtree rooted at the node, *vl* is the vector length, *vc* is the amortized vectorized execution cost, *sc* is the sequential execution cost, *def* and *use* mean the definition and use operands respectively.

In our system, VP rules are expressed similarly with BURS rules. In each tuple, field *sc* is used to compare with field *vc* to show whether the VP is profitable. When recognizing a multi-statement construct, we need to know its operands from VPs of its statements. Thus, we add field *def* and *use* here.

As mentioned above, the result of VP can be represented by its goal non-terminal. Thus, for simplicity, we denote the vectorization plan as *VP(t, n)*, in which *t* is the tree and *n* is the goal non-terminal.

To be easy, we encode the VP rule information into its left-side non-terminal. It is named as *[<op>_]<category>[suffix]*$_{[datatype]}$. E.g. $ssub\_expr_{s16}$ denotes the result of using signed short saturated subtract rule to vectorize an expression.

Op	category	Datatype
sub(normal subtract)	*expr*(vectorizable expression)	*s16* (signed short)
ssub(saturated subtract), etc.	*stmt*(vectorizable statement)	*u32*(unsigned int), etc

In our system, we express statements as trees. For each statement, its body and every condition in its guard are expressed as a tree, respectively. For a tree *t*, *t* is vectorizable if $\begin{cases} \exists VP(t,n) \wedge n \in stmt \text{ category} & t \text{ is statememt body} \\ \exists VP(t,n) \wedge n \in expr \text{ category} & t \text{ is guard condition} \end{cases}$.

Definition 2. Best vectorization plan BVP(t, n) is the minimal vectorized cost VP of all VP(t, n).

Definition 3. Candidate vectorization plan set CVP_Set(t) contains all the BVPs for tree t.

If we ignore fields *vl, sc*, *def* and *use* of each tuple in VP, we can find that CVP_Set is just the result set of using BURS and rules to match a tree.

Definition 4. Multi-statement vectorization plan MVP(s_1,s_2,…,s_n) is a vectorization plan for multi-statement construct which is composed of statements s_1, s_2,…,s_n . It is a tree of 6-tuples <*MVP rule, vl, vc, sc, def, use*> (one for each multi-statement operation). Each field has similar meaning as its counterpart in VP tuple.

Our recognition algorithm is shown in Fig. 4. Each main step is discussed below.

```
Normalize program respresention;
for each statement s in loop {
  compute CVP_Set(s.body);
  for each condition expression c in s.guard
    compute CVP_Set(c);
}
find all MVPs in the loop;
select best VPs for loop;
```

Fig. 4. Recognition Algorithm

4.2 Normalize the Program Representation

To normalize the program representation, first, we perform a series of normalization techniques: e.g. scalar expansion, variable range analysis [20], loop rerolling [3] etc.

As mentioned above, we perform IF-conversion [18] in the loop body to reduce variations of multi-statement operations and eliminate complex control flow. It removes branches by replacing the statements in the loop body with an equivalent set of guarded statements. Besides normal statement body, every guarded statement includes a guard (relative to the loop) which is composed of several condition expressions combined by *and* operation. As to nested loops, the inner loop as a whole is regarded as a statement when outer loop is processed. The below code illustrates the conversion for the outer loop.

```
if(guard1)                          if(guard1)
   for(…){                             for(…){
     if(guard2){                S1:      (guard2, guard3)    stmt1;
      if(guard3)                S2:      (guard2)     for(…) stmt2;
          stmt1;          ⇒            }
       for(…)
           stmt2;
}}
```

Thus, variation (a) and (b) in Fig. 1 have the same code sequence as in Fig. 5(a) after scalar expansion and IF-conversion. And variation (c) has the form as in Fig. 5(b).

```
S1:   ( )       t[i]=a[i]-b[i];
S2: (t[i]>32767)  c[i]=32767;
S3: (t[i]<-32768)  c[i]=-32768;
S4:  (t[i]≥-32768,  t[i]≤32767)
                  c[i]=t[i];
```

(a)

```
S1:(sum[i]>32767.0)  samples[i]=32767;
S2:(sum[i]>32767.0)  clip++;
S3:(sum[i]<-32768.0) samples[i]=-32768;
S4:(sum[i]<-32768.0)  clip++;
(sum[i]≥-32768, sum[i]≤32767)
                    samples[i]=sum;
```

(b)

Fig. 5. Saturated Operations after IF-conversion

Generally, IF-conversion can be used to utilize bit-masking instructions on MMEs [3]. Here, we extend its usage. We regard program representation after IF-conversion as a new IR on which our further recognition is based. After recognition, we roll back those converted statements (including their guard) if they are not vectorizable.

4.3 Compute CVP_Sets

As mentioned above, we use BURS [9] to generate CVP_Sets for every statement's body and guard conditions. The first bottom-up pass of BURS matcher is performed here while the second top-down pass is performed at code generation stage.

To use BURS to compute CVP_Set, we define VP rule as extended BURS rule. The extension is the addition of 5 fields: *vl*, *sc*, *def*, *use* and *constraint*. Field *constraint* is added to represent constraints, e.g. dependence issues, data type etc., that are required before applying this rule. Original *cost* field in BURS rule is used as *vc*.

We define costs of VP rule as $\begin{cases} vc \cong (latency\ of\ SIMD\ instrcutions)/(vector\ length) \\ sc \cong latency\ of\ sequential\ instrcutions \end{cases}$.

In its bottom-up traversal, BURS matcher matches a tree and labels each tree node its CVP_Set. In each tuple <*r, vl, vc, sc, def, use*>, cost *vc* and *sc* are set as the sum of VP rule *r*'s *vc, sc* and each subtree's *vc*, *sc*, respectively. Field *def* and *use* are simply set as VP rule *r*'s *def* and *use*.

As to vector length *vl*, its computation is a little bit subtle because vector length of VP rule and that of each subtree may not be equal. E.g. Assuming we have an expression *a[i]+b[i]*, the element type of *a*, *b* is *short* and *int*, respectively. According to the rules in Fig. 6, Rule tree *11(10(2),4)* can successfully match it. Vector length of each rule is 4, 4, 8 and 4, respectively. If vector length of the tree is 4, then part of load result *a[i+4:i+7]* will be discarded. To avoid such waste, we set *vl* as the least common multiply (LCM) of vector length of VP rule and that of each subtree. This means the tree will be executed LCM times in one vectorized loop iteration. Thus, SIMD code generated by VP rule's and subtrees' semantic actions will be duplicated LCM/vl_i times respectively. As to the above example, we will generate code:

```
load a[i:i+7];                                                    #rule 2
load b[i:i+3]; load b[i+4:i+7];                                   #rule 4
convert a[i:i+7] to int vectors a'[i:i+3] and a'[i+4:i+7]; #rule 10
a'[i:i+3]+b[i:i+3]; a'[i+4:i+7]+b[i+4:i+7];                       #rule 11
```

Take the process of computing CVP_Set for statements in Fig. 5(a) as an example. Parts of the related VP rules are shown in Fig. 6. Fig. 7 shows BURS matching result.

#	Rule	vl	vc	sc	def	use	constraint
1	$lval\_expr_{s16}$ →arr[index]	8	0.75	1.5	root		vec_arr(arr,index, s16)
2	$expr_{s16}$ →arr[index]	8	0.75	1.5		{root}	vec_arr(arr,index, s16)
3	$lval\_expr_{s32}$→arr[index]	4	1.5	1.5	root		vec_arr(arr,index, s32)
4	$expr_{s32}$→arr[index]	4	1.5	1.5		{root}	vec_arr(arr,index, s32)
5	$ssub\_stmt1_{s16}$→$lval\_expr_{s32}$ =$expr_{s16}$–$expr_{s16}$	4	0.5	0.5	$lval\_expr_{s32}$	{$expr_{s16}$[1], $expr_{s16}$[2]}	no_dep(def, use[1]) no_dep(def,use[2])
6	sub_stmt_{s32}→$ssub\_stmt1_{s16}$	4	0	0	$ssub\_stmt1_{s16}$.def	$ssub\_stmt1_{s16}$.use	
7	$upplimit\_expr_{s32}$→$expr_{s32}$>32767	4	0.5	0.5		{$expr_{s32}$}	
8	$lowlimit\_expr_{s32}$→$expr_{s32}$<-32768	4	0.5	0.5		{$expr_{s32}$}	
9	$ssub\_stmt2_{s16}$→$lval\_expr_{s16}$=32767	8	0.5	0.5	$lval\_expr_{s16}$		
10	$expr_{s32}$→$expr_{s16}$	4	0.5	0.5		{$expr_{s16}$}	
11	$expr_{s32}$→$expr_{s32}$ + $expr_{s32}$	4	0.5	0.5		{$expr_{s32}$[1],$expr_{s32}$[2]}	

Fig. 6. Some VP Rules. Function *vec_arr(arr, index, type)* checks if *arr[index]* is a vectorizable continuous array visit expression with element type as *type*. Function *no_dep(a, b)* checks if there is no dependence between *a* and *b*

The final work before MVP finding is to compute sequential cost *sc* of each statement (including its body and guard). It is set as the sum of each part's lowest *sc* and the fixed instruction latency for *if* statement dispatch.

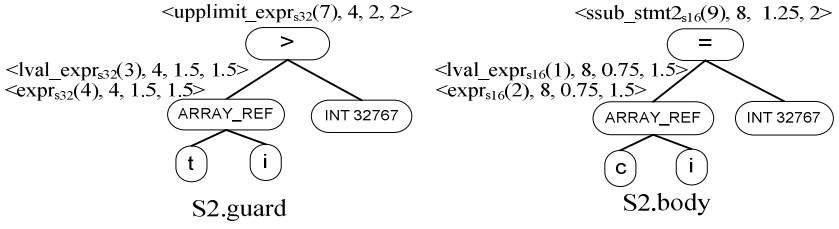

Fig. 7. CVP_Set Computation Result for S2 in Fig. 5(a). The VP rules for S1, S3 and S4 and their match processes are similar, hence omitted. Their results are:
CVP_Set(S1.guard) = ∅, CVP_Set(S1.body) = {<$ssub\_stmt1_{s16}$,…>, <sub_stmt_{s32},…>},
CVP_Set(S3.guard) = {<$lowlimit\_expr_{s32}$,…>}, CVP_Set(S3.body)={<$ssub\_stmt3_{s16}$,…>},
CVP_Set(S4.guard.condition1)={<!$upplimit\_expr_{s32}$,…>}, CVP_Set(S4.guard.condition2) = {<!$lowlimit\_expr_{s32}$,…>}. , CVP_Set(S4.body) = {< $ssub\_stmt4_{s16}$,…>}

4.4 Find All MVPs

Based on CVP_Sets, we now begin to find all MVPs in the loop according to predefined MVP rules. We define each MVP rule as *<non-terminal→Stmt_Set, vl, vc, def, use, constraint, action>*. Set *Stmt_Set* contains statements (represented by goal non-terminals to indicate their roles). Other fields have the same meaning with their counterparts in VP rules. As an example, Fig. 8 shows some MVP rules.

NT	Stmt_Set	vl	vc	def	use	constraint
$ssub\_stmt_{s16}$	{s1: () $ssub\_stmt1_{s16}$, s2: (upplimit_$expr_{s32}$) $ssub\_stmt2_{s16}$, s3: (lowlimit_$expr_{s32}$) $ssub\_stmt3_{s16}$, s4: (!upplimit_$expr_{s32}$, !lowlimit_$expr_{s32}$) $ssub\_stmt4_{s16}$}	8	0.5	s4.body.def	s1.body.use	DU_Chain(def(s1.body)) ≡{s2.guard, s3.guard, s4.guard, s4.body.right} def(s2.body)≡def(s3.body) def(s2.body)≡def(s4.body)
$stmt_{s16}$	{s1: () $ssub\_stmt_{s16}$}	8	0	s1.def	s1.use[1].parent	
$bitmask\text{-}ing\_stmt_{s16}$	{s1: ($expr_{s16}$) $stmt_{s16}$ s2: ($expr_{s16}$) $stmt_{s16}$}	8	0.75	s1.body.def	{s1.guard.use, s1.body.use, s2.body.use}	def(s1.body)≡def(s2.body) s1.guard≡!s2.guard

Fig. 8. MVP Rules Expamples. Action part of each rule is straightforward, thus omitted. E.g. action of the last rule will generate code (in form of ICC intrinsic function): "xmm1 = _mm_and_si128(use[1], use[2]); xmm2 = _mm_andnot_si128(use[1] , use[3]); xmm3 = _mm_or_si128(xmm1, xmm2); _mm_store_si128(def, xmm3)"

Since a MVP rule represents a vectorizable operation, statements in it as a whole constitute a simpler semantic expression than their respective original expressions. As a result, we use the action of MVP rule and those of operands in *use* and *def* to generate code, instead of using the actions in each statement's CVP_Set. Semantic actions of operands will be executed before that of MVP rule.

We find out all MVPs by constructing a VP DAG. In this DAG, every node represents a possible MVP rule match or BVP. Edge represents the inclusion relationship between MVP node and corresponding VP nodes.

The construction algorithm is as follows:

1) Construct initial nodes. For each statement in the loop, we create a node for every element in the Cartesian product (every possible combination) of CVP_Sets of statement body and conditions in the guard.
2) Find a MVP rule match. It means to find a node set that meets the MVP rule, i.e. node set is an instance of the *Stmt_Set* (possible with additional guards) and constraints are satisfied. For each found MVP rule match, we construct a new node for it and make the node set as its children.
3) Repeat step 2 until no new MVP rule match can be found.

When a MVP rule successfully matched, a MVP node is created and annotated as *<MVP rule, vl, vc, sc, def, use>*. Field *def* and *use* are set according to the *def* and *use* field of MVP rule. E.g. the MVP rule match node for Fig. 5(a) has {a[i], b[i]} as *use* and c[i] as *def*. Field *vl* is set as the least common multiply of MVP rule's *vl* and *vl* of each operand (*def* and *use*). Field *vc* is set as the sum of MVP rule's *vc* and *vc* of each operand. Field *sc* is set as the sum of *sc* of each original statement it included.

For example, for statements in Fig. 5(a), we first construct node 1 to 5 in Fig. 9 according to their CVP_Set. Then, after matching saturated subtract MVP rule, we

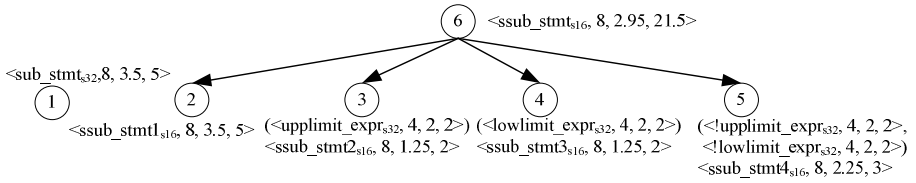

Fig. 9. MVP DAG for Fig. 5(a). The *def* and *use* field in each node is ignored for simpilicity

construct node 6. Thus, all the MVP matches have been found and shown in this DAG. For statements in Fig. 5(b), similar DAG can be created.

When matching MVP rules, it may be found that the statements constituting the MVP match node have additional guard conditions besides the ones needed by the MVP rule. If different statement has different additional conditions, this MVP rule cannot be matched because these statements actually are embraced by different *if* statements. However, if there is only one additional condition, we decompose each node representing the related statement that hasn't such condition into two node and continue the match. E.g. assuming there are two nodes: *(guard)S1* and *S2*. *S1* and *S2* constitute a MVP match *M* if *guard* does not exist. In such case, we decompose *S2* into two new nodes: *(guard)S2* and *(!guard)S2*. Then *(guard)S1* and *(guard)S2* are matched as a MVP node *(guard)M*. Such decomposition is performed on DAG. Only when the related MVP is finally chosen will the decomposition really be performed on statements. Such decomposition may increase the number of possible MVPs. However, our goal of this step is to find all the possibilities. Next step will choose from them the best ones. Thus, such treatment is harmless.

Fig. 10(a) introduces an example to show the MVP recognition process. It is slightly modified from a code segment (Fig. 10(b)) in ADPCM Decoder to make it vectorizable. The original version is similar to a saturated operation, but not vectorizable because *valpred* is a reduction variable and has different type from *vpdiff*. This test case is hard to recognize because saturated subtract operation is mingled with the saturated add in that they share the clip statement (the second *if* statement). We have not found any previous research work that could deal with it. However, our approach can vectorize

```
if(sign[i])
  t[i]=valpred[i]
       -vpdiff[i];
else
  t[i]=valpred[i]
       +vpdiff[i];
if(t[i]>32767)
  valpred[i]=32767;
else if (t[i]<-32768)
  valpred[i]=-32768;
else valpred[i]=t[i]
```

(a)

```
if(sign)
  valpred-=vpdiff;
else
  valpred+=vpdiff;
if(valpred>32767)
  valpred=32767;
else
if(valpred<-32768)
  valpred=-32768;
```

(b)

```
S1:(sign[i])
    t[i]=valpred[i]-vpdiff[i];
S2:(!sign[i])
    t[i]=valpred[i]+vpdiff[i];
S3:(t[i]>32767)
    valpred[i]=32767;
S4:(t[i]<-32768)
    valpred[i]=-32768;
S5:(t[i]≤32767,t[i]≥-32768)
     valpred[i]=t[i]
```

(c)

Fig. 10. Code Segment in ADPCM Decoder

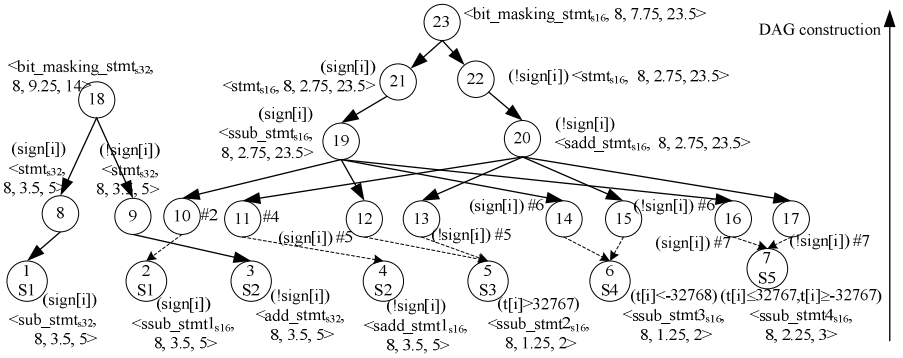

Fig. 11. Match Result for Fig. 10(c). Guards are shown as real conditions instead of VPs for easy understanding. The *def* and *use* field in each node is ignored. Dotted line is used to show the node decomposition process. For example, node 12 and 13 are decomposed from node 5(S3 of Fig. 10(c)). So, node 13 is *(!sign[i], t[i]>32767) <ssub_stmt2$_{s16}$, 8, 1.25, 2>* ,which represents statement: *(!sign[i], t[i]>32767) valpred[i] = 32767;*. Solid line shows the inclusion relationship. E.g. MVP *sadd_stmt $_{s16}$* (node 20) needs 4 statements (node 11, 13, 15, 17) to constitute. Therefore, they are connected by directed lines

this example very well. Its normalized code is shown in Fig. 10(c). According to the MVP rules in Fig. 8 and some other similar MVP rules, we can get the MVP matching DAG in Fig. 11. It clearly shows our system's power.

4.5 Select VPs for the Loop

After constructing the DAG, the problem now becomes how to select for each statement its best VP because each statement can only be vectorized using one VP.

First, we decompose the DAG into a series of connected components (trees) (ignoring dotted lines). For example, DAG in Fig. 11 is decomposed as following trees: {1, 3, 8, 9, 18}, {2}, {4}, {5}, {6}, {7}, {10–17, 19–23}. Each tree as a whole shows how to vectorize a construct.

Then, we delete those trees whose root satisfies any of the following conditions: 1) root.VP is only meaningful as part of MVP; 2) root.guard ≠ ∅. It means vectorized code shall be embraced by a vectorized *if* statement (not bit-masking operation) which is impossible; 3) root.sc ≤ root.vc. Such MVP is not profitable. After it, DAG in Fig. 11 has trees {1, 3, 8, 9, 18} and {10–17, 19–23} left. The former will vectorize statements S1-S2 while the latter will vectorize S1-S5. However, S1-S2 can only be vectorized by one VP tree. Thus, these two trees are incompatible.

We try to find the compatible tree subset with maximum weight. We define each tree's weight (time save of vectorization) as its root.sc–root.vc. This problem can be formulated as a NP-complete set-covering problem (by using similar technique in [19]). In practice, because conflicts are rare and easy to solve, we use the greedy algorithm: choose the tree that has the most number of statements and lowest cost first. Thus, VP tree {10–17, 19–23} is selected for Fig. 11.

At dependence-graph-construction stage, we make all the statements in each chosen tree share one node in the graph. The dependence relations between these statements are ignored since they are already checked by the constraints of VP/MVP rules.

4.6 Code Generation

In code generation stage, vectorizable nodes in the dependence graph, i.e. those that are associated with a VP tree and not strongly connected, are vectorized on IR in the classic way: first distribute it into a loop; then generate vectorized code according to actions of its VP tree; loop step is set as the *vl* field of the VP tree root; rest loop is generated for un-vectorizable iterations.

Since we regard each VP tree as a high-level operation, it shall contain assignment(s) only to one variable. Thus, the "store" action in each non-root VP node is not performed. E.g. the store operations to variable *valpred* in nodes 19-22 of Fig. 11 are prohibited.

After vectorization on IR, two optimizations: alignment analysis and redundant load/store are performed. Because MMEs prefer aligned data load/store, we need to lay out the arrays and determine the alignment information for each memory access. At present, we only scan alignment requirements for array references and try to meet them as many as possible. For pointer references, we conservatively regard them as unaligned. This strategy seems to work well for our benchmarks.

To reduce the redundant load/store for arrays in vectorized constructs, we perform common sub-expression elimination for loads and dead code elimination for stores.

5 Experimental Results

In this section, we demonstrate the effectiveness of the presented algorithm with experimental results. All experiments were conducted on a 2.8G Pentium 4E Processor and 1G memory system with Redhat 9.0. The benchmark we use is Accelerating Suite of Berkeley Multimedia Workload (ASBMW) [4][5]. We also compare the results of our method with ICC[10], the Intel compiler that has the state of the art vectorization techniques. We use two versions of ICC, *v7* and the latest *v8*, to vectorize the applications. We implement our vectorization algorithm in our C-to-C compiler Agassiz [8] which is a research compiler developed by University of Minnesota and us. Agassiz transforms vectorizable parts of multimedia source code into Intel SSE/SSE2 instruction set (in form of ICC intrinsic functions). The rules we added to our system are the ones we found profitable and general enough in our real-life application study. The output of Agassiz is compiled by ICC 7 and ICC 8 with vectorization off, respectively. Fig.12 lists the results. All results are obtained as the average of 5 runs.

As illustrated by Fig.12, Agassiz achieved an average more than 10% speedup. In contrast, ICC 7 and ICC 8 only achieved 1.94% and 2.37% speedup, respectively. This is because Agassiz can vectorize almost all the constructs ICC can, which mainly are memory copies, arithmetic operations, MAX/MIN operations, etc. Moreover, Agassiz

Application	ICC 7 option1	ICC 8 option1	Agassiz +ICC 7 option2	Agassiz +ICC 8 option2
gsm_decode	-1.88%	-3.74%	18.07%	19.25%
gsm_encode	-0.70%	-15.81%	12.49%	13.13%
lame	7.60%	6.29%	7.49%	6.14%
mesa_gears	-2.22%	0.17%	-1.67%	-2.10%
mesa_morph3d	-0.77%	-0.78%	-0.19%	-0.79%
mesa_reflect	2.21%	1.38%	2.46%	2.25%
mpeg2_decode	1.07%	0.86%	3.05%	1.67%
mpeg2_encode	9.76%	37.69%	43.55%	42.03%
mpg123	3.28%	0.00%	18.90%	18.29%
timidity	1.06%	-2.34%	3.78%	2.87%

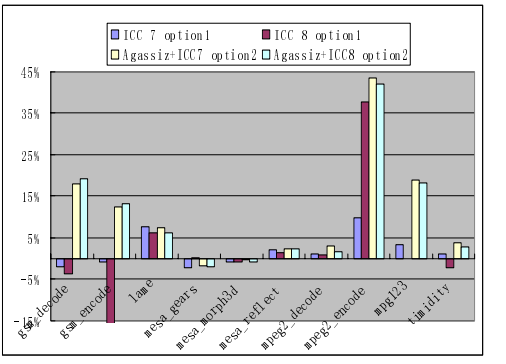

Fig. 12. Speedup of vectorization. ICC compiling option1: *-O2 -xW* (with vectorization turned on); ICC compiling option2: *-O2 -xW -vec-* (with vectorization turned off). The baseline is compiled with option2

can vectorize constructs that ICC cannot. This fact leads to the results that Agassiz outperforms ICC on six applications while the rest have similar performance.

We can also find that ICC 8 generated better scalar code than ICC 7 since Agassiz+ICC 8 had slightly smaller speedup than Agassiz+ICC 7 on most applications. As to the vectorization capability, ICC 8 greatly outperformed ICC 7 on *mpeg2_encode* since it vectorized the Sum of Absolute Difference (SAD) operation. A very strange thing is that, though they vectorized the same parts of *gsm_encode*, ICC 8 greatly slowed down it.

Though Agassiz have vectorized lots of constructs, the most important ones (contributing most to speedup) are just variations of several important operations. As to *gsm_decode* and *gsm_encode*, the most important one is saturated operation. Concerning *lame*, the most important one is MAX operation. As to *mpeg2_encode*, the key operations are SAD operation and float arithmetic operation. As to *mpeg2_decode*, it is saturated pack. To *mpg123*, it is also saturated arithmetic. Regarding *timidity*, it is floating-point operation.

Fig. 13 lists the comparison of two performance monitors after Agassiz+ICC 8 vectorization and its scalar counterparts.

We can see that execution time (clockticks, column 2) is somewhat proportional to the number of dynamic instructions retired (column 3). In the listed applications, the great reduction of dynamic instructions is attributed to the vectorization of multimedia typical operations (mainly multi-statement operations) in hot loops. Thus, these operations contribute to most of the speedups. The rest speedups mainly come from float operation and integer operation of small data type. Other performance monitors such as mis-predicted branches and L2 cache miss etc. have not changed much due to vectorization. Thus, they are not listed here.

	Clockticks	Instructions
gsm_decode	86.47%	62.18%
gsm_encode	88.73%	83.98%
lame	92.77%	85.22%
mesa_gears	98.03%	100.91%
mesa_morph3d	97.18%	100.69%
mesa_reflect	96.96%	89.48%
mpeg2_decode	98.41%	96.46%
mpeg2_encode	68.72%	42.99%
mpg123	84.42%	75.94%
timidity	96.39%	99.68%

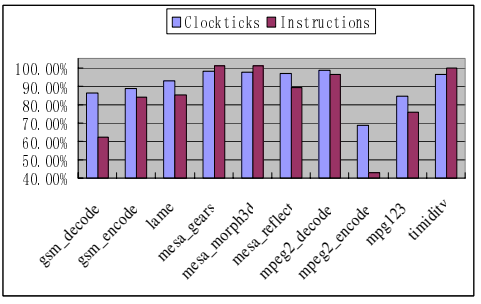

Fig. 13. Performance monitors after vectorization vs. its scalar counterparts

6 Related Work

The application of traditional automatic vectorization techniques on MMEs [11][15][16] and new methods such as SLP [7] neither recognize nor vectorize those important multi-statement idioms in real-life multimedia programs.

Realizing the importance of recognizing multimedia typical idioms, researchers have proposed other methods. Simple pattern match based algorithm [6] requires compiler to have one pattern for each variation of multimedia operations, thus too rigid to be acceptable. Domain-specific language, SWARC [17], is developed to provide a portable way to program for MMEs. But it is not popular enough.

In [12], a preprocessing before classic vectorization algorithm to detect two multimedia typical operations (saturation and max/min) is presented. It uses tree rewriting system to rewrite the tree step by step to recognize them. Speedup was reported for several small kernels and 164.gzip in SPEC2000. In comparison, our algorithm shows more applicability and power. First, our algorithm can recognize almost all kinds of SIMD idioms in a uniform and flexible way. Second, as to the two kind operations this method focuses on, our method puts much less constraints on the recognizable operations. For example, ours does not require the exact order of statements in a multi-statement construct, e.g. statements *s2, s3, s4* in Fig. 5(a) can appear in any order and any other irrelevant statements could be inserted between these three statements. We also allow the multi-statement operations appearing in forms that are more complex. Thus, our method is able to recognize those variations such as Fig. 1(b), Fig. 1(c) and Fig. 10(a) which cannot be handled by [12].

7 Conclusion

In this paper, we first showed that the key difficulty in utilizing MMEs to boost the performance of real-life multimedia applications is how to recognize many different profitable and vectorizable operations, especially how to recognize the variations of the same multimedia typical operation in an efficient and general way.

Then, we introduced a powerful recognition engine to overcome such difficulty, which only needs a small amount of rules to recognize and vectorize many operations in real-life source code. In addition, it can find the best VP set for each loop. Thus, it can fully exploit benefits from MMEs. It also enjoys great extendibility in that we only need to add new operation patterns (rules) into it if new SIMD instructions appear. We integrated this engine into the classic vectorization framework and obtained satisfactory speedup for several real-life multimedia applications.

References

[1] Allen R, Kennedy K. Automatic Translation of Fortran Programs to Vector Form. *ACM Trans. on Programming Languages and Systems*, 1987, 9(4): 491-542.

[2] Padua D, Wolfe M. Advanced Compiler Optimizations for Supercomputers. *Comm. of the ACM*. 1986, 29(12): 1184-1201.

[3] Ren G, Wu P, Padua D. A Preliminary Study On the Vectorization of Multimedia Applications for Multimedia Extensions. *Proc. of the 16th Int'l Workshop on Languages and Compilers for Parallel Computing,* 2003.

[4] Slingerland N, Smith A J. Design and Characterization of the Berkeley Multimedia Workload. *Multimedia Systems,* 2002, 8(4): 315-327.

[5] Slingerland N, Smith A J. Measuring the Performance of Multimedia Instruction Sets. *IEEE Trans. Computers*, 2002*,* 51(11): 1317-1332.

[6] Boekhold M, Karkowski I, Corporaal H. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. *Lecture Notes in Computer Science*, 1999, 1593: 673.

[7] Larsen S, Amarasinghe S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *ACM SIGPLAN Notices*, 2000, 35(5): 145-156.

[8] Zheng B, Tsai J Y, Zhang BY, Chen T, Huang B, Li J H, Ding Y H, Liang J, Zhen Y, Yew P C, Zhu C Q. Designing the Agassiz Compiler for Concurrent Multithreaded Architectures. *Proc. of the 12th Int'l Workshop on Languages and Compilers for Parallel Computing*, 1999: 380-398.

[9] Fraser C W, Hanson DR, Proebsting T A. Engineering Efficient Code Generators Using Tree Matching and Dynamic Programming. TR-386-92, Princeton University.

[10] Intel Corporation. Intel C++ Compiler User's Guide. 2003: http://developer.intel.com/.

[11] Sreraman N, Govindarajan R. A Vectorizing Compiler for Multimedia Extensions. *Int'l Journal on Parallel Processing*, 2000.

[12] Bik A J C, Girkar M, Grey P M, Tian X. Automatic Detection of Saturation and Clipping Idioms. *Proc. of the 15th Int'l Workshop on Languages and Compilers for Parallel Computers,* July 2002.

[13] Intel Corporation. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. 2001: http://developer.intel.com/.

[14] Intel Corporation. Intel Architecture Optimization Reference Manual. 2001: http://developer.intel.com/.

[15] Cheong G, Lam M S. An Optimizer for Multimedia Instruction Sets. *Second SUIF Compiler Workshop*, Stanford, August 1997.

[16] Krall A, Lelait S. Compilation Techniques for Multimedia Processors. *Int'l Journal of Parallel Programming*, 2000, 28(4): 347-361.

[17] Fisher R J, Dietz H G. Compiling for SIMD within a Register. *Workshop on Languages and Compilers for Parallel Computing*, University of North Carolina, August 1998.

[18] Allen J R, Kennedy K, Porterfield C, Warren J. Conversion of Control Dependence to Data Dependence. *Proc. of the 10th ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages,* Austin, Texas, 1983: 177-189.

[19] Liao S, Devadas S, Keutzer K. A Text-Compression-Based Method for Code Size Minimization in Embedded Systems. *ACM Trans. on Design Automation of Electronic Systems, 1999, 4(1): 12-38*

[20] Stephenson M, Babb J, Amarasinghe S. Bitwidth Analysis with Application to Silicon Compilation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation,* June 2000

[21] Fuller S. Motorola's AltiVec Technology. White Paper, May 6, 1998