A Preliminary Study On the Vectorization of Multimedia Applications for Multimedia Extensions

Gang Ren
University of Illinois at Urbana-Champaign
gangren@uiuc.edu

Peng Wu
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
pengwu@us.ibm.com

David A. Padua
University of Illinois at Urbana-Champaign
padua@cs.uiuc.edu

ABSTRACT
In 1994, the first multimedia extension, MAX-1, was introduced to general-purpose processors by HP. Almost ten years have passed, the present means of accessing the computing power of multimedia extensions are still limited to mostly assembly programming and the use of system libraries and intrinsic functions. Because of the similarity between multimedia extensions and vector processors, it is believed that traditional vectorization can be used to compile multimedia extensions. Can traditional vectorization effectively vectorize for multimedia extensions? If not, what additional techniques are needed? This paper tries to answer these two questions. Our experiment shows that traditional vectorization is not very effective in compiling multimedia applications for multimedia extensions. Based on a code study of the Berkeley Multimedia Workload, we identified several new challenges arise in vectorizing for multimedia extensions, and provide some solutions to these challenges.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *code generation, compilers, optimization.*

General Terms
Performance, Experimentation, Languages.

Keywords
Multimedia Applications, Multimedia Extensions, Subword Parallelism, Vectorization.

1. INTRODUCTION
The past decade has witnessed multimedia processing become one of the most important computing workloads, especially on personal computing systems. To respond to the ever-growing performance demand of multimedia workloads, multimedia extensions (MME) have been added to general-purpose microprocessors to accelerate these workloads [1]. The multimedia extensions of most processors have a simple SIMD architecture based on short, fixed-length vectors, a large register file, and an instruction set targeted at the very specific multimedia application domain.

Although it was almost ten years ago when the first multimedia extension, MAX-1, was introduced by HP, today multimedia extensions are usually programmed in assembly language and using libraries and intrinsic functions [2].

A promising alternative is to compile programs written in high-level languages directly to MME instructions. Because of the similarity between multimedia extensions and vector processors, one may naturally consider applying traditional vectorization techniques to multimedia applications. However, satisfactory results are yet to be obtained for the vectorization of realistic multimedia programs on MME. Our experiments showed that less than 6% of the core loops in the Berkeley Multimedia Workload (BMW) can be vectorized by a state-of-the-art commercial MME compiler.

Therefore, this paper sets out to answer two questions: 1) Can traditional vectorization techniques effectively vectorize for multimedia extensions? If not, 2) what additional techniques are needed? To answer these questions, we conducted a code study on the BMW benchmark, which is a set of multimedia programs written in C [3]. During the code study, we identified the differences between MME and traditional vectorization, and discuss new analyses and transformations to bridge the difference between the two.

The rest of the paper is organized as follows: Section 2 gives an overview of MME architectures and the BMW benchmark. In Section 3, we survey current programming models and existing compiler supports for multimedia extensions. Section 4 discusses the difference between vectorizing for MME and traditional vector machines, and presents solutions to address some of these differences. Section 5 concludes and outlines the future work.

2. BACKGROUND

2.1 Multimedia Extensions (MME)
Because of the increasing importance of multimedia workloads, most major microprocessor vendors have added multimedia extensions (MME) to their micro-architectures. Multimedia extensions that are available today include MMX/SSE/SSE2 for Intel [24][26], VMX/AltiVec for IBM [27], 3DNow! for AMD [25], MAX1/2 for HP [21], VIS for SUN [20], DVI for DEC [22], and MDMX/MIPS-3D for MIPS [23]. Most multimedia extensions are vector units that support operations in fixed-length vectors that are short, typically are no longer than 16 bytes. The purpose of the SIMD design is to exploit the data parallelism inherent in multimedia processing.

Multimedia extensions have evolved rapidly in recent years. Early MMEs often provided very limited instruction sets. For example, the very first multimedia extension, MAX-1, offers only 9 instructions for processing 64-bit vectors of 16-bit integers [21]. Today's MMEs support wider vector length, more vector types, and a much more comprehensive instruction set architecture (ISA). VMX, for instance, supports 128-bit vectors of 8-, 16-, and 32-bit integers, or 32-bit single-precision float with an ISA of 162 instructions [27].

2.1.1 An Example of MME: Intel's SSE2

Announced in 2000 with Pentium 4 processor, SSE2 evolves from SSE (Streaming SIMD Extensions) by incorporating double-precision floating-point support and more instructions [24].

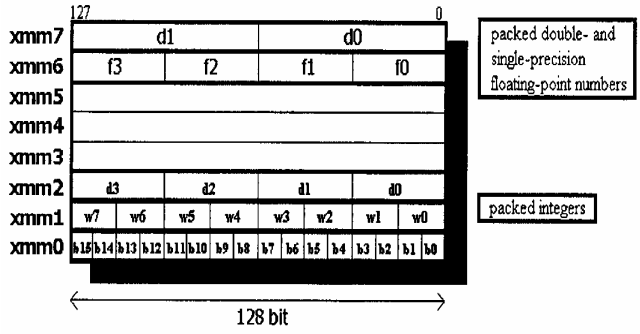

Figure 1. Streaming SIMD Extensions [16].

SSE2 supports 128-bit vectors of almost all data types, including single- and double-precision floating-point and 8-, 16- and 32-bit integers as shown in Figure 1. It provides 144 instructions that can be grouped into arithmetic, compare, conversion, logical, shift or shuffle, and data movement instructions.

SSE2 instruction set is non-uniform. That is, not all vector types are equally supported by the ISA. For example, SSE2 provides max and min operations for vectors of signed 16-bit integers and unsigned 8-bit integers, but not for vectors of other integer types.

2.1.2 Multimedia Extensions vs. Vector Processors

Despite the similarity between multimedia extensions and traditional vector processors, there are three key differences between the two architectures.

First, a multimedia extension instruction only processes a small number of data elements, limited by its register width, often no longer than 16 bytes. This is in contrast with the very long vectors typical of traditional vector machines.

Second, multimedia extensions provide much weaker memory units. For cost reasons, multimedia extensions do not support gather/scatter type memory operations as vector machines usually do. In addition, many multimedia extensions, such as VMX, can access memory only at vector-length aligned boundaries. Others like SSE2 allow misaligned memory accesses, but such accesses incur additional overhead. For example, in SSE2, a misaligned load, involves two loads and the execution of several micro-ops [26].

Finally, multimedia extension ISA tend to be less general-purpose, less uniform, and more diversified. Many operations are very specialized and are only supported for specific vector types. A good example is SSE2's max/min operation mentioned before.

2.2 Berkeley Multimedia Workload

Our code study is based on the Berkeley Multimedia Workload (BMW) benchmark [3]. The BMW benchmark is written in C and evolves from MediaBench [4]. Table 1 lists the BMW benchmark programs.

Table 1. Berkeley Multimedia Workload Applications [3]

Name	Description	# of Line
ADPCM	Audio compression: IMA ADPCM	300
GSM	Audio compression: European GSM	5,473
LAME	Audio encoder: MPEG-1 Layer III	19,704
mpg123	Audio decoder: MPEG-1 Layer III	7,790
DVJU	Image compression: AT&T IW44	25,419
JPEG	Image compression: DCT based	33,714
MPEG2	Video compression: MPEG-2	17,437
POVray	Persistence of vision ray tracer	151,346
Mesa	OpenGL 3D rendering API clone	120,038
Doom	Classical FPS video game	57,868
Rsynth	Klatt speech synthesizer	7,089
Timidity	MIDI music rendering	40,514

One characteristic of multimedia applications is that a few core procedures take up most of the execution time. In fact, the hot-spot behavior is much more pronounced in multimedia workloads than in integer programs or even floating point programs. This characteristic makes multimedia programs suitable for both hand and compiler optimizations. Table 2 gives the execution time distribution of several representative multimedia, integer, and floating-point workloads on a 2.0 GHz Pentium IV processor. In Table 2, Column "#Proc" gives the number of procedures that take up more than 10% of the total execution time, "%Exec" gives the total percentage of execution time spent on these procedures (excluding the time spent on the procedures they call), and "%Line" gives the percentage of total source lines of these procedures.

Table 2. Distribution of Execution Time

Class	Application	#Proc	%Exec	%Line
Image	JPEG (Decoder)	3	77.78	0.9
Audio	GSM (Encoder)	2	73.81	3.0
Graphics	Mesa (Gears)	1	81.34	0.2
Speech	Rsynth	1	70.49	2.7
Integer	255.vortex	2	28.01	0.1
Scientific	172.mgrid	3	79.69	29.23

3. OVERVIEW of MME COMPILATION

3.1 Programming Multimedia Extensions

For a long time, assembly language programming or embedding inline assembly in C programs has been the dominant means to access multimedia extensions. Due to the difficulties in programming, debugging, and maintaining assembly programs, usually only very important processing kernels are off-loaded to multimedia extensions.

As multimedia extensions become more powerful, the need for more efficient programming methods grows in importance. Some computer vendors provide high-level language interfaces to multimedia extensions through intrinsic functions to facilitate accesses to MME. Intrinsic functions embedded in high-level programming languages are translated to MME instructions by native compilers. *Gcc v3.1*, for instance, supports intrinsic functions for several multimedia extensions including AtliVec, SSE2 and 3DNow! [5]. Compared to assembly coding, intrinsic function programming model achieves better productivity, readability, and portability without sacrificing much performance.

Programming in standard high-level languages and relying on the compiler to produce optimized codes offer programmers a much easier way to utilize multimedia extensions. However, this approach can only be feasible if the compiled codes match the performance of the previous two approaches.

3.2 Compilers for Multimedia Extensions

Automatically compiling C programs to multimedia extension instructions have been tried out in both academia and industry.

Because of architectural similarities between vector processors and multimedia extensions, traditional vectorization was naturally considered to compile programs for multimedia extensions. Traditional vectorization techniques were developed for vector processors mainly at the end of the 1980's and the beginning of the 1990's [6]. It is based on the notion of data dependence, which was developed by Kuck and his co-workers at the University of Illinois [7]. An overview of vectorizing compiler technology is given in [8].

In 1997, Cheong and Lam [9] developed an optimizer for VIS, the SUN multimedia extension, based on a SUIF vectorizer from MERL [10]. The focus of this work was to address alignment issues during code generation. Krall and Lelait [11] applied traditional vectorization to the code generation for VIS. Sreraman and Govindarajan [12] developed a vectorizer for the Intel MMX based on SUIF. However, only experiments with small kernels were reported. Larsen and Amarasinghe [18] proposed the SLP algorithm to do vectorization within basic blocks. Instead of vectorizing across loop iterations, SLP algorithm packs isomorphic instructions from the same basic block to vector instructions. The vectorizer was implemented in SUIF and was targeting AltiVec. Speedups were reported on few programs from SPECfp. In [17], a domain-specific C-like language, SWARC (SIMD-within-a-register), was developed to provide a portable way of programming for MMEs.

To date, only a few commercial compilers that support automatic vectorization for multimedia extensions are available. The Crescent Bay Software extends VAST to generate codes for AltiVec extension [13]. The Portland Group offers the PGI® Workstation Fortran/C/C++ compilers that support automatic usage of SSE/SSE2 extensions [14]. The Codeplay™ announces the VectorC compiler for all x86 extensions [15]. Also, Intel extended its own product compiler to vectorize for MMX/SSE/SSE2 [16].

3.3 A Case Study: Intel Compiler for SSE2

In this section, we present our findings of the effectiveness of a state-of-the-art product compiler for MME. Among all the commercial compilers with vectorization support for MME, the Intel compiler is one of the most widely used and available. It is also well documented in both user manuals [26] and research reports [16]. In addition, the Intel compiler can successfully vectorize 73 out of 135 loops from the Callahan-Dongarra-Levine Fortran test suite [19]. This proves that the Intel compiler is a competent vectorizer in the traditional vectorization standard. Therefore, we chose the Intel compiler as the target of the study and SSE2 as our test platform.

In the experiments, we use the Intel compiler to compile core subroutines from the BMW benchmark. Out of 185 loops, only 10 can be fully vectorized. The Intel compiler also reports reasons why vectorization fails at a particular loop. Table 3 lists the number of loops that can and cannot be vectorized and the reasons why vectorization fails.

Table 3. Vectorization in Intel's Compiler

Reasons of Vectorization Failures	# of Loops
Fully Vectorized	10
Outer Loop	53
Irregular Loop Structure	23
Data Dependence	55
Unsupported Instructions	16
Too Complex	20
Others	8
Total Number of Loops	**185**

Since we do not have access to the compiler source codes, it is very hard for us to identify the actual reasons behind these failures. However, our manual analysis of some of the loops mentioned in Table 3 gives us some insight on the difficulties to vectorize multimedia applications.

4. BRIDGING THE GAP BETWEEN MME AND TRADITIONAL VECTORIZATION

Our studies show that despite the success of the vectorization for traditional vector machines, vectorization for multimedia extension still has a long way to go. In this section, we identify the key differences between traditional and MME vectorizations. The difference is the natural result of their differences in programming style (Section 4.1), in common data types and operations (Section 4.2), in application code patterns (Section 4.3), and in the architectures (Section 4.4).

4.1 Difference in Programming Styles

4.1.1 Use of Pointers vs. Arrays

Traditional vectorization is most effective for programs where most cycles are spent on tight loops involving mostly array accesses. Multimedia applications, on the other hand, rely on pointers and pointer arithmetic to access data in computationally intensive loops. Figure 2 gives an example of such pointer accesses extracted from *LAME*, an MPEG audio encoding application from the BMW benchmark suite. In this example, *xp1* and *xpn* point to input buffers, *ep* and *pp* point to output buffers, and are initially passed into the procedure as parameters.

```
for (i=1;i<512;i++) {
  *ep = *xp1 * *xp1 + *xpn * *xpn;
  if (*ep < 0.0005) {
    *ep++ = 0.0005;
    *pp++ = 0.0;
    xpn--;
    xp1++;
  }
  else {
    ep++;
    *pp++ = atan2( -(double)(*xpn--),
           (double)(*xp1++) );
 }
}
```

Figure 2. Pointer Access Example from BMW/LAME.

All twelve programs in the BMW benchmarks use pointers in their core procedures, and six of them also use pointer arithmetic. The pervasive use of pointers and pointer arithmetic has a great impact on vectorization in terms of memory disambiguation and dependence testing.

```
for (i=1;i<512;i++) {
  ep'[i-1]=xp1'[i-1]*xp1'[i-1]+xpn'[1-i]*
xpn'[1-i];
  if (ep'[i-1] < 0.0005) {
    ep'[i-1] = 0.0005;
    pp'[i-1] = 0.0;
  }
  else
    pp'[i-1]=atan2(-(double)(xpn'[1-i]),
(double)(xp1'[i-1]));
}
```

Figure 3. BMW/LAME after Transforming Pointers to Closed-form Expressions

Using Figure 2 as an example, before conducting any dependence analysis, the compiler needs to determine whether there is any overlapping between the *regions* accessed through variables *xpl*, *xpn*, *ep*, and *pp* during the iterations. This is not exactly a pointer aliasing problem. The complication comes from the fact that *xpl*, *xpn*, *ep*, *pp* are changing their values within the loop. A conventional alias analysis may determine whether **xpl* and **ep* are aliased at a particular iteration, but not whether **xpl at any iteration* may be aliased to **ep at any other iterations.*

One may observe that *xpl*, *xpn*, *ep*, and *pp* change their values in a regular way. Not only does each variable change monotonically (either increasing or decreasing), but also their values change by a constant per iteration. In fact, *xpl*, *xpn*, *ep*, and *pp* are induction variables and can be represented by closed-form expressions of the iteration counter *i*. In Figure 3, we present the loop after replacing *xpl*, *xpn*, *ep*, and *pp* by their closed-form expressions. To avoid confusion, we use *xpl'*, *xpn'*, *ep'*, and *pp'* to represent the values of *xpl*, *xpn*, *ep*, and *pp* before entering the loop.

One must keep in mind that, although represented in array syntax, *xpl'*, *xpn'*, *ep'*, and *pp'* are still pointers. This means that accesses through them can still be aliased. In fact, in this example, *xp1'* and *xpn'* are pointing to the first and the last element of an array of 1024 double elements, respectively. If the compiler knows that *xp1'* accesses up-to 512 elements onward and *xpn'* accesses up-to 512 elements backward during the loop, and that *xp1'* and *xpn'* are 1024 elements apart, the compiler can prove that the accesses through *xp1'* and *xpn'* are non-overlapping. The region access information can be obtained by analyzing subscripts and loop bounds [28]. The task of pointer analysis is then to find out the distance between the memory location pointed to by *xpl'*, *xpn'*, *ep'*, and pp'.

Once able to disambiguate the regions accessed by *xpl'*, *xpn'*, *ep'*, and *pp'*, we can apply traditional dependence analyses to resolve the dependences in the transformed loop in Figure 3.

```
for (y=ymin;y<=ymax;y++) {
  for (x=xmin;x<=xmax;x++) {
    ....
    GLfloat dist2 = dx*dx + dy*dy;
    if (dist2<rmax2) {
      ....
      (PB)->x[(PB)->count] = x;
      (PB)->y[(PB)->count] = y;
      (PB)->z[(PB)->count] = z;
      ....
      (PB)->count++;;
    }
  }
}
```

Figure 4. Non-closed-form Pointer Access Example from BMW/Mesa.

There may be loops that contain pointers with no closed-form expressions as shown in Figure 4. In this case, we can still exploit the monotonicity of the pointers to estimate the access region as well as conducting dependence analysis [29].

4.1.2 Manually Unrolled Loops

Because of the high performance demand of multimedia workloads, many multimedia programs are hand-optimized. One typical example is manually unrolled loops. For example, six of

the BMW benchmark programs contain unrolled inner loops. Figure 5 gives an example of a manually unrolled loop extracted from *mpeg2*, a video encoder application, where an inner loop has been completely unrolled 16 times to accumulate the absolute difference of two input arrays.

```
for (j=0; j<h; j++){
   if ((v=p1[0]-p2[0])<0) v= -v; s+= v;
       ....
   if ((v=p1[15]-p2[15])<0) v= -v; s+= v;

   if (s >= distlim)  break;
   p1+= lx;
   p2+= lx;
}
```

Figure 5. Manually Unrolled Loop from BMW/MPEG2.

In this example, it is difficult and expensive to vectorize statements across loop *j* because the accesses through *p1[0]* and *p2[0]* are non-continuous across iterations. The opportunity lies in vectorizing the 16 unrolled statements within the loop body.

```
for (j=0; j<h; j++){
   for(i=0; i<16; i++) {
     if((v=p1[i]-p2[i])<0) v = -v;
     s+=v;
   }
   if (s >= distlim) break;
   p1+= lx;
   p2+= lx;
}
```

Figure 6. Rerolled Loop from Figure 5

One solution is to first reroll the loop body, and then apply vectorization. Figure 6 shows the code after rerolling the loop in Figure 5. Fortunately, most of the unrolled loops we have seen in BMW benchmark are quite simple.

Another approach is to vectorize unrolled loops directly. The SLP algorithm mentioned in Section 3.2 offers such a solution by identifying isomorphic operations within a loop body and groups them to form MME SIMD instructions.

4.2 Limitations of the C Language

The mismatch between the C language and the underlying MME architecture also widens the gap between traditional and MME vectorization.

4.2.1 Integral Promotion and Subword Types

In ANSI C semantics, all *char* or *short* types (i.e., sub-word data types) are automatically promoted to *integer* type before conducting any arithmetic operations. This is known as *integral promotion* [3]. In essence, ANSI C supports the storage of sub-word data types but not operations on them. This design is a perfect match for general-purpose architectures because general-purpose ISA often only support integer operations on whole registers. In addition, integer extensions are often combined with load operations and incur no additional overhead.

Table 4. Major Data Type Used in BMW Applications

Application	Type	Application	Type
DVJU	*short*	ADPCM	*char/short*
JPEG	*short*	GSM	*short*
MPEG2	*char*	LAME	*double*
Doom	*char*	mpg123	*single*
Mesa	*char*	Rsynth	*single*
POVray	*double*	Timidity	*short*

On the other hand, since 8- or 16-bit integers are natural representation of many types of multimedia data, subword types are widely used in multimedia applications. As a result, it is very common to see a MME ISA that provides better support for subword operations than for 32-bit operations. From the vectorization point of view, when dealing with subword data types, following integral promotion rule means wasting more than half of the total computation bandwidth, and incurring additional overhead due to type extension. Vectorization of subword operations to word operations may even introduce slowdowns if the underlying ISA provides a native support for the former but not for the latter.

Therefore, the issue is to automatically avoiding unnecessary integral promotion without affecting program semantics. We need a backward data-flow analysis to trace the effective width of the result of any operation based on how the result is consumed. The effective width of the result operand is propagated to the source operands according to the operations. We can then safely convert a word operation to a subword operation if all the source operands of a word operation have a subword effective width.

4.2.2 Saturated Operations

```
/* short a, b; int ltmp; */
#define GSM_ADD(a, b) \
((unsigned)((ltmp=(int)(a)+(int)(b)) \
  - MIN_WORD) > MAX_WORD - MIN_WORD ?  \
 (ltmp>0 ? MAX_WORD : MIN_WORD): ltmp)
```

Figure 7. Saturated Add Implemented in C from BMW/GSM.

Saturated arithmetic is widely used in multimedia programs, especially in audio and image processing applications. Since C semantics does not support saturated arithmetic as native operators, programmers must express saturated operations in native C operations. Figure 7 gives such one example.

The code sequence in Figure 7 can be vectorized into a sequence of compare, mask, subtract and add with the help of *if-conversion*. However, for MME that directly supports saturated add, the best performance can only be achieved by recognizing the sequence and transforming it into a saturated add instruction. Idiom recognition, which has been used to identify max and min operations in scientific applications, can be extended to identify these saturated operations [30].

```
/* INT16 *bp, UINT8 *rfp */
for (i=0; i<8; i++) {
  for (j=0; j<8; j++) {
    *rfp = Clip[*bp++ + *rfp];
    rfp++;
  }
```

Figure 8. Another C Implementation of Saturated Add from BMW/MPEG2.

Interestingly, within BMW benchmark, there are other implementations of saturated add. In the example of Figure 8, array *Clip* is generated on-the-fly and it maps a subscript to its corresponding saturated 16-bit value. In this case, it becomes very difficult for a compiler to recognize this pattern.

4.3 Code Patterns

4.3.1 Bit-wise Operations

Due to the nature of multimedia processing, bit-wise operations are often used in multimedia applications. Figure 9 gives an example of bit-wise operations extracted from *mesa*, an OpenGL 3D graphics library. To vectorize this code, the key techniques are if-conversion and recognizing *tmpOrMask*, and *tmpAndMask* as reduction of bit-wise AND and OR operations.

```
for (i=0;i<n;i++) {
  ....
  UINT8 mask = 0;
  if(cx>cw)   mask|= CLIP_RIGHT_BIT;
  else if(cx<-cw) mask|= CLIP_LEFT_BIT;
  if(cy>cw)   mask|= CLIP_TOP_BIT;
  else if(cy<-cw) mask|= CLIP_BOTTOM_BIT;
  if(cz>cw)   mask |= CLIP_FAR_BIT;
  else if(cz<-cw) mask |= CLIP_NEAR_BIT;
  if (mask) {
    clipMask[i] |= mask;
    tmpOrMask |= mask;
  }
  tmpAndMask &= mask;
```

Figure 9. Reduction on bit-wise operations from BMW/mesa.

4.4 Limitations of the MME Architecture

In Section 2.1.2, we have thoroughly discussed the architectural difference between MME and traditional vector architectures. To summarize, multimedia extension uses short, fixed-length vectors, has a much weaker memory unit, and provides a less uniform and general-purpose ISA. We believe that these architectural differences lead to many differences between MME and traditional vectorization, which oftentimes make the former more difficult. Some of the new challenges are still open questions.

The short fixed-length SIMD architecture (typically with vector length less than 16-byte) implies that we can vectorize not only across iterations but also within iteration or even within a basic block. For the latter, a Super-word Level Parallelism (SLP) approach may be more effective [18].

The weak memory unit imposes a significant challenge to MME vectorization. The lack of native support for gather/scatter type of memory operations makes it very difficult to vectorize codes with non-continuous memory accesses. Figure 10 gives an simple example of stride accesses on *tmp [8*k+j]*.

```
for (j=0; j<8; j++)
  for (i=0; i<8; i++) {
    s = 0.0;
    for (k=0; k<8; k++)
      s += c[i][k] * tmp[8*k+j];
    block[8*i+j]=(int)floor(s+0.499999);
}
```

Figure 10. Strided Memory Access from BMW/mpeg2.

In addition, many multimedia extensions support only vector-aligned loads and stores. Precise alignment information not only benefits vectorization but also simplifies code generation. There are two aspects of alignment optimization for vectorization purposes: to obtain alignment information and to improve alignment by program transformation, such as loop unrolling [31]. Because of the pervasive use of pointers in multimedia applications, alignment analysis is in essence alignment analysis of pointers, and may require whole-program analysis. An alternative is to version different vectorization of the programs according to different alignment assumptions.

The non-uniform and domain-specific ISA complicates code generation. When we identify an expression that satisfies all the dependence, continuous access, and alignment requirements, we may still find that the expression does not have a direct mapping in the underlying ISA. Very likely this is because the operands of the expression are of less supported data types. For the vectorization to be successful, the vector code generator must be able to map a non-supported vectorizable expression into a sequence of native vector instructions. In essence, the code generator serves as a layer that hides the difference between the underlying non-uniform, domain-specific ISA and the uniform general-purpose "ISA" of the high-level programming languages.

4.5 Summary

All these features we discussed in this section are summarized in Table 5 for core procedures from Berkeley Multimedia Workload.

Table 5. Code Patterns in Berkeley Multimedia Workload Applications*

Name	Core Procedures	# of Loop	Pointer Access	Unrolled Loop	Sat Ops	Mixed Types	Bit-wise Reduction	Func Calls	If-Inst
adpcm	adpcm_coder	1x1	*		*	*			*
	adpcm_decoder	1x1	*		*	*			*
gsm	Calculation_of_the_LTP_parameters	1x2	*	*		*			
	Gsm_Short_Term_Analysis_Filter	1x2	*		*				
	Gsm_Short_Term_Synthesis_Filter	1x2	*		*				
lame	quantize	3x1	*			*		*	*
	calc_noise2	2x3	*			*		*	*
mpg123	synth_1to1	2x1	*	*	*	*			
jpeg	encode_mcu_AC_refine	1x2	*			*		*	*
	encode_mcu_AC_first	1x2	*			*		*	*
	jpeg_idct_islow	2x1	*	*	*	*			
	decode_mcu_AC_refine	1x2	*			*		*	*
	ycc_rgb_convert	1x2	*	*	*	*			
mpeg2	dist1	4x2	*	*		*			*
	fdct	2x3	*			*			
	form_component_prediction	4x2	*			*			*
	Dither_Frame	4x2	*	*		*			*
	idctcol	1x0	*	*	*	*			
	idctrow	1x0	*	*	*	*			
djvu	backward_filter	3x1	*			*			
	forward_filter	3x1	*			*			
	_IWCodec::decode_buckets	4x2	*			*	*	*	*
	_IWCodec::encode_buckets	4x2	*			*	*	*	*
	IWPixmap::init	1x2	*		*				
mesa	flat_TRUECOLOR_z_triangle	1x3	*					*	*
	smooth_TRUECOLOR_z_triangle	1x3	*					*	
	gl_shade_rgba_fast	1x2	*	*		*		*	*
	gl_depth_test_span_generic	12x1	*						*
	write_span_mono_ximage	1x1	*					*	*
	dist_atten_antialiased_rgba_points	2x3	*	*		*			*
	persp_textured_triangle	17x3	*	*		*			
rsynth	parwave	1x2						*	*
timidity	rs_vib_loop	1x2	*			*			
	rs_bidir	1x2	*			*			
	mix_mystery_signal	2x2	*			*			
	mix_single_signal	2x2	*			*			
povray	DNoise	1x0	*	*				*	
doom	R_DrawColumn	1x1	*						
	R_DrawSpan	1x1	*						
	R_RenderSegLoop	1x1	*			*		*	*

*In column "# of Loop", "*axb*" means there are *a* important loops in the procedure, which are *b*-level nest loops. "Sat Ops" stands for "Saturation Operations". "Func Calls" and "If-Inst" show whether the function calls and if-statements are used in the loop body, respectively.

5. CONCLUSION

Our study showed that despite the success of vectorizing for traditional vector processors, the vectorization for multimedia extensions still has a long way to go. The gap between MME vectorization and traditional vectorization is the natural result of both the architectural differences between multimedia extensions and traditional vector processors and the differences between multimedia applications and numerical applications.

In this paper, we conducted an in-depth study of the BMW benchmark suite. Based on the code study, we identified the key differences between MME and traditional vectorization, code patterns that are common in multimedia applications and new issues that arise in MME vectorization. We also discussed solutions to address some of them.

This work is only the first step towards unleashing the power of multimedia extensions through vectorization. Our study focuses more on identifying the new requirements and challenges faced by MME vectorization than on providing the actual solutions. Therefore our immediate future work is to propose new techniques to address the issues we identified and measure the effectiveness of these techniques on the BMW benchmark. At the same time, we would also like to extend our study on other application domain, such as numerical applications. It would be interesting to see how numerical programs can be vectorized and benefit from multimedia extensions.

6. REFERENCES

[1] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, Vol. 30, No 9, 43-45, September 1997.

[2] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. International Journal of Parallel Programming, Vol. 18, No 4, 347-361, 2000.

[3] Nathan T. Slingerland and Alan J. Smith. Design and characterization of the Berkeley multimedia workload. *Multimedia Systems*, Vol. 8, No 4, 315-327, 2002.

[4] Chunho Lee, Miodrag Potkonjak and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *Proceedings of Micro '97*, 330-335.

[5] *Using the GNU Compiler Collection (GCC).* Free Software Foundation, Boston, MA, 2002.

[6] David A. Patterson and John L. Hennessy. *Computer Architecture :A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1996.

[7] David J. Kuck at all. Measurements of Parallelism in ordinary FORTRAN programs. *IEEE Computer*, Vol. 7, 37-46, Jan. 1974.

[8] David Padua and Micheal Wolfe. Advanced Compiler Optimizations for Supercomputers. *ACM Communication*, Vol. 29, No 12, 1184-1201, Dec. 1986.

[9] Gerald Cheong and Monica S. Lam. An Optimizer for Multimedia Instruction Sets. *Second SUIF Compiler Workshop*, Stanford, August 1997.

[10] Venkat Konda, Hugh Lauer, Katsunobu Muroi, Kenichi Tanaka, Hirono Tsubota, Ellen Xu, Chris Wilson. A SIMDizing C Compiler for the Mitsubishi Electric Neuro4 Processor Array. *First SUIF Compiler Workshop*, Stanford, January 1996.

[11] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, Vol. 28, No 4, 347-361, 2000.

[12] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, Vol. 28, No 4, 363-400, 2000.

[13] Crescent Bay Software Corp. http://www.psrv.com/vast_altivec.html.

[14] The Portland Group Compiler Technology. http://www.pgroup.com/products/workpgi.htm.

[15] Codeplay Software Limited. http://www.codeplay.com/vectorc/features.html.

[16] Aart J. C. Bik, Milind Girkar, Paul M. Grey, Xinmin Tian. Automatic Intra-Register Vectorization for the Intel® Architecture. *International Journal of Parallel Programming*, Vol. 30, No 2, 65-98, April 2002.

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

[18] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *Proceeding of the SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, B.C., June 2000.

[19] D. Callahan, J. Dongarra and D. Levine. Vectorizing Compilers: A Test Suite and Results. *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, Orlando, Florida, 1988.

[20] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. *Proc.of Compcon '95*, San Francisco, California, March 1995.

[21] Ruby Lee, Larry McMahan. Mapping of Application Software to the Multimedia Instructions of General Purpose Microprocessors. *IS&T/SPIE Symp. on Electric Imaging: Science and Technology*, San Jose, California, February 1997.

[22] David A. Carlson, Ruben W.Castelino, Robert O.Mueller. Multimedia Extensions for a 550-MHz RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, Vol.32, No 11, 1618-1624, November 1997.

[23] MIPS Technologies, Inc. MIPS Extension for Digital Media with 3D. White Paper, March 1997.

[24] Intel Corporation. IA32 Intel Architecture Software Developer's Manual with Preliminary Intel Pentium 4 Processor Information Volume 1: Basic Architecture.

[25] Stuart Oberman, Greg Favor, Fred Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, Vol. 19, No 2, 37-48, March 1999.

[26] Intel Corporation. Intel Architecture Optimization Reference Manual.

[27] Sam Fuller. Motorola's AltiVec Technology. White Paper, May 6, 1998.

[28] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94*, Washington D.C., 528-537, November 1994.

[29] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic Evolution: An Alternative to Induction Variable Substitution for Dependence Analysis. *Proceedings of the 15th International Conference on Supercomputing*, Sorrento, Italy, June 2001.

[30] A.J.C. Bik, M. Girkar, P.M. Grey, and X. Tian. Automatic Detection of Saturation and Clipping Idioms. *In Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computers*, July, 2002.

[31] Sam Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. *In Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Charlottesville, VA, September, 2002.