Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment

Joe H. Wolf III, Microprocessor Products Group, Intel Corporation

Index words: VTune™, Intel® C/C++ Compiler, intrinsics, vector class library, vectorization, event-based sampling, Intel® Performance Library Suite.

ABSTRACT

This paper describes the programming methods available to software developers wishing to utilize the performance capabilities of the Streaming SIMD Extensions of the Pentium® III processor. The tools in the VTune™ Performance Enhancement Environment, Version 4.0, have unique capabilities that help software developers understand the Streaming SIMD Extensions, develop applications for them, and performance tune those applications.

The tools are the Intel® C/C++ Compiler, the VTune Performance Analyzer, the Intel® Architecture Performance Training Center, the Intel® Performance Library Suite, and the Register Viewing Tool. The programming methods offered by these tools are as follows:

(1) *Intrinsics*. These are function-like calls the user inserts in an application for which the Intel C/C++ compiler generates inlined code.

(2) *Vector Class Library*. This is a C++ abstraction of the intrinsics.

(3) *Vectorization*. This is a special case of compiler optimization that finds loops operating upon arrays of char, short, int, or float, and creates a more efficient loop using the SIMD instructions.

(4) *The Intel Performance Library Suite*. These libraries have highly tuned routines to take advantage of the Streaming SIMD Extensions for a number of commonly used algorithms. The libraries include the Intel® Signal Processing Library, the Intel® Image Processing Library, the Intel® Recognition Primitives Library, the Math Kernel Library, and the Intel® JPEG Library.

In addition, the VTune Performance Analyzer offers a number of ways of looking at the performance of an application, and gives feedback on ways to tune for the Pentium III processor. Examples of several of these features are given.

INTRODUCTION

Intel® MMX™ technology was introduced into the Intel® Architecture in 1996. It provided, and still provides, unique performance opportunities through a Single-Instruction, Multiple-Data (SIMD) instruction set architecture (ISA) for integer-based code. However, when it was introduced, and for almost two years afterwards, the only way for developers to access and utilize the SIMD technology was through assembly, either assembly files or inlined assembly in C or C++ code. While assembly programming arguably may offer the best performance compared to compiled high-level languages, it is difficult and inefficient to write, performance tune, maintain, and port to new ISA's. Clearly, developers wanted then, and demand now, high-level language support for the SIMD ISA's like that of MMX technology and the Streaming SIMD Extensions of the Pentium III processor.

This demand for high-level language support was the motivation behind developing the VTune™ Performance Enhancement Environment, Version 4.0. Its unique development methods allow programmers to obtain all of the performance available in the SIMD ISA through high-level language support in the Intel® C/C++ Compiler, VTune Analyzer, and Performance Library Suite.

An example of a simple loop is given in the first section of this paper on the Intel C/C++ Compiler. This example is expanded by showing different methods of support for the Streaming SIMD Extensions, as well as by giving guidelines for optimal use of the methods. The same example is also used in the VTune Analyzer section to illustrate how to find performance-critical sections of an application suitable for recoding using the Streaming SIMD Extensions. Also shown are methods of using the VTune Analyzer to obtain advice for recoding or tuning the application, and methods of getting information on cache utilization vital to analysis for insertion of prefetch or streaming stores.

There is little or no performance difference between the methods, but each offers significant performance improvements over the scalar floating-point implementation. This performance improvement comes at a fraction of the development costs associated with writing assembly code. The conclusion is that the user has several different programming options using the SIMD ISA, the only differences being coding style and efficiency of implementation.

THE INTEL® C/C++ COMPILER

The Intel® C/C++ Compiler is a highly-optimizing compiler that plugs into the Microsoft[*] Developer's Studio environment. It is a C++ standard conforming compiler that is also language, debug, and object format compatible with Microsoft's Visual C++, Versions 4.2 and higher.

The compiler offers several options for programmers to utilize the Streaming SIMD Extensions: inlined assembly, intrinsics, vector class libraries, and vectorization. Since the Streaming SIMD Extensions require 16-byte alignment of data for maximal performance, the compiler offers several different methods to ensure that data are properly aligned. All of these methods are discussed in detail in this section.

Data and Stack Alignment

Data must be 16-byte aligned to obtain the best performance with the Streaming SIMD Extensions of the Pentium® III processor. In addition, exceptions can occur if the aligned data movement instructions are used, but data are not properly aligned. To eliminate these problems, the compiler provides the following mechanisms:

- A new data type, *__m128*, that can be thought of as a *struct* of four single-precision floats or an XMM register. Data that are declared with this type are automatically aligned to a 16-byte boundary, whether they be global or local data.
- Another new data object for use in C++ code is the *F32vec4* class. This is a class object whose data member is a *__m128* data item. The compiler treats these objects similarly to the *__m128* type.
- *__declspec(align(16))* is a new specifier for data declarations that tells the compiler to align the given data items. This is particularly useful for global data items that may be passed into routines where the Streaming SIMD Extensions are used. For example:

  __declspec(align(16)) float buffer[400];

  The variable, *buffer*, could then be used as if it contained 100 objects of type *__m128* or *F32vec4*. In the following example, the construction of the *F32vec4* object, *x*, will then occur with aligned data. Without the *__declspec(align(16))*, however, a fault may occur. An example of such usage is

  ```
  void foo() {
          F32vec4 x = *(__m128 *) buffer;
          ...
  }
  ```
- In some cases, for better performance, the compiler will align routines with *__m64* (the MMX™ technology, or integer SIMD data type) or *double* data to 16-bytes by default. The compiler also has a command-line switch, *-Qsfalign16*, which can be used to limit the compiler to only do the alignment in routines that contain Streaming SIMD Extensions' data. The default behavior is to use *-Qsfalign8*, which says to align routines with 8- or 16-byte data types to 16-bytes.

The compiler automatically aligns the stack frame for both debug and non-debug code for functions in which these extensions are used. The actual layout of the stack frames are shown in detail in [1]. References [2] and [5] give more details and examples of how to efficiently use these extensions.

INTRINSICS

Intrinsics are C-like function calls for which the compiler generates optimal inlined code. Each intrinsic maps to a specific Streaming SIMD Extensions instruction, or an MMX™ technology instruction. Most take *__m128* or *__m64* (integer) data types as their arguments. Even though there is a one-to-one mapping between an intrinsic

[*]All other brand names are the property of their respective owners.

and its corresponding assembly instruction, the intrinsics are much more efficient to write than assembly code because the compiler takes care of register allocation and instruction scheduling for the programmer. There are also a number of data initialization intrinsics to easily allow the loading of a *__m128* data type (or an XMM register).

The example shown in Figure 1 shows a simple loop written in C++. This loop is used as an example throughout this article.

```
float  xa[ARRAY_SIZE],  xb[ARRAY_SIZE],
       xc[ARRAY_SIZE];
float q;

void do_c_triad() {

 for (int j = 0; j < ARRAY_SIZE; j++) {
   xa[j] = xb[j] + q *  xc[j];
 }
}
```

Figure 1: Original C++ triad loop

This figure shows a single-precision floating-point triad operation. It performs a scaling of a vector *(q *xc[j])*, adding it to another vector, and storing the result. Note that there is no re-use of the data in the loop.

Figure 2 gives some examples of the syntax of some intrinsics that may be used for coding the example in Figure 1.

```
__m128 _mm_set_ps1(float f)

__m128 _mm_load_ps(float *mem)

__m128 _mm_mul_ps(__m128 x, __m128 y)

__m128 _mm_add_ps (__m128 x, __m128 y)

void _mm_store_ps(float *mem, __m128 x)
```

Figure 2: Intrinsic syntax

_mm_set_ps1() is used to replicate or broadcast a scalar float variable or constant across a *__m128* variable.

_mm_load_ps() is used to load a *__m128* variable from a memory location, such as a float array.

_mm_mul_ps() and *_mm_add_ps()* each take two *__m128* operands and perform a multiply or addition, respectively, returning the result in a *__m128* data type.

_mm_store_ps() takes a *__m128* variable and stores it to the given memory location.

A complete listing of the intrinsics can be found in references [2] and [6] along with a complete listing of the Pentium® III processor instructions.

```
#define VECTOR_SIZE 4
__declspec(align(16)) float  xa[ARRAY_SIZE],
    xb[ARRAY_SIZE],  xc[ARRAY_SIZE];
float q;

void do_intrin_triad() {
  __m128 tmp0, tmp1;

  tmp1 = _mm_set_ps1(q);
  for (int j = 0; j < ARRAY_SIZE;   j+=VECTOR_SIZE){
    tmp0 = _mm_mul_ps(*((__m128 *) &  xc[j]), tmp1);
   *(__ m128 *) & xa[j] =
           _mm_add_ps(tmp0, *((__m128 *) &  xb[j]));
  }
}
```

Figure 3: Intrinsics encoding of the triad loop

Recoding the example in Figure 1 using the intrinsics entails several considerations:

1. Since the example loop is operating on global data, be sure the data is 16-byte aligned. This requires the use of *__declspec(align(16))* for the float array declaration in the global program scope.
2. In any SIMD encoding of a loop, strip-mining or adjusting the loop iteration count by the vector size (the number of elements able to be operated upon per SIMD operation) is necessary. Therefore, the iteration count in this example is reduced by four, the size of a Streaming SIMD Extension's XMM register or data type. This is done via the *j+=VECTOR_SIZE* loop index variable increment.
3. We used the *_mm_set_ps1()* intrinsic to broadcast the scalar *q* across the *tmp1 _mm128* variable. Also note that this is used outside of the loop since it is invariant to the loop.
4. Rather than explicitly loading from the arrays *xb* and *xc* into *__m128* types using the *_mm_load_ps()* intrinsics, we coerced them into *__m128* types for use as operands to the *_mm_mul_ps()* and

_mm_add_ps() intrinsics. This gives the compiler complete control over the register allocation, and it allows it to determine when it is really necessary to do the loads. Similarly, the result of the add intrinsic was cast directly to the result array, *xa*, rather than using the *_mm_store_ps()* intrinsic. The compiler generates the appropriate store instruction for the programmer.

One can see that the compiler does a lot of the work for the programmer when the intrinsics are used, enabling the programmer to be much more efficient at encoding an SIMD algorithm.

SIMD INTRINSICS USAGE GUIDELINES

The following are guidelines for getting optimal performance from the intrinsics. All of these are excerpted from the Intel® C/C++ Compiler, Version 4.0, release notes.

1. Do not use static or extern variables when a local variable could be used. Static and extern variables are not usually kept in registers. In addition, C language alias rules usually cause assignments through pointers to alias static and extern variables, thus restricting instruction scheduling.

Not So Good:

```
void foo (m128 *dst,m128 *src, m128 junk) {
  static m128 t;
  int i;

  for (i = 0; i < 1000; i++, dst++, src++)

  {
    t = _mm_mul_ps(*src, junk);
    *dst = _mm_add_ps(*dst, t);
  }
}
```

Better:

```
void foo (__m128 *dst, __m128 *src, m128
junk) {
  m128 t;
  int i;

  for (i = 0; i < 1000; i++, dst++, src++)

  {
     t = _mm_mul_ps(*src, junk);
     *dst = _mm_add_ps(*dst, t);
  }

}
```

2. Do not reference the address of variables or parameters. Using the address of a variable or parameter, via the address operator, &, makes the variable no longer a candidate for being kept in a register. It therefore must be kept in memory, possibly causing poor performance. Also, like static and extern variables, any assignments through pointers will now alias the variable, constraining instruction scheduling. This is particularly bad for parameters, because referencing the address of any parameter aliases all other parameters in the Intel C/C++ Compiler, Version 4.0, implementation.

Not so good:

```
void f(float *dst, float dscale, int n) {
  m128 t1; int i;
  t1 = _mm_load_ps1(&dscale);

  for (i = 0; i < n; i++) {
    *(__m128 *)dst =
     _mm_mul_ps(*(__m128 *)dst, t1);
    dst += 4;
  }
}
```

Better:

```
void f(float *dst, float dscale, int n) {
  m128 t1; int i;
  t1 = _mm_set_ps1(dscale);

  for (i = 0; i < n; i++) {
    *(__m128 *)dst =
     _mm_mul_ps(*(__m128 *)dst, t1);
    dst += 4;
  }
}
```

3. Where possible, make loop bounds compile-time constants. When this is not possible, make the expressions for the loop bounds refer only to local variables whose addresses are never taken. This helps ensure that the loop termination condition doesn't cause unnecessary work inside the loop.

Not So Good:

```
 int i, n;
 get_bounds(&n);
 /* In this example, we'll have to reload n

    and do the divide every loop iteration,

    causing poor performance. */
  for (i = 0; i < n / 4; i++) { ... }
```

Better:

```
int i,n, l_end;
get_bounds(&n);
l_end = n / 4;
for (i = 0; i < l_end; i++) { ... }
```

4. Code the loops using intrinsics so the last thing the loop does is write values back into memory. Use local variables for intermediate calculations. This allows the scheduler maximum freedom to rearrange the code, and it keeps the number of memory references to a minimum. It is a general problem in the C/C++ language that references through pointers alias other references through pointers.

Not so good:

```
m128 *dst, *src, c; int i, n;
for (i = 0; i < n; i += 2) {
   dst[i]   = _mm_add_ps(src[i], c);
   dst[i+1] = _mm_add_ps(src[i+1], c);
}
```

Better:

```
m128 *dst, *src, c, t1, t2; int i, n;
for (i = 0; i < n; i+= 2) {
   t1 = _mm_add_ps(src[i], c);
   t2 = _mm_add_ps(src[i+1], c);
   dst[i] = t1;
   dst[i+1] = t2;
}
```

5. Do not use the following intrinsics in loops:

```
_mm_set_ps()
_mm_setr_ps()
_mm_set_ps1()
_mm_set_ss()
```

   These intrinsics are used for data initialization of *__m128* data types and do not correspond directly to machine instructions. There are typically several machine instructions needed to implement each of these and therefore they may have a high run-time cost. The best way to use these intrinsics is to set a local __m128 variable to be the result produced by the intrinsic prior to entering a loop, and then use the local variable within the loop. The example in Figure 3 illustrates this.

6. For short loops, where loop unrolling is desired for improved performance, unroll the loop in the source code. Loop unrolling is a technique for replicating the operations in a loop and reducing the number of iterations correspondingly. For further information and examples on loop unrolling, refer to reference [8].

Not So Good:

```
m128 *a, *b, *c; int i;
for (i=0; i < 16; i++) {
    a[i] = _mm_add_ps(b[i], c[i]);
}
```

Better:

```
m128 *a, *b, *c, t1, t2; int i;
for (i=0; i < 16; i+=2) {
/* This loop has been unrolled twice */
   t1 = _mm_add_ps(b[i], c[i]);
   t2 = _mm_add_ps(b[i+1], c[i+1]);
   a[i] = t1;
   a[i+1] = t2;
}
```

Vector Classes

The vector classes provide an easy, efficient way of using the intrinsics in C++ code. The class, *F32vec4*, is defined for the floating-point Streaming SIMD Extensions. The *I32vec2*, *I16vec4* and *I8vec8* classes are defined for the three different types of data used in MMX^{TM} technology (*char*, *short*, and *int*). Each of these are abstractions of the *__m64* and *__m128* data types and the intrinsics supported for them. The implementation for these classes is provided with the Intel C/C++ Compiler in the *ivec.h* (integer SIMD) and *fvec.h* (float SIMD) header files. The member functions are overloads of the basic operators, like *, +, -, /, square root, and comparisons. Users may redefine and extend the classes to their own liking.

The example in Figure 4 shows the encoding of the triad function using the *F32vec4* class.

```
#define VECTOR_SIZE 4

__declspec(align(16)) float xa[ARRAY_SIZE],

    xb[ARRAY_SIZE],  xc[ARRAY_SIZE];

float q;


void do_fvec_triad() {

  F32vec4 q_xmm = (q, q, q, q);

  F32vec4 *xa_xmm = (F32vec4 *) & xa;

  F32vec4 *xb_xmm = (F32vec4 *) & xb;
  F32vec4 *xc_xmm = (F32vec4 *) & xc;

  for (int j = 0;

       j < (ARRAY_SIZE/VECTOR_SIZE); j++) {
    xa_xmm[j] = xb_xmm[j] +

                 q_xmm * xc_xmm[j];

  }

}
```

Figure 4: Vector class encoding of the triad loop

Note the following when using the vector classes:

1. There are several constructors defined to allow constants, variables, or pointered data (for example, arrays) to be converted to an SIMD class object. In the example in this figure, we used the broadcast or replication constructor to load the scalar *q_xmm*. To keep constructor usage and memory references to a minimum, we coerced the input global arrays to pointers to *F32vec4* objects for use in the loop.
2. Since we cast the arrays of floats to be pointers to *F32vec4* objects (*xa_xmm, xb_xmm, xc_xmm*), the memory references in the loop are to arrays of *F32vec4* objects, where each object is a *__m128* type. Therefore, we iterate over individual *F32vec4* objects so we have to keep the loop index increment set to 1. We then changed the loop exit condition to be the original array size divided by the vector size to reflect the compressed operations.

The vector classes provide a very clean implementation of the SIMD code. Since the classes contain overloaded operators for the most common operations, a programmer can redefine float data types to be the *F32vec4* classes and get the benefit of the Streaming SIMD Extensions everywhere the class is used, with minimal program changes.

Vectorization

The final method of support for SIMD coding in the Intel C/C++ Compiler is through vectorization. This is where the compiler attempts to generate the appropriate SIMD code for a given array operation within a loop with some hints from the programmer. The hints are in the form of *#pragma*'s in C or C++, and/or command-line switches that guide the compiler. There are a number of each, so the reader is advised to consult the Intel C/C++ Compiler User's Guide, [2], for more details. The most commonly used hints are given in Figures 5, 6, and 7.

```
#define VECTOR_SIZE 4
__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];
float q;

void do_vector_triad() {

#pragma vector aligned
  for (int j = 0; j < ARRAY_SIZE; j++) {
      xa[j] = xb[j] + q * xc[j];
  }
}
```

Figure 5: Compiler vectorization of the triad loop

The example in Figure 5 is easily vectorized by the compiler. The only hint needed is the *#pragma vector aligned*. This tells the compiler that the data are properly aligned so that the aligned data move instructions can be used. Without this, or its corresponding command-line option, the compiler would have to generate the unaligned move instructions, causing a significant loss of performance compared to the aligned instructions.

In Figure 6, we show an example of a slightly different version of the triad loop where the data are passed into the routine as parameters.

```
#define
__declspec(align(16))   xa[ARRAY_SIZE
    xb[ARRAY_SIZE]xc[ARRAY_SIZE
float

void do_vector_triad(float
                     float
                         float *c)
#pragma vector
  for (int j = 0; j < ARRAY_SIZE;
     a[j] = b[j] + q *
  }
}
```

Figure 6: Pointer version of the triad loop

Passing the arrays into the loop as shown in Figure 6 greatly impacts the vectorizability of the routine. This is because the compiler is now looking at pointered data instead of simple array references. As a result, the compiler must now assume that there are conflicts in the memory references in the loop where data written on one iteration may be used on the next, preventing a straightforward SIMD encoding of the loop. This is due to pointer aliasing. For example, to ensure program correctness, the compiler may assume that *xa* may be pointing to *xb[1]*, causing the value stored into *xa[j]* on every iteration to be reused as *xb[j]* on each subsequent iteration.

In order for the compiler to not have to make such assumptions, a new keyword, *restrict*, has been implemented. It tells the compiler that the data to which a pointer points is only accessible via that pointer variable in the current scope. Figure 7 shows its use.

```
#define VECTOR_SIZE 4
__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];
float q;

void do_vector_triad(float *restrict a,
                     float *restrict b,
                     float *restrict c) {
#pragma vector aligned
  for (int j = 0; j < ARRAY_SIZE; j++) {
     a[j] = b[j] + q * c[j];
  }
}
```

Figure 7: *restrict* keyword usage

Now the compiler is allowed to assume that each pointer reference is to different arrays or memory locations.

Vectorization Restrictions

The following are restrictions on the use of vectorization:

1. Loops must be countable; in other words, the iteration count must not change within the loop:

Good: `for (i=0; i<N; i++) …`

`while (i<100) { … i = i + 2; … }`

Bad: `while (p) { … p=p->next …}`

2. The body of a loop must consist of a single basic-block. In other words, there can be no if-statements and no internal branching. The loop must also have a single entry and exit.

3. The supported datatypes are *float*, *char*, *short*, and *int*. Do not mix these data types within the loop.

4. Alignment for Streaming SIMD Extensions data is up to the user. Use the *#pragma vector aligned* on a loop-by-loop basis, or the *–Qvec_alignment2* command-line option to state that all vectorizable data are properly aligned. If vectorized data are not aligned (and cannot be aligned), the compiler will use *movups*, the unaligned memory reference instruction.

5. The data accesses in the loop must be single-unit strides, or accessed contiguously in increments of one.

6. Assignments to scalar data are not allowed. The scalar memory references must be on the right-hand side of the equal sign.

7. There can be no function calls in the loop. This includes the intrinsic/transcendental calls like *sqrt()* or *cos()*.

The compiler generates messages stating if a loop was vectorized and gives a reason if it was not. This is done through the *–Qvec_verbose{0,1,2,3}* compiler switch, where the number indicates the level of detail in the messages. Although the restrictions above limit the types of loops that may be vectorized, a user can use the vectorization messages to coerce the compiler into vectorizing a loop. It may take a few iterations of compile, look at the messages, restructure the loop, add pragmas, and re-compile. However, this can be significantly less time consuming than recoding the loop in intrinsics.

Performance Considerations

The difference in performance between the methods is negligible. The vector class implementation can sometimes be slightly degraded compared to the intrinsics because of C++ overhead. However, this should be rare. The performance of each these methods is typically within 10%-15% of the performance of optimized hand-coded assembly. This difference, however, is more than offset by the ease of coding, maintainability, and portability using C or C++.

VTUNE™ PERFORMANCE ANALYZER

The VTune™ Performance Analyzer (or VTune Analyzer) is a tool that gives a user a graphical view of the performance of an application via a number of different methods, and it gives feedback on tuning the applications. The following is a brief description of each of these methods.

Event-Based Sampling

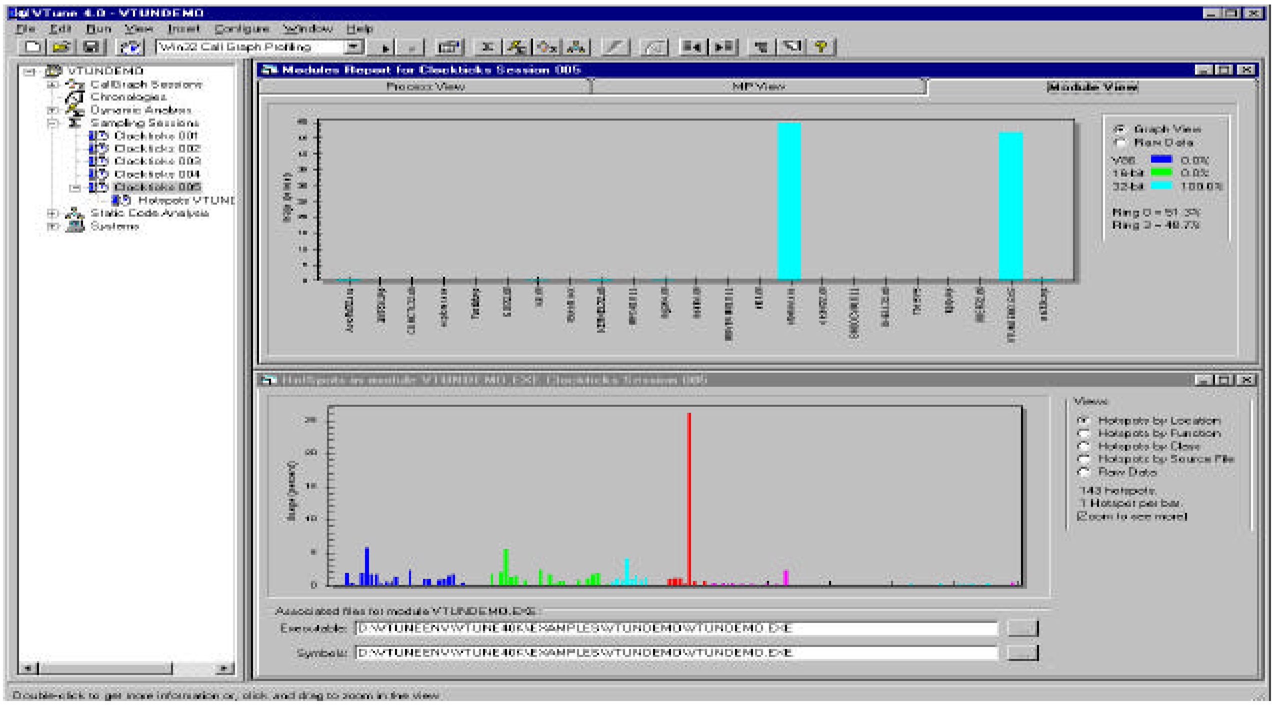
Event-based sampling is the most commonly used method for analyzing application performance with the VTune Analyzer. It allows the user to select any of a number of different events implemented in the processor. These events allow the user to hone in on specific aspects of an application's use of the processor from clocktick events or time, to specific types of operations that retired, to penalties that occur.

The processor is sampled after a specified number of the chosen events have occurred and the program counter address is noted. The analyzer then reports where in the user's program, or any other program running on the system, the events occurred.

The analyzer displays this information in a Modules report. This is a bar graph showing the occurrences of the events for all applications and modules making use of the processor, and in which events were collected. The Hotspots report is similar; it shows the same information for a single module.

For Streaming SIMD Extensions performance tuning, use event-based sampling with the *Clockticks* and *Floating-point operations retired* events and the event ratios, an indication of where the most time is spent performing floating-point operations is given. In this way likely candidates for Streaming SIMD Extensions coding can be found.

Double-clicking on a hotspot bar takes one to the source code corresponding to the occurrence of the events in the graph. Figure 8 shows the Modules and Hotspots report for a simple application using the clockticks event. Figure 9 shows the source code view for an application that shows both the *Clockticks* and *Floating-point operations retired* event occurrences.



V86	🟦	0.0%
16-bit	🟩	0.0%
32-bit	🟦	100.0%

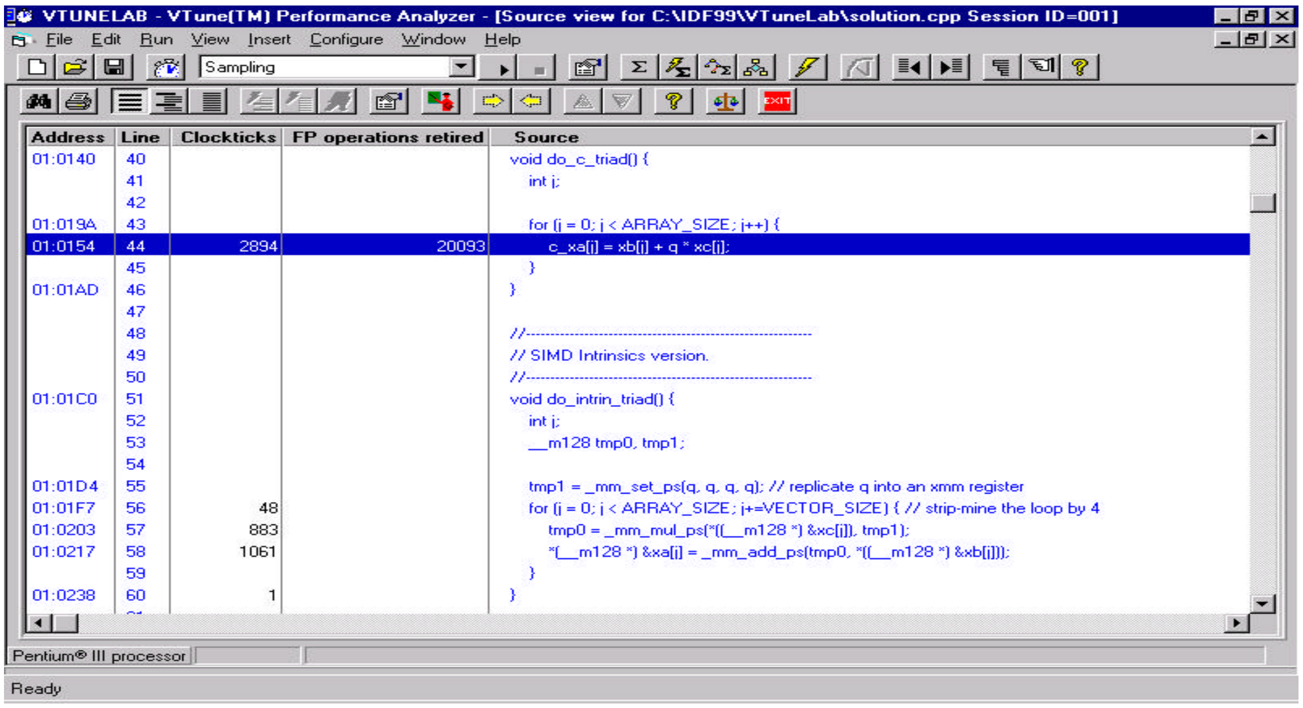
Ring 0 = 51.3%
Ring 3 = 48.7%

143 hotspots.
1 Hotspot per bar.
[Zoom to see more]

Associated files for module VTUNDEMO.EXE:
Executable: D:\VTUNEENV\VTUNE40K\EXAMPLES\VTUNDEMO\VTUNDEMO.EXE
Symbols: D:\VTUNEENV\VTUNE40K\EXAMPLES\VTUNDEMO\VTUNDEMO.EXE

Double-click to get more information or, click and drag to zoom in the view

Figure 8: Modules and Hotspots report


VTUNELAB - VTune(TM) Performance Analyzer - [Source view for C:\IDF99\VTuneLab\solution.cpp Session ID=001]
File Edit Run View Insert Configure Window Help
Sampling
EXIT
Pentium® III processor
Ready

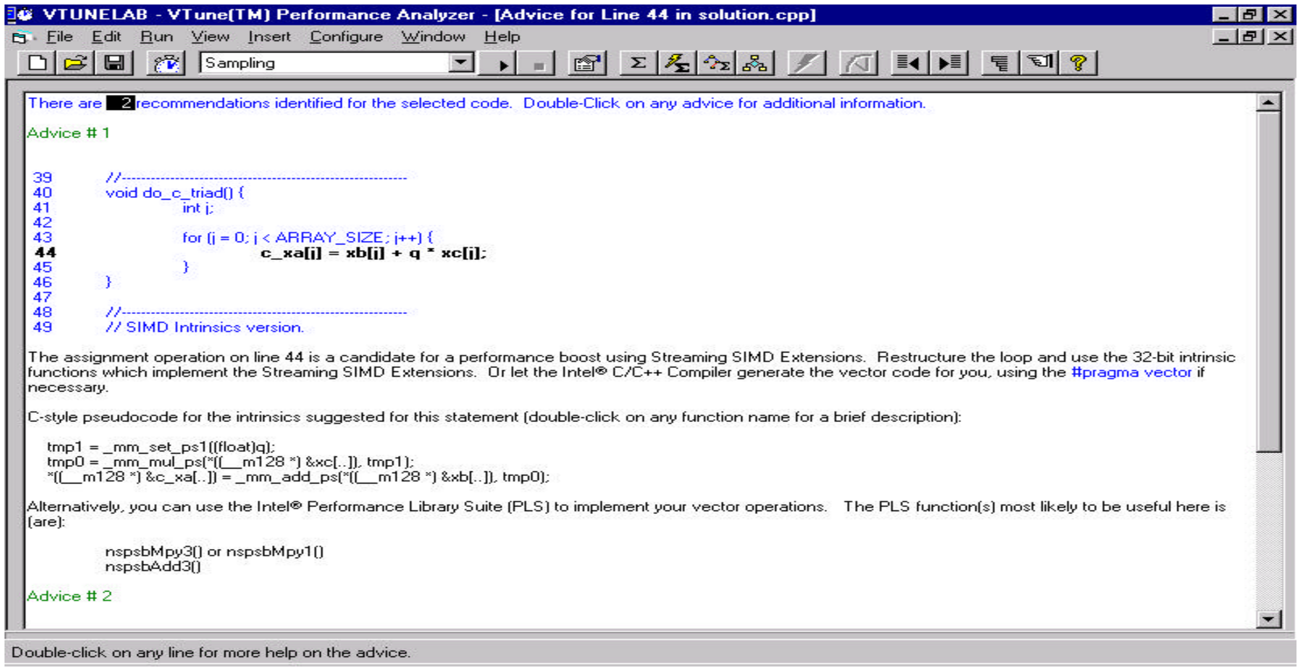
Address	Line	Clockticks	FP operations retired	Source
01:0140	40			void do_c_triad() {
	41			int j;
	42			
01:019A	43			for (j = 0; j < ARRAY_SIZE; j++) {
01:0154	44	2894	20093	c_xa[j] = xb[j] + q * xc[j];
	45			}
01:01AD	46			}
	47			
	48			//--
	49			// SIMD Intrinsics version.
	50			//--
01:01C0	51			void do_intrin_triad() {
	52			int j;
	53			__m128 tmp0, tmp1;
	54			
01:01D4	55			tmp1 = _mm_set_ps(q, q, q, q); // replicate q into an xmm register
01:01F7	56	48		for (j = 0; j < ARRAY_SIZE; j+=VECTOR_SIZE) { // strip-mine the loop by 4
01:0203	57	883		tmp0 = _mm_mul_ps(*((__m128 *) &xc[j]), tmp1);
01:0217	58	1061		*(__m128 *) &xa[j] = _mm_add_ps(tmp0, *((__m128 *) &xb[j]));
	59			}
01:0238	60	1		}

Figure 9: Source code view

Code Coach

The Analyzer's Code Coach analyzes the source code using some information from the compiler and offers advice on ways to tune the application. The advice ranges from tips on restructuring search algorithms, to unnecessary casts or conversions of data types, to advice on using the intrinsics or Performance Library Suite to take advantage of the Streaming SIMD Extensions or MMX^{TM} technology.

The advice is obtained by double-clicking on a statement in a source code view. Figure 10 shows the Coach advice for the *do_c_triad()* function used in Figure 1.


VTUNELAB - VTune(TM) Performance Analyzer - [Advice for Line 44 in solution.cpp]
File Edit Run View Insert Configure Window Help
Sampling

There are 2 recommendations identified for the selected code. Double-Click on any advice for additional information.

Advice # 1

```
39    //------------------------------------------------------------
40    void do_c_triad() {
41            int j;
42
43            for (j = 0; j < ARRAY_SIZE; j++) {
44                    c_xa[j] = xb[j] + q * xc[j];
45            }
46    }
47
48    //------------------------------------------------------------
49    // SIMD Intrinsics version.
```

The assignment operation on line 44 is a candidate for a performance boost using Streaming SIMD Extensions. Restructure the loop and use the 32-bit intrinsic functions which implement the Streaming SIMD Extensions. Or let the Intel® C/C++ Compiler generate the vector code for you, using the #pragma vector if necessary.

C-style pseudocode for the intrinsics suggested for this statement (double-click on any function name for a brief description):

```
tmp1 = _mm_set_ps1((float)q);
tmp0 = _mm_mul_ps(*((__m128 *) &xc[..]), tmp1);
*((__m128 *) &c_xa[..]) = _mm_add_ps(*((__m128 *) &xb[..]), tmp0);
```

Alternatively, you can use the Intel® Performance Library Suite (PLS) to implement your vector operations. The PLS function(s) most likely to be useful here is (are):

```
nspsbMpy3() or nspsbMpy1()
nspsbAdd3()
```

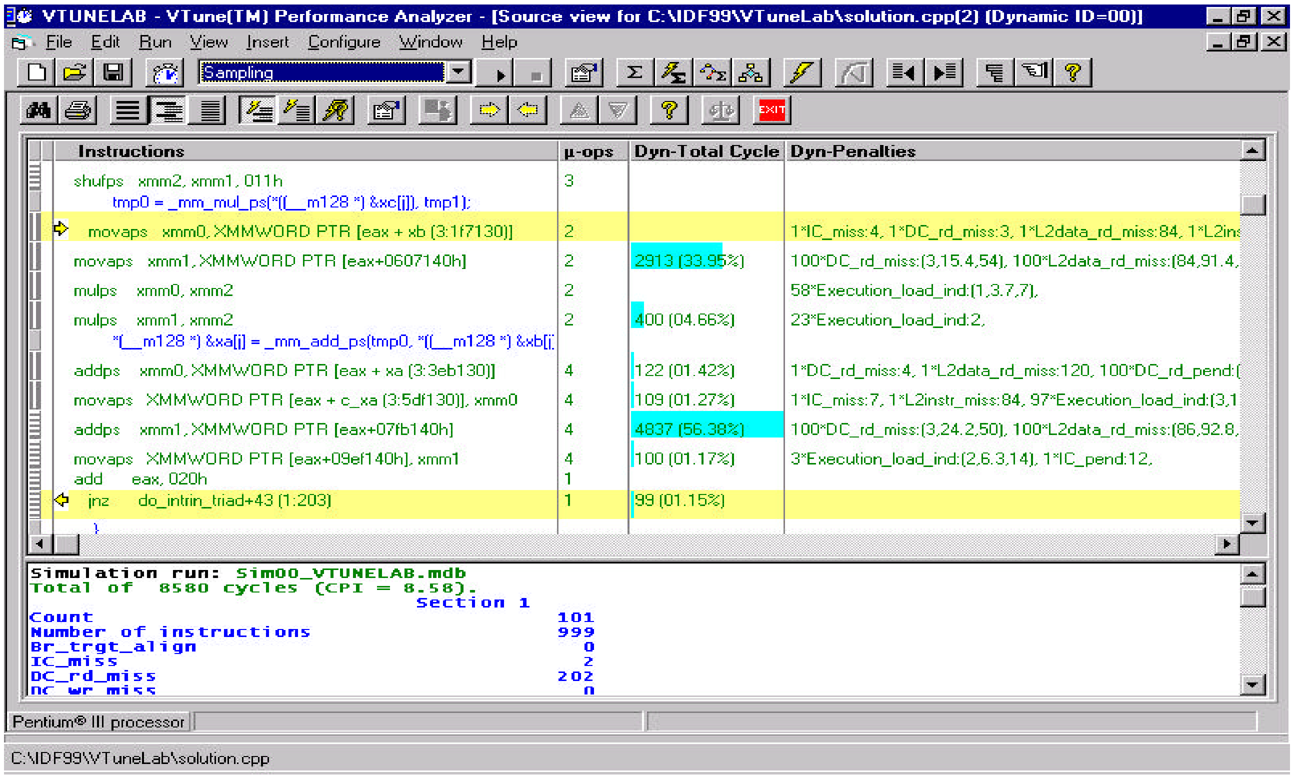
Advice # 2

Double-click on any line for more help on the advice.

Figure 10: Streaming SIMD Extensions Code Coach advice

Dynamic Analysis

Dynamic analysis uses the same software simulator the Pentium® II and Pentium® III processor architects used in designing the processors. It is useful for honing in on specific micro-architectural information for hotspots identified by event-based sampling such as penalties and retirement time. It is especially useful for analyzing branch mispredictions and cache utilization. The basic method of using dynamic analysis is to select a region of code to be simulated. Figure 11 shows the dynamic analysis results for the *do_intrin_triad()* function in Figure 3.



VTUNELAB - VTune(TM) Performance Analyzer - [Source view for C:\IDF99\VTuneLab\solution.cpp(2) (Dynamic ID=00)]
File Edit Run View Insert Configure Window Help
Sampling
EXIT
Pentium® III processor
C:\IDF99\VTuneLab\solution.cpp

Instructions	µ-ops	Dyn-Total Cycle	Dyn-Penalties
shufps xmm2, xmm1, 011h	3		
tmp0 = _mm_mul_ps(*((__m128 *) &xc[j]), tmp1);			
movaps xmm0, XMMWORD PTR [eax + xb (3:1f7130)]	2		1*IC_miss:4, 1*DC_rd_miss:3, 1*L2data_rd_miss:84, 1*L2ins
movaps xmm1, XMMWORD PTR [eax+0607140h]	2	2913 (33.95%)	100*DC_rd_miss:(3,15.4,54), 100*L2data_rd_miss:(84,91.4,
mulps xmm0, xmm2	2		58*Execution_load_ind:(1,3.7,7),
mulps xmm1, xmm2	2	400 (04.66%)	23*Execution_load_ind:2,
*(__m128 *) &xa[j] = _mm_add_ps(tmp0, *((__m128 *) &xb[j			
addps xmm0, XMMWORD PTR [eax + xa (3:3eb130)]	4	122 (01.42%)	1*DC_rd_miss:4, 1*L2data_rd_miss:120, 100*DC_rd_pend:(
movaps XMMWORD PTR [eax + c_xa (3:5df130)], xmm0	4	109 (01.27%)	1*IC_miss:7, 1*L2instr_miss:84, 97*Execution_load_ind:(3,1
addps xmm1, XMMWORD PTR [eax+07fb140h]	4	4837 (56.38%)	100*DC_rd_miss:(3,24.2,50), 100*L2data_rd_miss:(86,92.8,
movaps XMMWORD PTR [eax+09ef140h], xmm1	4	100 (01.17%)	3*Execution_load_ind:(2,6.3,14), 1*IC_pend:12,
add eax, 020h	1		
jnz do_intrin_triad+43 (1:203)	1	99 (01.15%)	
}			

```
Simulation run: Sim00_VTUNELAB.mdb
Total of  8580 cycles (CPI = 8.58).
                              Section 1
Count                                    101
Number of instructions                   999
Br_trgt_align                              0
IC_miss                                    2
DC_rd_miss                               202
DC_wr_miss                                 0
```

Figure 11: Dynamic analysis results

The dynamic analysis results shown in Figure 11 indicates that there are a lot of cache misses potentially impacting performance. This suggests the loop is a candidate for using the prefetching or streaming store instructions of the Streaming SIMD Extensions.

CONCLUSION

We have shown several unique methods of taking advantage of the performance capabilities of the Streaming SIMD Extensions of the Pentium® III processor. The performance of each method is very close to that of optimized hand-coded assembly, but the development costs associated with these methods are significantly lower than those of assembly programming. There are several other tools provided with the VTune™ Performance Enhancement Environment that are not described in this article. These are the Intel® Performance Library Suite, the Intel® Architecture Performance Training Center (please see reference [7]), and the Register Viewing Tool. Each of these tools provides further performance improvement capabilities and invaluable information on the use of the Streaming SIMD Extensions. Combined, these tools make the VTune Performance Enhancement Environment, Version 4.0, the definitive toolkit for Streaming SIMD Extensions programming.

REFERENCES

The following documents are referenced in this paper, and they provide background or supporting information for understanding the topics presented.

1. "AP-589: Software Conventions for the Streaming SIMD Extensions" at http://developer.intel.com/vtune/cbts/strmsimd/589down.htm. Order No. 243873-001, Intel Corporation, 1998.

2. Intel® C/C++ Compiler User's Guide, Order No. 664711-007, Intel Corporation, 1998.

3. *C++ Class Libraries for SIMD Operations*, Order No. 693500-002, Intel Corporation, 1998.

4. "AP-814: Software Development Strategies for the Streaming SIMD Extensions" at http://developer.intel.com/vtune/cbts/strmsimd/814down.htm. Order No. 243648-001, Intel Corporation, 1998.

5. "AP-833: Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel® C/C++ Compiler" at http://developer.intel.com/vtune/cbts/strmsimd/833down.htm. Order No. 243872-001, Intel Corporation, 1998.

6. "Intel® Architecture Software Developer's Manual, Volume 2: Instruction Set Reference" at http://developer.intel.com/design/pentiumii/manuals/243191.htm. Order No. 243191, Intel Corporation, 1999.

7. Intel Architecture Training Center at http://developer.intel.com/vtune/cbts/contents.htm, Intel Corporation, 1999.

8. Intel Architecture Optimization Reference Manual, Order No. 730795-001, Intel Corporation, 1999.

AUTHOR'S BIOGRAPHY

Joe Wolf is a staff software engineer with the Platform Tools Operation in the Microprocessor Products Group. He has been with Intel since 1996 and has worked in compiler development, technical marketing, and customer support. Before joining Intel, he was a compiler developer for nine years working in the supercomputing industry, focusing on vector and multi-processing and code generation. He received an M.S. degree in computer science from California Polytechnic State University, San Luis Obispo in 1987, and a B.S. degree in Management Information Systems from the University of Arizona in 1984. His e-mail is joe.wolf@intel.com.