Improvements in the Intel® Core™2 Penryn Processor Family Architecture and Microarchitecture

James Coke, Mobile Microprocessor Group, Intel Corporation
Harikrishna Baliga, Mobile Microprocessor Group, Intel Corporation
Niranjan Cooray, Mobile Microprocessor Group, Intel Corporation
Edward Gamsaragan, Mobile Microprocessor Group, Intel Corporation
Peter Smith, Mobile Microprocessor Group, Intel Corporation
Ki Yoon, Mobile Microprocessor Group, Intel Corporation
James Abel, Software Solutions Group, Intel Corporation
Antonio Valles, Software Solutions Group, Intel Corporation

Index words: SSE4.1, super-shuffle, radix-16, MOVNTDQA, streaming reads, CLI, STI, return stack buffer, super shuffle, SMC detection, Inclusion filter

Citation for this paper: Harikrishna Baliga, Niranjan Cooray, Edward Gamsaragan, Peter Smith, Ki Yoon, James Abel, Antonio Valles "Original 45mn Intel® Core™2 Processor Performance" Intel Technology Journal. http://www.intel.com/technology/itj/2008/v12i3/3-paper/1-abstract.htm (October 2008).

ABSTRACT

Intel® Corporation continuously seeks to improve the performance of each Intel Architecture microprocessor generation through architectural initiatives as well as process and circuit improvements. The predecessor to the Penryn family of processors, the 65nm Intel Core microarchitecture, codename Merom, led the competition in performance. This paper illustrates architecture techniques used by Intel in the family of processors to maintain this leadership position.

The new SSE ISA improvements (dubbed SSE4.1) are discussed, and we look at how the Penryn family of processors was able to utilize the Merom SSE enhancements to both enable SSE4.1 and improve legacy instructions. The instruction set is also examined to determine how instructions were targeted to improve various super-scalar workloads.

The paper explains how in the Penryn family of processors, the divide instructions are updated from Radix-4 to Radix-16. To minimize the hardware investment, integer divides are handled as floating point divides, so conversion techniques between integer and floating point are also discussed.

There were many other changes to improve the performance of the family of processors including improved data forwarding from stores to loads, removal of serialization from Set Interrupt Flag Clear Interrupt Flag (STI CLI), enhanced Self-modifying Code (SMC) detection, and "renaming" of the Return Stack Buffer.

INTRODUCTION

The family of processors is the latest production version of the Intel® Core™2 and Intel Xeon® product lines implemented on Intel's 45nm Hi-k silicon process. The Penryn microarchitecture is based on the 65nm Intel® Core™2 microarchitecture (codename Merom). A significant component of the Penryn value proposition was the addition of architectural and microarchitectural performance enhancements above the expected conversion to 45nm, so there was a strong desire to improve the performance of the architecture over that of its predecessor, the 65nm Intel Core™ 2 microarchitecture. There are many techniques to improve performance on a microprocessor, and each brings its own value to the final result. In this paper, we examine various architectural and microarchitectural changes that were used to attain the goal of improved performance. The majority of the performance improvements achieve a performance benefit on existing binaries, while the SSE4.1 changes require software changes to enable the added performance. A detailed discussion of the tradeoffs leading to these changes and the performance evaluation are documented in "45nm

Core 2 Silicon Performance Enhancements'' [1]. In this paper we focus on actual changes that led to a successful result.

SUPER SHUFFLE

One shuffler is better than two

Merom architecture dramatically improved SSE performance through a simple but highly effective method—doubling the width of SSE execution from 64 bits to 128 bits by instantiating two 64-bit execution units side by side and adding two small shuffle units to communicate between the 64-bit halves that handled only quad words. While increasing the execution width to 128 bits dramatically improved performance, the 64-bit ''wall'' between the execution halves was left in place.

The 64-bit wall caused some legacy instructions to be slower than would be expected. For example, the legacy instruction SHUFPS followed these steps in the Merom architecture:

1. Gathered all input DEST bits to [63:0] side of the ''wall.''
2. Gathered all SRC bits to the [127:64] side of the ''wall.''
3. Shuffled according to the immediate instruction.

SHUFPS output bits [127:64] always come from the SRC, and output bits [63:0] always come from the DEST, so we had to move SRC bits [63:0] to the upper half of the SSE execution unit. SHUFPS output bits [63:0] always come from the DEST input, so we had to move DEST bits [127:64] to the lower half of the SSE unit.

Some of the performance penalty of having three operations is covered by the Merom architecture having shuffle units on two ports as well as another SSE shuffler that handled only 64-bit data sizes on a third port.

The family of processor's solution to this problem is to merge the two shuffle units into a single Super-Shuffle unit that does not have a 64-bit wall. The Super-Shuffle is significantly more costly in terms of routing, but the added area cost is covered by merging the area from the two old shuffle units into the Super-Shuffle.

In the Penryn family of processors, the SHUFPS shuffling algorithm becomes:

1. SHUFPS!

While the Merom algorithm had a nominal throughput of 1, it used three operations. The implementation also has a throughput of 1, but it uses only one operation, leaving more execution bandwidth for other instructions. In the architecture we reduced the SHUFPS latency from 2 to 1.

We made similar improvements to SSE instructions PACKxSxx, PUNPCKxxx, PHADD*, PSHUFB, PALIGNR, PINSRW, PEXTRW, UNPCKLPS, UNPCKHPS, HPADDPS, HSUBPS, and PSHUFD.

INTRODUCTION TO SSE4.1

Many of the SSE4.1 instructions were created by noticing patterns in kernels that were commonly repeated using multiple instructions and that could be readily converted to a single instruction in hardware. Some of the instructions fill in gaps in the existing instruction set such as the new PMINxx, PMAXxx, PEXTRx, PINSRx, PACKUSDW, PCMPEQQ, and PMULLD. PMULDQ is the signed version of PMULUDQ. Please refer to the Software Developer's Manual for details [4].

The Super Shuffle breaking the 64-bit wall is a key enabler of many SSE4.1 instructions' performance improvement. The PMOVSX, PMOVZX, PEXTRx, PINSRx, and INSERTPS all require the new Super Shuffle to realize their full potential. Figure 1 shows PMOVZXDQ moving data across the 64-bit wall without a special operation to move data from the low 64 bits to the high 64 bits. The MPSADBW and PTEST instructions also depend on crossing the 64-bit boundary albeit in other execution blocks.

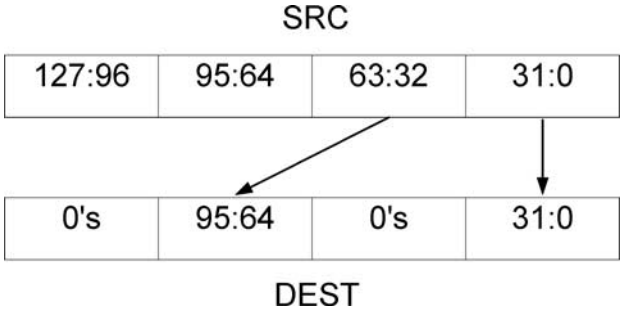

Figure 1: ***PMOVZXDQ crosses 64-bit boundary on the family of processors without a special 64-bit operation.***

SSE4.1 instructions DPPS, DPPD, and INSERTPS solve the problem of requiring additional instructions to selectively zero portions of the register. This zeroing effectively compresses two instructions into one for INSERTPS and four instructions into one for DPPS and DPPD.

As shown in Figure 2, DPPD and DPPS are the first floating-point SSE instructions to have multiple floating-point operations.

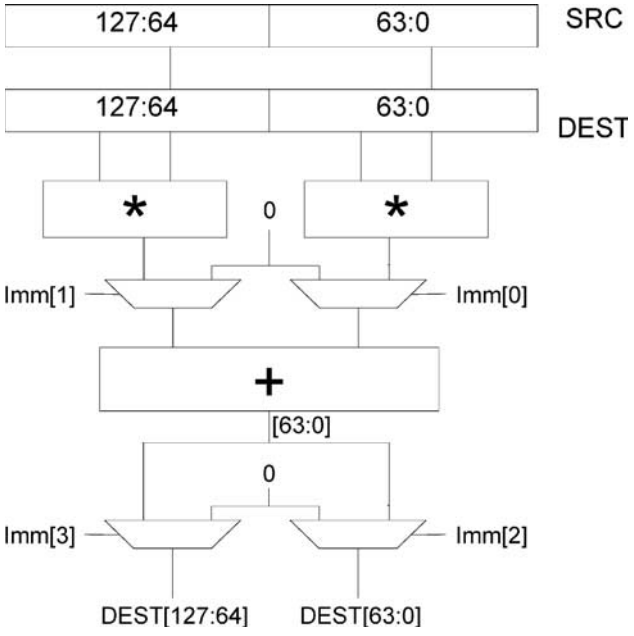

Figure 2: ***DPPD provides zeroing after both floating-point operations.***

Two new rounding instructions, ROUNDPS and ROUNDPD, provide rounding of floating-point values to integers and return the values in floating-point format. The rounding control is selectable from either the immediate instruction or MXCSR.RC. The user also has control over suppressing Precision Exceptions based on an immediate bit.

Prior to SSE4.1, SSE instructions always operated on subsets of 128 bits. PTEST is the first SSE operation to operate on the entire 128 bits as a single entity. PTEST is very useful for detecting all 0s and all 1s and reporting the result in the flags to simplify decisions.

MPSADBW performs a series of eight 4 X 4 SAD (Sum of Absolute Differences) operations across an 8-byte window of the destination. The starting point for the SRC and DEST windows is selectable using the immediate instruction.

PHMINPOSUW forms a very useful counterpart to MPSADBW because it finds the minimum word and returns both the value and position of the minimum word.

INSERTPS is a very generalized insertion between XMM registers. It allows any packed single quantity to be selected from the source and inserted into any position in the destination. The control to select the packed single position from the source and the position to insert the packed single in the destination are controlled by the instruction immediate. INSERTPS also allows selected packed single positions in the destination to be zeroed, also under control of the immediate.

EXTRACTPS sounds as if it should be the complement of INSERTPS, but it isn't. EXTRACTPS extracts to a General Purpose (GP) Register instead of to another XMM register, which makes EXTRACTPS very similar to PEXTRD. The only difference between EXTRACTPS and PEXTRD is the handling of REX.W. EXTRACTPS will zero extend the 32-bit value, whereas PEXTRD will become the 64-bit instruction PEXTRQ.

The BLENDxx and BLENDVxx instructions are a per-element select of the source or the destination register. Control of the select comes from the immediate for BLEND instructions, and for BLENDV instructions the control comes from the element sign bits of a third XMM register that must be XMM0.

STREAMING READS

Previous generations of Intel architecture processors supported a fast write mechanism from the processor to memory (such as to video and graphics memory) via streaming non-temporal writes. This greatly improved the write bandwidth from the processor to memory. However, up to now, Intel architecture was lacking a fast memory read mechanism for memory regions that are typically mapped as uncacheable with weak ordering—typically graphics video memory. The fast cacheable memory read mechanism cannot be utilized in this case because we do not want these data to be cached in the processor caches. In addition, we also do not want this type of data to expel useful data from the processor caches. The SSE4 instructions in the processor introduced a new streaming read IA instruction, MOVNTDQA, to fill this void. This new instruction, which is introduced on the second production stepping of the architecture, performs very high-bandwidth reads from weakly ordered, uncacheable (USWC) memory regions, typically used for graphics memory, without any pollution of the processor caches. This gives the programmers the ability to utilize the fast execution units inside the processor to operate on graphics-type data, which until now was not desirable due to very slow read bandwidth by the processor.

By allowing fast non-coherent transfers across PCIe or access to UMA graphics directly, streaming reads help increase the performance of analog and uncompressed high-definition video capture (20–30 percent of these workloads involve readback). It also makes hardware accelerated transcode (encode followed by decode) and video motion estimation feasible, with the fast

readback mechanism after HW accelerated decode in the northbridge.

The semantics of the MOVNTDQA instruction is to load an aligned 16-byte quantity. It is a demand load operation with a streaming hint. When this instruction is used to load 16 bytes from a memory region that is mapped as USWC, the processor automatically converts the load operation to a "streaming" load operation. By treating the load as a streaming load operation, the processor automatically converts the 16-byte load to a full cache line (64-byte) load operation and uses the maximum Front Side Bus (FSB) bandwidth to transfer the data from memory. For a 333-MHz FSB (1.333-GHz data transfer rate) we could achieve a 10.6-GB/s data transfer rate from USWC memory using MOVNTDQA loads, which is the same maximum bandwidth achievable by cacheable loads. This is compared to the maximum data transfer rate of 1.3 GB s for loading from USWC memory using non-streaming load instructions, assuming the FSB is 100 percent utilized.

When the processor treats a load operation as a streaming type (via MOVNTDQA), the entire USWC cache line (aligned 64 B) that contains the address of the load is loaded into an internal processor buffer, and the requested 16 bytes of data are served. The use of a temporary buffer for streaming along with a read-once policy helps maintain the uncacheable semantics of the USWC memory type. As shown in Figure 3, this internal buffer is not drained at the completion of the requested 16-byte load but is kept alive so that subsequent NT loads (MOVNTDQA) can be serviced from the same buffer rather than initiating new memory transactions. Thus, a program issuing four MOVNTDQA loads will be satisfied by a single buffer and a single memory transaction. A program that is designed to loop on four MOVNTDQA loads (such as operating on a block of memory, loading one cacheline at a time, and operating on it) can achieve data-read bandwidths up to the maximum FSB bandwidth. Once the entire contents of the temporary buffer are consumed (by four MOVNTDQA load operations), the buffer is automatically deallocated. Since the processor contains a limited number of internal temporary buffers, care must be taken while programming to not overflow or underutilize these resources.

Here is an example usage of MOVNTDQA instructions to efficiently utilize the streaming read buffers. Note that eax addresses are aligned to a line boundary.

```
MOVNTDQA xmm0, [eax]
MOVNTDQA xmm1, [eax+16]
MOVNTDQA xmm2, [eax+32]
MOVNTDQA xmm3, [eax+48]
PAVGB xmm0, xmm1
PAVGB xmm2, xmm3
PAVGB xmm0, xmm2                <Code 1>
```

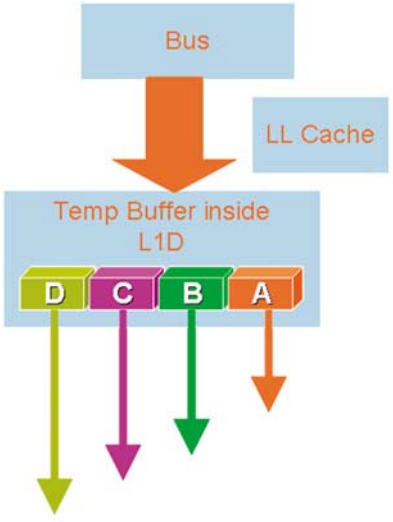

Figure 3: ***MOVNTDQA allows use of the full temp buffer before starting a new bus cycle.***

SSE4.1 EXAMPLES

SSE4 instructions were created to provide speedup on various types of applications. Two instructions in particular, MPSADBW and PHMINPOSUW, in combination with the Super Shuffler, can provide large performance improvements in block-matching algorithms commonly used in motion estimation. A detailed discussion on the block-matching performance (a 1.6x–3.8x function-level speedup) and how these instructions provide the performance improvements is documented in Penryn Silicon Performance [1].

Two other SSE4 examples are discussed in this section. First, we briefly showcase the measured performance of streaming loads [2] to conclude the discussion in the previous section. Then we discuss the DPPS DPPD instruction and usage models where DPPS DPPD will improve performance. We provide an example that uses the DPPS instruction to showcase how it and another SSE4.1 instruction (EXTRACTPS) can be used to speed up collision detection performance.

Streaming loads

In this section we continue the discussion from the previous section by briefly examining streaming loads measured results and optimization guidelines [2]. To maximize streaming load throughput, users need to utilize the streaming load buffers of two cores at the same time. That is, two software threads executing on

two different cores perform streaming loads from separate USWC parts at the same time and copy the data into separate WB cacheable memory buffers (see Figure 4). The WB buffers have to be small enough to fit in the first-level cache to minimize resource contentions, and the four streaming loads making up one cacheline (64 bytes) need to be done close together.

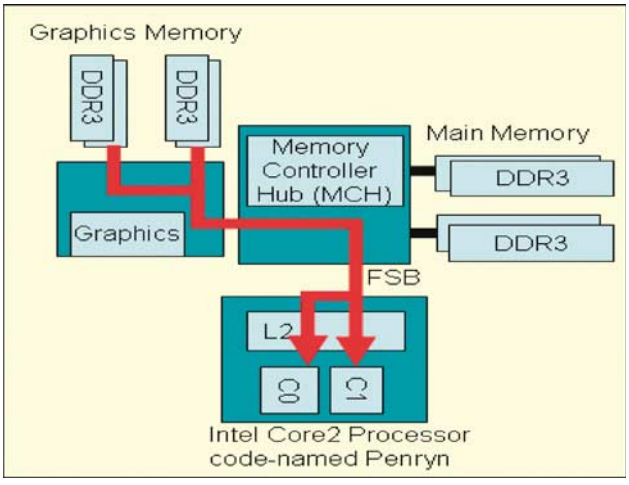

Figure 4: Example of streaming load: accessing graphic card memory and utilizing two threads to maximize memory throughput

Tests were conducted on a 45nm Intel Core 2 desktop processor (E7200) with a 1067-MT sec FSB.

The theoretical memory throughput (cacheable and uncacheable memory) can be calculated as follows:

Theoretical memory throughput

```
= FSB Transfer/sec * bytes/transfer
= 1067 MT/sec * 8 B/T
= 8.53 GB/sec
```

The single-threaded streaming load implementation that utilized one core's streaming load buffers was measured to provide approximately 50 percent of the theoretical memory throughput. The dual-threaded streaming load implementation that utilized the streaming load buffers of two cores was measured to provide approximately 90 percent of the theoretical memory throughput. Utilizing two core's streaming load buffers is the recommended way to get the highest memory throughput out of streaming loads.

Single-Precision Floating-Point Dot Product

The Dot Product of Packed Single Precision Floating-Point Values (DPPS) instruction and the DPPD instruction for Double Precision Floating-Point numbers can provide performance benefits in games, multimedia, and high-performance computing applications. This instruction has a high latency due to multiple numbers of operations being done at once. Thus, this instruction provides the most benefit in situations in which the Array of Structures (AOS) data layout is being used as opposed to the Structure of Arrays (SOA) data layout [3]. The AOS layout is usually not Single Instruction Multiple Data (SIMD)—friendly except for the horizontal instructions such as DPPS and HADDPS. Users can use these horizontal instructions to avoid the heavy data swizzling [3] costs in converting to the SOA data layout. An SSE3 implementation of a dot product of Vector Length 4 in the AOS format can be implemented by using the HADDPS instruction as shown:

```
void dot_product_vlength4_SSE3
(float *src,float *dst,int Count)
{
__asm {
      mov esi, dword ptr [src]
      mov edi, dword ptr [dst]
      mov ecx, Count
start:
      //a3, a2, a1, a0
      movaps xmm0, [esi]
      //a3*b3,a2*b2,a1*b1,a0*b0
      mulps xmm0, [esi + 16]
      //a3*b3+a2*b2,a1*b1+a0*b0,
      //a3*b3+a2*b2,a1*b1+a0*b0
      haddps xmm0, xmm0
      movaps xmm1, xmm0
      psrlq xmm0, 32
      addss xmm0, xmm1
      movss [edi],xmm0
      add esi, 32
      add edi, 4
      sub ecx, 1
      jnz start
   }
 }          <Code 2>
```

Notice that the SSE3 implementation of the dot product requires the MULPS + HADDPS + MOVAPS + PSRLQ + ADDSS instructions. The SSE4 implementation replaces all of these instructions with one: DPPS (Code 3).

```
void dot_product_vlength4_SSE4
(float *src,float *dst,int Count)
{
__asm {
      mov esi, dword ptr [src]
      mov edi, dword ptr [dst]
      mov ecx, Count
start:
      movaps xmm0, [esi]
      dpps xmm0, [esi + 16]
      movss [edi],xmm0
      add esi, 32
      add edi, 4
      sub ecx, 1
      jnz start
  }
}            < Code 3>
```

Table 1 shows the measured performance of the two different dot product implementations in AOS data layout as compared to the C implementation. The DPPS instructions can provide performance speedups on multiple vector matrix operations that require a dot product such as vector normalization [3] and collision detection.

Table 1: Performance of dot product implementations.

Implementation	Cycles/Loop	Speedup over C
C	9.8	1.0 ×
SSE3	7.8	1.26 ×
SSE4	5.7	1.72 ×

COLLISION DETECTION

The dot product can be used for collision detection in games. This is another example of using the DPPS instruction in an AOS layout. In this example, the DPPS instruction is used to speed up collision detection of two spheres. To explain collision detection, consider two circles. The circles are said to have collided if the sum of radii is greater than or equal to the distance between the centers (Figure 5). This same equation applies to spheres, except in that case there is a z-axis that contributes to the distance formula.

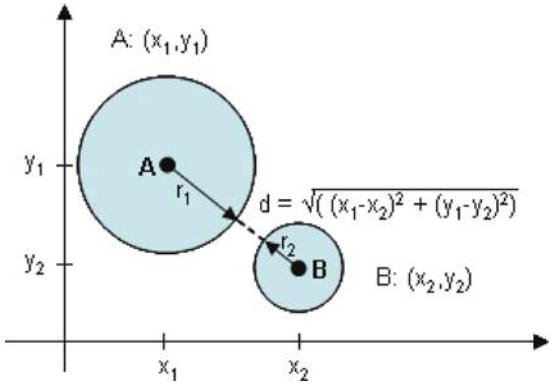

Figure 5: ***Circles/spheres collide if the sum of the radii is greater than or equal to the distance between the two centers.***

Two spheres collide if

```
The distance between two centers <= sum
of radii
sqrt((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2)
<= r1 + r2
(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2 <=
(r1 + r2)^2
Dot Product (point A, point B) <=
(r1 + r2)^2
```

$$A * B <= (r1 + r2)^2 \quad (1)$$

As an example, imagine a fast-moving, hot fire particle about to incinerate other objects/particles. Collision detection can be used to find out which of these objects comes in contact with the fire particle and will need to be set on fire (see Figure 6).

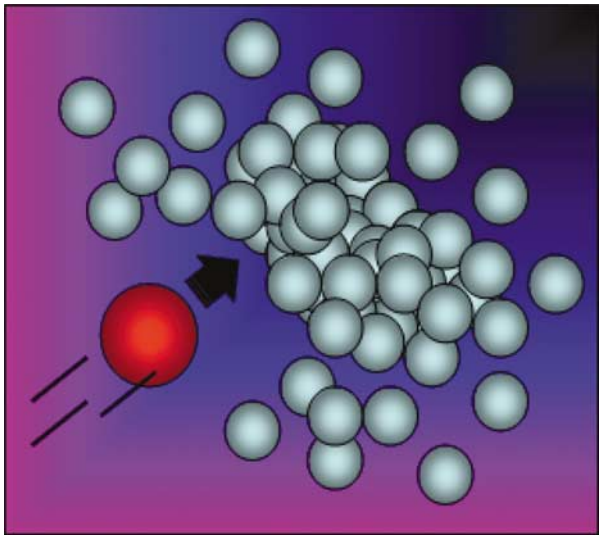

Figure 6: ***Collision detection example: one fast-moving fire particle is about to slam into and incinerate other objects/particles.***

The C code implementation (Code 4) uses Eq. 1 to detect sphere collision between the fire particle and a thousand other particles.

```
struct VEC3
{
float x,y,z;
float dot() const
{return x*x + y*y + z*z;}
};
VEC3 operator -
(const VEC3 a, const VEC3 b)
{
VEC3 res;
res.x = a.x - b.x;
res.y = a.y - b.y;
res.z = a.z - b.z;
return res;
}
struct SPHERE
{
VEC3 center;
float rad;
};
void sphere_collision_C
(SPHERE *sp1,SPHERE *sp,int *coll)
{
for(int i=0; i<gNumSpheres;i++)
{
float center_distance_squared =
(sp1[i].center-sp->center).dot();
float radii_sum_squared =
(sp1[i].rad + sp->rad)*
(sp1[i].rad + sp->rad);
if(center_distance_squared <
radii_sum_squared)
{
//which spheres collided
coll[i]++;
    }
  }
}
```

<Code 4>

The collision detection code can be optimized with SSE4.1 instructions. One single DPPS instruction can be used to make the three different calculations in Eq. 1: the dot product of AB: $(x1-x2)^2+(y1-y2)^2+(z1-z2)^2$, the radii sum: $(r1+r2)^2$, and the subtraction of the radii sum from the dot product.

$$res = A * B - (r1 + r2)^2$$

Collision occurred if res's sign-bit is set (Code 5).

```
int sphere_collision_intrinsics (SPHERE
*sp1,SPHERE *sp,int *coll)
{
int res;
__declspec(align(16))
static long _mask[4] =
{0,0,0,0x80000000};
__m128  s,s1,s2;
__m128 _mask128 =
*(__m128*)_mask;
s = _mm_load_ps((float *)sp);
//set sign bit to add: r1 - (-r2)
```

```
s = _mm_xor_ps(s,_mask128);
for(int i=0;i<gNumSpheres;i++)
{
s1 = _mm_load_ps((float *)sp1);
s1 = _mm_sub_ps(s1,s);
//set sign bit on radii sum
s2 = _mm_xor_ps(s1,_mask128);
//s1[0-31] =
//center_distance_sqaured -
//radii_sum_squared
s1 = _mm_dp_ps(s1,s2,0xff);
//get sign bit of subtraction
res = _mm_extract_ps(s1,0);
res >>= 31;
//coll if radii_sum_squared
// > center_distance_squared
res &= 1;
coll[i] += res;
}
}                          <Code 5>
```

To use a single DPPS instruction to do all of this, we had to use a few SIMD tricks. First, we laid out the data so x,y,z,r of the sphere could be loaded with one aligned load. Then we used a mask to modify the sign bit of one of the radii before the packed subtract. We did this so that the subtract operation actually causes an addition, (r_1-(-r_2)). Then we used the mask again to modify the sign bit of one of the radii sums before the DPPS instruction. This causes the radii sum squared to be subtracted from the dot product of A and B.

These are the contents of the __m128 variables before the DPPS instruction:

```
//sign bit set on upper 32-bytes of _m128
s2 = [-(r1+r2), z1-z2, y1-y2, x1-x2]
s1 = [r1+r2, z1-z2, y1-y2, x1-x2]
```

These are the contents of s1[0-31] after the DPPS instruction: If s1[0–31] is negative, then the radii sum squared is greater than the dot product of A and B, and a collision occurred.

Another SSE4.1 instruction, EXTRACTPS, is used to extract the single precision floating-point value from an XMM register to a GP register to check if the sign-bit is set. The EXTRACTPS instruction removes the branch and enables GP registers to be used to do data manipulations. Both the DPPS and EXTRACTPS instructions provided the 1.5x speedup over C as shown in Table 2.

Table 2: SSE4.1 Collision detection speedups.

Implementation	Cycles/Iteration	C Speedup
C	14.3	1.0 ×
SSE3	17.7	0.8 ×
SSE4	9.5	1.5 ×

The analogous SSE3 solution has to use MULPS + HADDPS + MOVAPS + PSRLQ + ADDSS instructions as shown in Code 2 for the dot product. It also has to move the result to a floating-point register to do the comparison similar to the C-code. The SSE3 implementation has too large of a latency to provide a speedup over C.

In this section we provided two SSE4.1 examples. We discussed the measured performance of the streaming loads and provided the recommended usage scenario. We also discussed the DPPS instruction and the recommended usage scenarios for this instruction as demonstrated in the collision detection example.

NEW RADIX-16 DIVIDER

The new Radix-16 floating-point divider with variable latency Radix-16 integer divide capability replaces the Merom Radix-4 floating point divide and Radix-2 square root and integer divide hardware. The preceding algorithm dated back to the Pentium® divide implementation.

Motivation and implementation

Divide hardware is costly both from die size and performance perspectives. Its large size makes it prohibitive to add multiple units on a single core. On the other hand, the long latency and low throughput of divides has a dramatic impact on CPU performance. The implementation provides a remedy for the latter by reducing the number of loop iterations for a single divide.

In the Sweeney, Robertson, and Tocher divide algorithm (SRT) [5–8], the divide operation is broken up into three parts: pre-processing, loop, and post-processing. The loop accounts for the predominant source of the latency and prevents subsequent micro

operations from utilizing the hardware in a pipelined manner. Specific implementations may choose to pipeline consecutive operations over the three parts (for example, a second operation's loop may be implemented to begin once the first enters post-processing). However, any de-pipelining pales in comparison to the loop latency's impact to divide throughput. The latency of different Radix implementations is shown in Table 3.

Table 3: Divide and Square Root latencies.

	Latency in cycles			
	Pre + post processing	R2 loop	R4 loop	R16 loop
Single precision	5 to 6	28	14	7
Double precision	5 to 6	57	29	15
Extended precision	5 to 6	68	34	17

The loop latency has a direct correlation to the number of quotient mantissa bits in any given precision. In Radix-2, one quotient bit is calculated in every cycle; thus, the number of cycles in the loop equals the number of mantissa bits. In Radix-4, two quotient bits are calculated every iteration. For Radix-16, four quotient bits are calculated. It can be seen why the enhanced divider had a profound impact on performance: divides were up to 1.75 times faster, and square roots were up to 3.3 times faster.

The new variable latency integer divide algorithm utilizes the underlying Radix-16 floating-point divider without the need to implement a different integer algorithm or build a separate integer divide unit. The same exact algorithm can be used on integer numbers after they undergo an integer normalization and shift amount recording, prior to the pre-processing performed by modified existing hardware.

In addition to improving performance by moving from Radix-2 integer divides on the Merom processor to Radix-16 on the family of processors, the integer divide operation can finish sooner than what is depicted in Table 3 depending on the specific data operands. Since the loop iteration count depends on the number of quotient bits produced, and given that integer operations produce an integer quotient and separate remainder, the integer divide algorithm stops the loop after the quotient is created and begins post-processing. However, due to other existing microarchitectural restrictions, the total divide micro operation latency is at least 11 cycles, excluding early out conditions (such as 0 div by n). Thus when there are 17 or more quotient bits produced but less than 29 bits (for r m32; less than 61 bits for r m64), then there is a further reduction in latency over the previous algorithm.

Challenges

Historically, implementing high-Radix fast dividers has been a design challenge. Finding the correct balance between implementing a high-Radix quotient and a fast-quotient selection logic (QSL) is a difficult task. In the family of processors, we addressed this by applying a new digit-redundant structure and an implicit bias bits concept to ordinary basic divide algorithms, such as Non-performed on a binary digit basis, without rippling the carry forward. Thus, each digit produces two outputs: the sum and the carry. After all of the redundant arithmetic is performed, completion adders are employed to roll in the carry bits in the final step.

As can be seen in Figure 7 the divider is essentially double pumped, producing two bits of quotient every phase to yield four bits per cycle. Contrast this with the previous Radix-4 design in Figure 8 in which two bits of quotient were produced per cycle. By using the new digit-redundant structures in conjunction with the implicit biasing for selecting the quotient, an efficient and fast way of selecting the quotient can be achieved with a small number of bits of the partial remainder and the divisor. This will allow for fast redundant implementations of the internal loop computation and the quotient selection logic. The simplified quotient selection logic that is based on only a few bits of the estimated value of the partial remainder will in turn allow a very fast implementation that enables a multiple of these QSL blocks to exist in the same cycle, allowing for very high Radix dividers. The paths in the main loop and QSL are equalized by overlapping them, and they were targeted to a delay of just MUX delay plus truncated adder/comparator delay on either path.

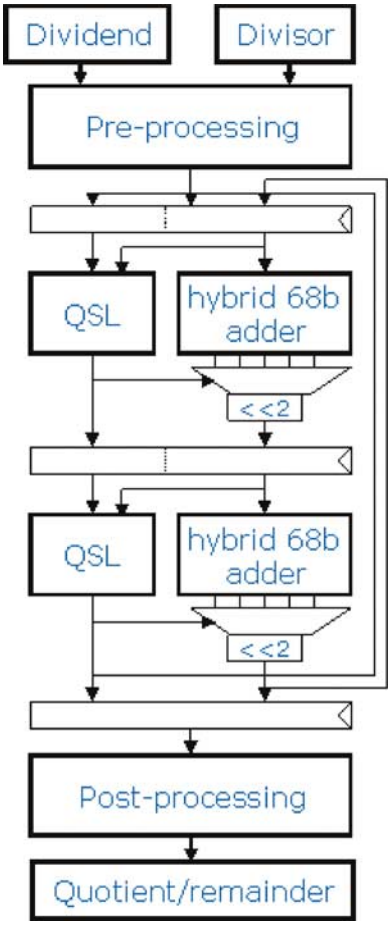

Figure 7: *New divider microarchitecture.*

CLI/STI PERFORMANCE TUNING

In the Software Developer's Manual, both the Clear Interrupt Flag (CLI) and the Set Interrupt Flag (STI) macro-instructions are said to be non-serializing. However, due to past microarchitectural simplifications, both instructions serialized the execution pipeline, leaving an unoptimized performance situation. The serialization was deemed necessary to ensure that an updated copy of the Interrupt Flag (IF) in System Flags was ready to be evaluated at the end of every macro-instruction. Furthermore, the IF masking that is done at the end of STI needs to occur only when the IF is transitioning from clear to set and needs an updated copy of IF at the beginning of the STI instruction.

It was determined that for multi-tasking operating systems, the IF can get frequently masked and unmasked during atomic operations to prevent other processes from obtaining the context in the middle of modification. As such, there can be a noticeable performance degradation due to the aforementioned CLI STI serialization. Pre-silicon performance simulations showed a 1.3 percent improvement on productivity workloads if this penalty was removed.

On the processor, instead of post-serializing on a CLI or STI, a serialization occurs only when the new IF value is consumed and only if the new value is not yet updated. Additionally, we added new dedicated hardware to the retirement logic to detect whether the IF transitions from clear to set during an STI, in order to

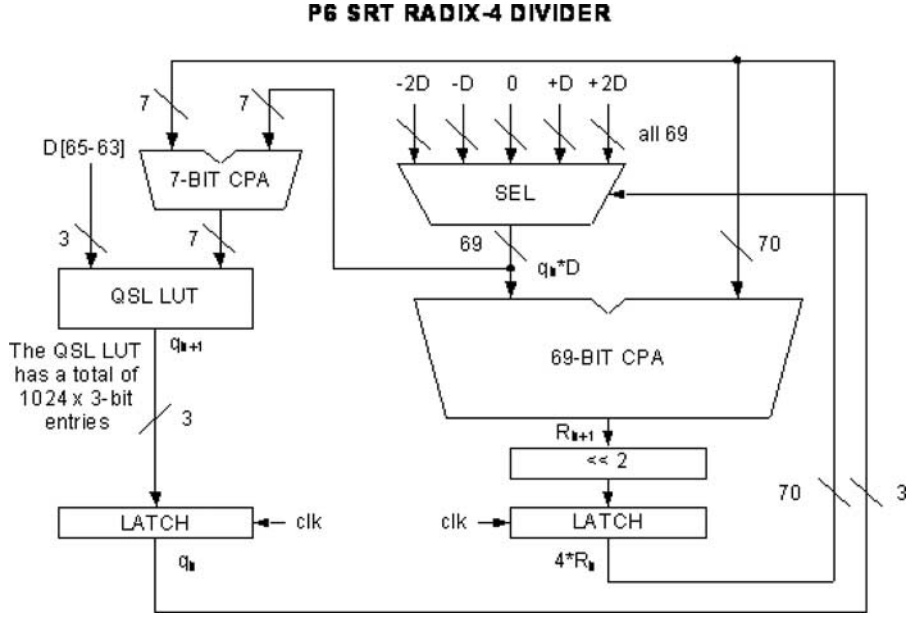

Figure 8: *Previous divider microarchitecture.*

avoid a new pre-serialization condition. The results are that the throughput of a CLI is 5 cycles, the throughput of an STI is 8 cycles, and a CLI-STI pair is 13 cycles, yielding a 2.5× improvement over the Merom architecture performance.

INCLUSION FILTER

The inclusion filter enables detection of instructions in the processor pipeline, mainly to support self-modifying code (SMC). To reduce design impact on this timing-sensitive area of the processor pipeline, detection techniques are architecturally minimized to provide pessimistic estimates. The goal is a logically optimized solution in which false SMC detection is sufficiently uncommon such that the resulting performance loss is negligible.

Motivation

As the processor pipeline capacity increases, the inclusion detection solution needed to be re-examined. The pipeline capacity includes instructions in all stages and structures between instruction fetch and retirement, which increased substantially when the issue width was increased for the Merom architecture from 3 to 4. The increased instruction capacity resulted in increased false SMC detection conditions during Merom silicon testing, which tended to have a more limiting impact on server performance. For example, Transaction Processing Performance Council Benchmark C performance increased by 2 percent with inclusion checking disabled. The Inclusion Filter in the Penryn processor significantly reduces false SMC detection by using an alternative technique to filter from the existing detection mechanism the most common false detection scenarios.

Solution

Most instructions in the pipeline will also naturally exist in the Instruction Cache (ICache), so the Inclusion Filter monitors ICache activity to algorithmically identify states in which this common property is guaranteed (Figure 9). Snoops in this state can then be filtered from the existing inclusion-detection mechanism, and this combination virtually eliminates false SMC detection.

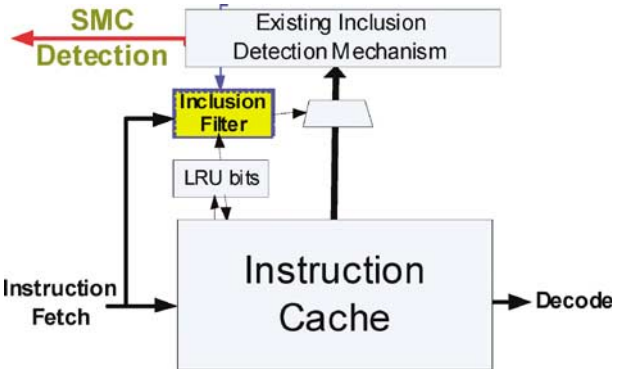

Figure 9: ***Inclusion Filter reduces false SMC detection.***

To increase confidence in this new microarchitectural solution, it was essential to minimize design complexity. To reduce logic risk and validation requirements, the Inclusion Filter has a single functional output. To avoid the risk of frequency degradation, it is logically separated from the existing ICache structure (which is ideal for separating logic vs. process-related debug) and uses only non-timing-critical signals, such as the ICache LRU bits.

For example, a property of the ICache pseudo-LRU algorithm is that for an X-way configuration, an accessed entry will not be evicted until at least log2 (x) different entries in the same set have been accessed. Therefore, for an 8-way cache, each set is allowed to filter at least three ICache evictions prior to resorting to the existing inclusion detection mechanism. Determining a "different entry" can be accomplished without additional storage by detecting a change of LRU bits. Monitoring changes to specific LRU bits and other control logic can increase the limit substantially using other similar properties.

When the Inclusion Filter is saturated and finally allows the existing mechanism to be used, it is more beneficial to have it return to its reset state than to continue filtering. From a reset, the average cycles needed for the Inclusion Filter to resort to the existing inclusion mechanism is 50 times greater than the cycles needed to ensure that a fetched instruction is no longer in the pipeline. Therefore, when the Inclusion Filter is finally saturated, it takes the opportunity to completely reset its state, but it disables filtering until it is certain that all instructions in the pipeline at the time of this reset have been retired.

In effect, a small window is opened during which the existing detection method is used, then it is closed for a very long time (98 percent closed on average). This translates to a 98 percent reduction in false SMC detection, and near optimal performance.

RENAMED RSB

A Renamed Return Stack Buffer (RRSB) was added to improve performance by increasing return prediction accuracy. The goal was to supplement the existing RSB by providing a recovery mechanism from a common source of RSB corruption.

Background

A single function (or procedure) can be called from multiple places within a program by using a "CALL" instruction. Exiting the function back to the calling program can be done with a "RET" (return) instruction. The CALL instruction is similar to a direct jump that also pushes the RET address onto the stack (in memory). The RET instruction is an indirect jump whose target address is popped from the stack.

The processor's Branch Prediction Unit (BPU) shares both its bimodal prediction resources to accurately predict the existence of CALL or RET instructions and also its Branch Target Buffer (BTB) to predict the target of a direct CALL. However, the target of a RET instruction is dependent on the CALL, so the Return Stack Buffer (RSB) is used.

All P6 microprocessors have implemented the RSB as a simple push pop stack structure. This "classic" RSB (CRSB) has the following basic behavior:

1. The BPU uses its Linear Instruction Pointer (LIP) to predict a CALL instruction.
2. The BPU "pushes" the CALL's Next Linear Instruction Pointer (NLIP) onto the CRSB stack.
3. The BPU predicts the target of the CALL from the BTB and redirects the instruction flow.
4. Later, the BPU predicts a RET instruction based on its LIP.
5. The BPU predicts the target of the RET from the CRSB and redirects the instruction flow.

CRSB corruption

Useful RET predictions in the CRSB are sometimes overwritten by bogus speculative updates. These bogus updates should be corrected after a branch misprediction to ensure accuracy. This requires saving the CRSB state for each potential misprediction and restoring that state after misprediction recovery. Practically, however, we can save only the CRSB Top-Of-Stack (TOS) pointer that is stored in the Branch Information Table (BIT). When the CRSB TOS is restored from the BIT, the contents may have been overwritten while traversing down the bogus path. For instance, if the bogus path has a RET followed by a CALL, a valid return address will be overwritten that will later result in a performance penalty. The TOS pointer will be restored, but the CRSB contents are corrupted. Figure 10 (top) describes this common CRSB corruption scenario.

Renamed RSB implementation

To address this corruption, we added the "Renamed RSB" (RRSB) to the Penryn family of processors. The RRSB is similar to the CRSB, but it incorporates an additional pointer (Alloc) and a linked-list structure for updating the TOS. Figure 10 (bottom) shows how the RRSB is able to recover from bogus updates. The pointers are updated as follows:

- The CALL NLIP is written to the Alloc entry. The TOS pointer is adjusted to point to the Alloc entry, and then the Alloc pointer is incremented (Column 3 in Figure 10). The Alloc pointer never decrements. The TOS linked-list is updated to retain the previous TOS.

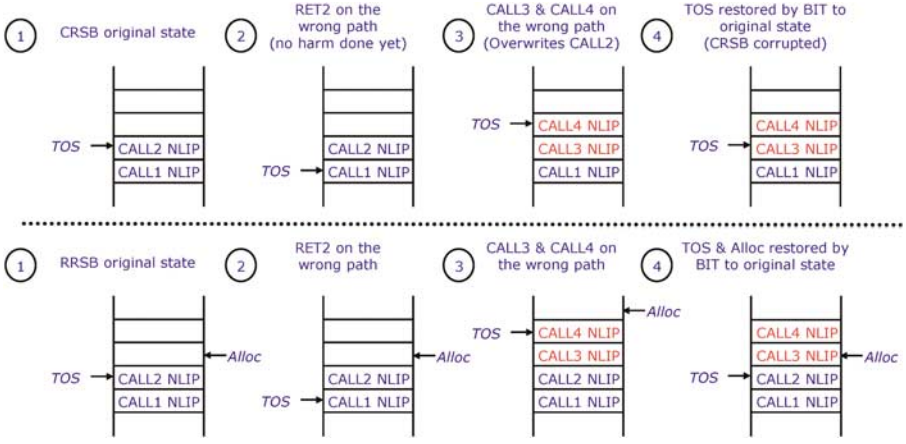

Figure 10: ***CRSB vs. RRSB.***

- The RET target is read from the TOS entry and uses the linked-list to adjust the TOS pointer to the previous TOS. The Alloc pointer is not updated on RET instructions.

The CALL NLIP is never overwritten and therefore retains entries that may be lost by the CRSB.

While the RRSB is more accurate on the speculative path, it overflows (wraps) more quickly since Alloc never decrements. Therefore, the return prediction defaults to the CRSB when RRSB detects the wrap condition. We added a 16-entry RRSB to the architecture that works in conjunction with the 16-entry CRSB as shown in Figure 11.

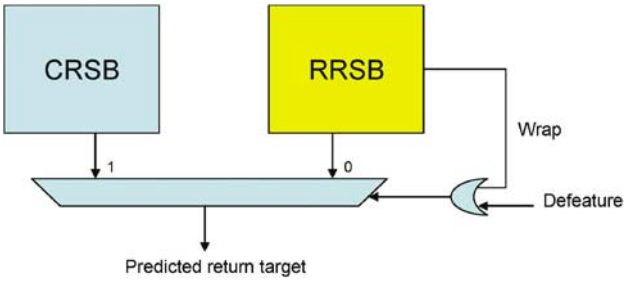

Figure 11: ***RRSB implementation.***

CONCLUSION

The improvements described in this paper should provide the reader a better understanding of the value architectural and microarchitectural enhancements have brought to the marketplace above and beyond a silicon process improvement. Each feature improves some aspect of performance so that in conjunction with the frequency improvement, the end user will realize real value in the product.

For a quantification of the performance improvement, please refer to the paper "Original 45nm Intel® Core™2 Processor Performance" in [1].

ACKNOWLEDGEMENTS

We thank the following contributors to our paper: Ronen Zohar for providing the collision detection example; Ashish Jha for providing the streaming load kernel white paper; Lihu Rappoport for his contribution to the Renamed RSB; and Mohammad Abdallah for his contributions to the development of the divider and SSE4.1.

REFERENCES

[1] Nisar A., Ekpanyapong M., Valles A., and Sivakumar K., 45nm Intel® Core™2 Processer Performance, Intel Technology Journal, Vol. 12, No. 3, 2008.

[2] Jha A. Yee D. Increasing Memory Throughput With Intel® Streaming SIMD Extensions 4 (Intel(4) SSE4) Streaming Load. http://softwarecommunity.intel.com/articles/eng/1248.htm.

[3] Intel® 64 and IA-32 Architectures Optimization Reference Manual. At http://www.intel.com/products/processor/manuals/.

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual. At http://www.intel.com/products/processor/manuals/.

[5] Atkins D.E. 'Higher radix division using estimates of the divisor and partial remainders.' IEEE Transactions on Computers, 1968; C-17: 925–934.

[6] Parhami B. Tight upper bounds on the minimum precision required of the divisor and the partial remainder in high-radix division. IEEE Transactions on Computers, Vol. 52, No.: 11, November 2003, pp. 1509–1514.

[7] Wey C.-L., Wang C.-P. Design of a fast radix-4 SRT divider and its VLSI implementation. Computers and Digital Techniques, IEE Proceedings, Vol. 146, No. 4, July 1999, pp. 205–210.

[8] "Design issues in radix-4 SRT square root & divide unit" Burgess N., Hinds C. "Signals, Systems and Computers." *Conference Record of the Thirty-Fifth Asilomar Conference,*Vol. 2, November 4–7, 2001, pp. 1646–1650.

AUTHORS' BIOGRAPHIES

Jim Coke is a Staff Architect and Microcoder in Intel's Mobile Microprocessor Group in Folsom, CA. He received his B.S.E.E. degree from the University of Michigan and his M.S.C.E. degree from the National Technological University. Jim joined Intel in 1982 and has worked in product engineering, design, and architecture. Jim was the lead implementation architect for SSE4.1. His primary interests are microcode and microarchitecture. His e-mail is James.S.Coke at intel.com.

Hari Baliga is a Senior Staff Engineer in Intel's Mobile Microprocessor Group in Folsom, CA. He received his Bachelor of Engineering degree from Regional Engineering College, Surathkal in India and his M.S. degree from Arizona State University in Tempe. Hari joined Intel in 1996 and has worked on many microprocessors developed by the Folsom Design Center. His e-mail is harikrishna.baliga at intel.com.

Niran Cooray is a Senior Staff Architect in Intel's Mobile Microprocessor Group in Folsom, CA. He received his B.Sc. degree from the University of

Moratuwa in Sri Lanka and his M.S. degree from Northeastern University in Boston. Niran joined Intel in 1995 and has worked on many microprocessors developed by the Folsom Design Center. Niran worked as a Senior Design Leader on P6-based microprocessors before moving on to become a microarchitect for the Intel® 45nm Core™2 Duo processor. His email is Niranjan.L.Cooray at intel.com.

Edward Gamsaragan is a Staff Architect in Intel's Mobile Microprocessor Group in Folsom, CA, working there since 1995. For the Penryn family of processors, he was the microarchitect responsible for the out-of-order and execution clusters. His current focus is on next-generation memory technologies. Ed holds a B.S.E.E. degree from the University of California at Los Angeles. His e-mail is Edward.Gamsaragan at intel.com.

Peter Smith is a Senior Architect with Intel's Mobile Platform Group in Folsom, CA. For the Penryn family of processors, he was the architect responsible for the front-end and MSID clusters. His previous experience includes software design, system administration, circuit design, silicon debug, and performance analysis. His primary interests include probability theory, heuristics, and creative problem solving. Peter received his B.S. degree from the University of Wisconsin-Madison and joined Intel in 1996. His e-mail is Peter.J.Smith at intel.com.

Ki Yoon is a Senior Staff Architect with Intel's Mobile Platform Group in Folsom, CA focusing on microprocessor microcode and debug. Ki developed microcode and played a key role in system debug on the Intel Pentium® III and the Intel® Core™2 processor generations. Most recently, Ki was involved in the definition of the 45nm Intel Core 2 Duo processor architecture and Intel Virtualization Technology. He received his B.S. degree from the University of Texas at Austin in 1994. His e-mail is ki.w.yoon at intel.com.

James Abel is a Principal Engineer in Intel in Chandler, Arizona. James obtained a Bachelor's Degree in Electrical Engineering from Bradley University in Peoria, Illinois in 1983 and a Master's Degree in Computer Science from Arizona State University in 1991. His interests include computer architectures, performance analysis tools, digital signal processing, and multimedia algorithms. His email is James.C.Abel at intel.com.

Antonio Valles is a Senior Software Engineer in Intel in Chandler, Arizona focusing on broad and in-depth pre-Si and early-Si tests of Intel microprocessors and chipsets. Antonio has created multiple internal pre-Si and post-Si tools and kernels for performance analysis and coordinates the development of tuning guidelines for the processors. He received his Bachelor's Degree in Electrical Engineering from Arizona State University in 1997. His email is antonio.c.valles at intel.com.