Preface

Lin Chao
Editor
Intel Technology Journal

This Q2, 1999 issue of the Intel Technology Journal focuses on the Pentium® III processor. In late 1995, two factors influenced Intel's processor roadmap. The first one was the emerging importance of 3D capabilities for the volume PC market. These 3D capabilities were important for games and workstation software. Floating point computation is the heart of 3D geometry capabilities. With the introduction of the Pentium® Pro (P6) architecture, the floating point performance was good enough to make 3D really viable for PC's. The second influencing factor was the realization that there was an opportunity to strengthen Intel's processor roadmap for the late'98/early'99 timeframe with a P6 based proliferation. The question was how to do this while at the same time addressing the emerging importance of 3D.

In February 1996, the product definition team at Intel presented Intel's executive staff with a proposal for a single-instruction-multiple-data (SIMD) floating point model as an extension to IA-32 architecture. In other words, the "Katmai" processor, later to be externally named the Pentium III processor, was being proposed. The meeting was inconclusive. At that time, the Pentium® processor with MMX instructions had not been introduced and hence was unproven in the market. Here the executive staff were being asked essentially to "double down" their bets on MMX instructions and then on SIMD floating point extensions. Intel's executive staff gave the product team additional questions to answer and two weeks later, still in February 1996, they gave the OK for the "Katmai" processor project. During the later definition phase, the technology focus was refined beyond 3D to include other application areas such as audio, video, speech recognition and even server application performance. In Febuary 1999, the Pentium III processor was introduced.

In this Q2, 1999 issue of the Intel Technology Journal, you will gain important insights into the features and capabilities of the Pentium III processor. The first and second papers describe the Streaming SIMD Extensions and the microarchitecture implementation challenges. The third paper discusses the processor serial number feature. The fourth and fifth papers cover tuning applications for Streaming SIMD Extensions and an optimized 3D architecture stack for performance. And, finally, in the fifth paper, programming methods for the Streaming SIMD Extensions are described.

The Internet Streaming SIMD Extensions

Shreekant (Ticky) Thakkar, Microprocessor Products Group, Intel Corp.
Tom Huff, Microprocessor Products Group, Intel Corp.

ABSTRACT

The paper describes the development and definition of Intel's new Internet Streaming SIMD Extensions introduced on the Pentium® III processor. The extensions are divided into three categories: SIMD-FP, New Media, and Streaming Memory instructions. The new extensions accelerate the 3D geometry pipeline by nearly 2x that of the previous-generation processor while enabling new applications, such as real-time MPEG-2 encode. The Pentium III processor implementations achieved the desired goal at a modest 10% increase in die size. The definition achieved the short-term goal while still providing the performance scalability needed for future implementations.

INTRODUCTION

In late 1995, it was becoming clear that visual computing would assume an increasingly important role in the volume PC market segments. To address this need, Intel launched an initiative in visual computing aimed at the 1999 volume PC market segments. This required a balanced platform for 3D graphics performance in order to scale from arcade consoles to workstation applications. Floating-point (FP) computation is the heart of 3D geometry; thus, speeding up FP computation is vital to overall 3D performance.

An increase of 1.5 – 2x the native FP performance in IA-32 processors was required in order to have a visually perceptible difference in performance. 3D graphics applications require the same computation to be performed on 3D data types (vertices), making it amenable to a *Single Instruction Multiple Data* (SIMD) parallel computation model. This is the most cost-effective way of accelerating FP performance of 3D applications in general purpose processors, and it is similar to the acceleration for the class of integer applications provided by the Intel® MMX™ technology extensions [1]. Thus, a SIMD-FP model was selected for the IA-32 extension.

The Instruction Set Architecture (ISA) scope expanded beyond 3D geometry to include feedback on the usage of the MMX technology from independent software vendors (ISV), as well as support for 3D software rendering, video encode and decode, and speech recognition. Cacheability instructions were added to increase concurrency between execution and memory accesses. In all, 70 new instructions and a corresponding new state were added to IA-32 architecture; this is the first new state added since the Intel® i386™ processor added the x87-FP. This paper describes the architectural features and instructions selected as part of the IA-32 definition process.

ARCHITECTURE DEFINITION

2-Wide Versus 4-Wide SIMD-FP

The key component of the new extension was accelerating single precision floating-point computation, which involved the choice of either 2-wide or 4-wide 32-bit floating-point data parallel computations. This crucial decision is discussed later in this paper. This choice shaped key aspects of the new architecture.

An initial feasibility study of SIMD-FP in IA-32 done by the development team indicated that a new microarchitecture could perform 4-wide single precision floating-point operations per clock period, without adding significant complexity or cost to die size. The basic approach was to double-cycle existing 64-bit hardware. The performance benefit of selecting this format was to deliver a realized 1.5 - 2x (or greater) speedup for the geometry pipeline, which supports much greater levels of visual realism.

Another solution for achieving similar gains would be via a superscalar design, by adding execution units. Although this approach may be simpler for a programmer, it incurs a much larger area and timing cost, by increasing busses, register file ports, execution hardware, and scheduling complexity.

Implementing a datapath greater than 128 bits was also not viewed as a reasonable option, again due to balancing cost against performance benefits. Busses and registers were already 80 bits wide due to x87-FP; 128 bits represented only an incremental increase, whereas 256 bits would have a much larger impact. As mentioned, 128-bit execution is actually performed in 64-bit chunks and yet the peak rate of one 128-bit operation can be sustained if, as commonly occurs, instructions alternate between different execution units (i.e., add-multiply-add-multiply). Implementing a 256-bit wide SIMD unit would require doubling the width of execution units in order to still attain peak throughput in the same manner. Increasing SIMD-width beyond 128 bits would also require an increase in memory bandwidth in order to feed the wider execution units. There is a cost to this additional bandwidth, which may not follow Moore's Law progression, required by other application areas. Also, since the primary focus for the extensions has been on 3D geometry, greater than 4-wide parallelism may offer diminishing returns, since triangular strip lengths in current desktop 3D applications tend to be fairly small (i.e., on the order of 20 vertices per strip).

Related to this decision were the following two issues:

- state space: overlap or new registers
- Pentium® III processor implementation

State Space

There were two choices: overlap the new state with the MMX/x87 FP registers or add a new state. One big advantage of the first choice is that it would not require any operating system (OS) changes, just like the MMX^{TM} technology extension. However, there were many disadvantages with this choice. First, we could only implement four 4-wide 128-bit registers in the existing space since we only had eight 80-bit registers, or we could go to a 2-wide format, thus sacrificing potential performance gains. Second, we would be forced to share the state with MMX registers, which was an issue for the already register-starved IA-32 architecture. The complexity of adding another set of overlapped state was overwhelming.

Adding a new state had the advantage of reducing implementation complexity and easing programming model issues. SIMD-FP and MMX or x87 instructions can be used concurrently. This clearly eased OS Vendor and ISV concerns. The disadvantage of the second approach was that Intel had a dependency of not being able to use new features without OS support. However, Intel worked around this by implementing the new state store and restore instructions in an earlier implementation. Thus by the time the Pentium III processor was released, the new OS's supported this new state.

To ensure no unusual corner cases, all of the new state was separated from the x87-FP state. Figure 1 shows the new 128-bit registers. There is a new control/status register MXCSR which is used to mask/unmask numerical exception handling, to set rounding modes, to set flush-to-zero mode, and to view status flags.

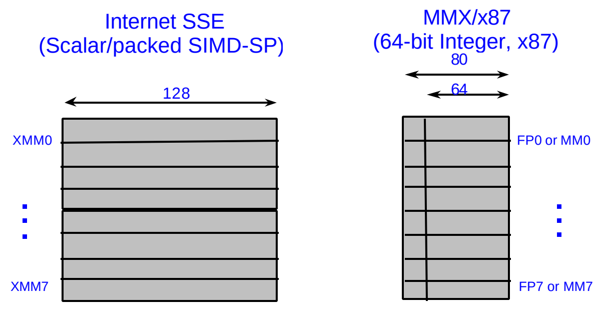


Figure 1: The Internet SSE 128-bit registers

There is also a new interrupt vector to handle SIMD-FP numeric exceptions.

Pentium® III Processor Implementation

The Pentium III processor implements each 4-wide computational macro-instruction as two 64-bit micro-instructions. However, since the processor is a superscalar implementation (i.e., two execution ports), a full 4-wide SIMD operation can also be done every clock cycle (assuming instructions alternate between execution units). With this approach, applications can theoretically achieve a full 4x performance gain; 2x is the realized gain on real applications in part because of micro-instruction pressure within the microarchitecture. A future 128-bit implementation can deliver a higher level of performance scaling.

Scalar Versus Packed Operations

We considered the inclusion of scalar floating instructions in the new SIMD-FP mode because applications often require both scalar and packed operations. It is possible to use x87-FP for scalar and the new registers just for SIMD-FP. However, this approach results in a cumbersome programming paradigm, since x87-FP is a stack register model while the SIMD-FP is a flat register model. Passing parameters would either require more conversion instructions or would be through memory, as currently implemented. Additionally, the results generated via x87-FP operations might differ from SIMD-FP results, due to differences between how computation is performed in the two paradigms (32 bit in SIMD-FP versus 80 bit in x87 FP).

Scalar implementation on the Pentium III processor was problematic because of its emulation of 4-wide SIMD-FP. Using packed instructions for scalar operations would impact performance since both 64-bit micro-instructions would still be executed. Also, it is particularly costly in terms of execution time for long latency de-pipelined packed operations, such as divide and square-root. Lastly, software would need to ensure that no faults occur in the unused slots. To address this issue, explicit scalar instructions were defined, which for the Pentium III processor execute only a single micro-instruction. The upper three components of the source register are passed directly to the destination register when a scalar operation is done; computation is performed only on the lower component pair (Figure 4). Thus, the Pentium III processor did not have to do any operation on the upper half of the data type.

While masked (selective) operations on SIMD-FP registers were another option, we decided against this on the grounds of design complexity and lack of compelling application requirements.

Improving Concurrency

High SIMD performance can only be achieved by balancing memory bandwidth and execution. Multimedia workloads such as 3D graphics and video are streaming applications that have situations where data are largely read once and then discarded. The caches local processors are not large enough to contain the entire data sets of these applications, which results in the execution units being stalled while data are fetched from memory. The out of order and speculative pipeline cannot hide the latency of these accesses without significantly increasing the hardware resources, which impacts die size and cost. A good alternative is to let the programmer overlap execution of one piece of data with the fetch of the next piece so that the latency of the fetch is hidden by the execution time. This works best if the algorithms have a compute-intensive component, such as 3D graphics, where scenes have multiple light sources. Thus we created cacheability hints that allow a programmer to prefetch the next data closer to the processor without touching the architectural state.

For these applications, programmers are the best judges of which data are going to be streaming and which are going to be reused. For example, in 3D graphics, the programmer wants code and transformation matrices to remain in the cache while the input display list and the output command list need to be streamed. This requires some primitives that allow a programmer to manage caching of the data and minimize cache pollution. Thus, the prefetching hints were expanded to let the programmer specify the cache hierarchy level where the prefetched data are going to be placed. Complementary instructions were added to perform non-allocating (streaming) stores so that needed data in the cache does not get replaced, and these stores do not generate unnecessary write-allocation.

The prefetch instructions do not update any architectural state. To some degree, the implementation is specific to each processor. The programmer may have to tune his/her application for each implementation to take full advantage of these instructions. However, it is a design goal to ensure that there are no performance glass jaws between implementations. These instructions merely provide a hint to the hardware: they do not generate exceptions or faults.

Figure 2 illustrates how the various features of the new extensions work together to improve concurrency and reduce total execution time. Prior to Internet Streaming SIMD Extensions, read miss latency and execution and subsequent store miss latency comprised total execution in a serial fashion. The extensions let read miss latency overlap execution via the use of prefetching, and they allowed store miss latency to be reduced and overlap execution via streaming stores. Moreover, SIMD-FP reduces the amount of time spent on execution.

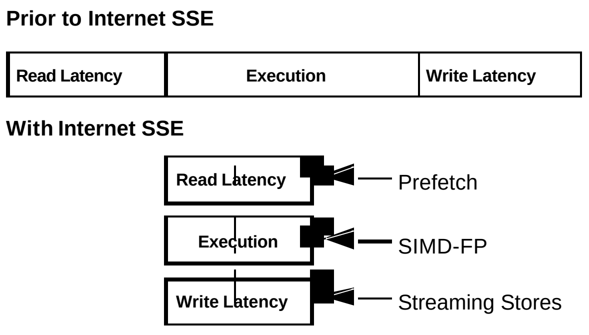


Figure 2: Increasing performance via concurrency

Data Alignment

Hardware support to efficiently handle memory accesses that are not aligned to a 16-byte (128-bit) boundary is expensive in both area and timing. Two options were explored: either detect and fix these cases using a micro-code assist, or generate a general protection fault. ISV feedback was unanimous in their desire to avoid the first option, which can silently introduce a degradation in performance that is difficult to track down. Instead, the ISVs preferred being alerted to misalignment via an explicit fault. As a result, all computation instructions that use a memory operand must be 16-byte aligned. Unaligned load and store instructions are also provided for cases where alignment cannot be guaranteed (i.e., video). These instructions are intended to operate as efficiently or more efficiently than would be the case if explicit code sequences were used to achieve alignment.

Flush-To-Zero and IEEE Modes

We decided to offer two modes of FP arithmetic: IEEE compliance for applications that need exact single-precision computation and a Flush-To-Zero (FTZ) mode for real-time applications. Full IEEE support ensures greater future applicability of the extensions for applications that require full precision and portability, while FTZ mode along with fast hardware support for masked exceptions enables high-performance execution. FTZ mode returns a zero result in an underflow situation during computation if the exceptions are masked. Most real-time 3D applications would use the FTZ mode since they are not sensitive to a slight loss in precision, especially if they can get faster execution by using the FTZ mode.

INSTRUCTION SET ARCHITECTURE

The Internet SSE supplies a rich set of instructions (shown in Table 1) that operate on either all, or the least significant pairs, of packed data operands in parallel. The packed instructions (with PS suffix) operate on a pair of operands as shown in Figure 3 while scalar instructions (with SS suffix) always operate on the least significant pair of the two operands as shown in Figure 4.

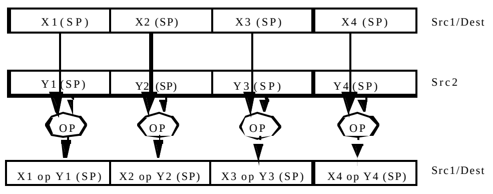


Figure 3: Packed operation

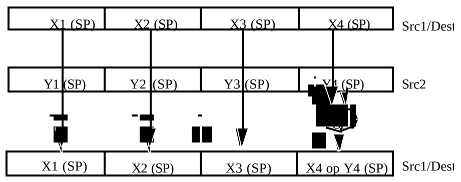


Figure 4: Scalar operation

		Packed Single	Scalar Single	Packed Integer
Arithmetic	ADD, SUB, MUL, DIV, MAX, MIN, RCP, RSQRT, SQRT	X	X	
Logical	AND, ANDN, OR, XOR	X		
Comparison	CMP, MAX, MIN	X	X	
	COMI, UCOMI		X	
Data Movement	MOV (load/store aligned),	X		
	MOVUPS (load/store unaligned), MOVLPS, MOVLHPS, MOVHPS, MOVHLPS (load/store), MOVMSKPS	X		
	MOVSS (load/store)		X	
Shuffle	SHUFPS, UNPCKHPS, UNPCKLPS	X		
Conversions	CVTSS2SI, CVTTSS2SI, CVTSI2SS		X	
	CVTPI2PS, CVTPS2PI, CVTTPS2PI	X		
State	FXSAVE, FXRSTOR, STMXCSR, LDMXCSR	X		
MMX™ Tech Enhancements	PINSRW, PEXTRW, PMULHU, PSHUFW, PMOVMSKB, PSAD, PAVG, PMIN, PMAX			X
Streaming/ Prefetching	MASKMOVQ, MOVNTQ (aligned store)			X
	MOVTPS (aligned store)	X		
	PREFETCH			
	SFENCE			

Table 1: Internet SSE ISA

Basic Building Blocks

These include instructions such as load, store, addition, multiplication, subtraction, division, and square root as well as instructions to access the new Control/Status Register and Save/Restore new state.

Cacheability Hints

As mentioned earlier, achieving high performance for multimedia applications requires some degree of overlap between the execution of a block of data and the fetch of the next block of data. The PREFETCH instruction was added to the new extensions to let the programmer control overlap in the application. This instruction also allows control over data placement in the cache hierarchy and further allows programmers to distinguish between the locality of temporal (i.e., frequently used) cached data and non-temporal (i.e., read and used once before being discarded) data. There are four possible prefetches currently defined with room for future definitions. Note that these instructions can also be used for non-SIMD applications.

Streaming store instructions, MOVNTPS (Packed Single Precision FP) and MOVNTQ (Packed Integer) allow the programmer to specify a write-combining (WC) memory type on a per instruction basis. This is true even for memory otherwise assigned a writeback (WB) memory type via the Memory Type Range Register's (MTRRs) or Page Attribute Table (PAT). This allows the user to obtain all the benefits of a WC memory type (i.e., write-combining, write-collapsing, uncacheable, non-write-allocating) on a per-instruction granularity.

Fencing

In order to allow efficient software-controlled coherency, a light-weight fence (SFENCE) instruction was also included in the new extension; this instruction ensures that all stores that precede the fence are observed on the front-side bus before any subsequent stores are completed. SFENCE is targeted for uses such as writing commands from the processor to the graphics accelerator or to ensure observability between a producer and consumer where communication of data uses stores with a WC memory-type semantic.

Comparison and Conditional Flow

The basic single precision FP comparison instruction (CMP) is similar to existing MMX instruction variants (PCMPEQ, PCMPGT) in that it produces a redundant mask per float of all 1's or all 0's, depending upon the result of the comparison. This approach allows the mask to be used with subsequent logic operations (AND, ANDN, OR, XOR) in order to perform conditional moves. Additionally, four mask (most significant of each component) bits can be moved to an integer register using the MOVMSKPS/PMOVMSKB instructions. These instructions simplify data-dependent branching, such as the clip extent and front/back-face culling checks in 3D geometry, and they address a common desire registered by many ISVs.

Another important conditional usage model involves finding the maximum or minimum between two values (packed or scalar). While this can be done as just described for conditional moves, the MAX/MIN and PMIN/PMAX instructions perform this function in only one instruction by directly using the carry-out from the comparison subtraction to select which source to forward as a result. Within 3D geometry and rasterization, color clamping is an example that benefits from the use of MINPS/PMIN. Also, a fundamental component in many speech recognition engines is the evaluation of a Hidden-Markov Model (HMM); this function comprises upwards of 80% of execution time. The PMIN instruction improves this kernel performance by 33%, giving a 19% application gain.

To provide a complete set of comparisons for CMP, an 8-bit immediate is used to encode eight basic comparison predicates, EQ, LT, LE, UNORD, NEQ, NLT, and NLE. Another four can be obtained by using these predicates and swapping source operands. Using an immediate to encode the predicate greatly reduces the number of opcodes that are assigned to these comparison operations.

Data Manipulation

SIMD computation gains are only realized if data can be efficiently reorganized into an SIMD format. For example, 3D geometry transformation with 4-wide SIMD-FP format can be done per vertex or on four separate vertex components, where a vertex has four components (x, y, z, and w). The method of organizing 3D data structures on a per vertex basis is called Array-of-Structures (AOS) since the display list is an array of individual vertices. Organizing the display list for an ideal SIMD format is called Structure-of-Arrays (SOA) since the structure contains separate x, y, z, and w arrays. An AOS approach is less efficient for two reasons: 1) not all SIMD computation slots may be utilized (i.e., the w vertex component may not be needed); 2) horizontal reduction operations (i.e., dot products such as a * x + b * y + c * z) are typically needed, which use multiple SIMD slots to generate only one unique scalar result. This is exacerbated if other long-latency operations (i.e., square-root and division) are used in conjunction with the horizontal reduction.

Often, it may not be possible to statically reorganize data if for example, in 3D geometry, either a standard API or the rasterization graphics controller do not directly support SOA. In order to efficiently transpose data into the ideal SOA format or vice versa, the new extension supports a number of data manipulation instructions, including the following:

- UNPCKHPS/UNPCKLPS. These interleave floats from the high/low part of a register or memory operand, similar to the MMX unpack instructions.
- SHUFPS/PSHUFW. These support swizzling of data from source operands, including such operations as broadcast, rotate, swap, and reverse.
- MOVHPS/MOVLPS. When used in conjunction with SHUFPS, these 64-bit load/store instructions enable efficient gathering of four individual vertex components from four non-adjacent AOS data structures into a single 128-bit register (SOA); these instructions can be similarly used to de-swizzle SOA to AOS.
- PINSRW/PEXTRW. These support scatter and gather operations on words within an MMX register from memory or the 32-bit integer registers. Examples include gathering texture components and supporting SIMD lookup tables. The PINSRW instruction also gives a performance gain of 22% for the Hidden-Markov Model (HMM) based speech

recognition kernel and a 13% gain at the application level.

A number of experiments were run using various 3D transform/lighting building blocks as well as a more complete geometry pipeline. Of the approaches described for utilizing SIMD computation, the static SOA case achieves the best performance. Computing directly in AOS format achieves only half of the static SOA throughput for a geometry benchmark that implements a full lighting case (ambient, diffuse, specular) due to the reasons listed above. Dynamic reorganization from AOS to SOA and vice versa using a combination of SHUFPS/MOVHPS/MOVLSP instructions incurs a 20%-25% overhead compared to static SOA, which is a 6%-10% better performance than is possible with other methods. Note this overhead is constant and diminishes as more SIMD computation is performed (e.g., with additional lights). Computing directly in AOS may appear to provide a simpler programming model since most APIs handle display lists on a per vertex basis. In order to improve performance for the horizontal operations that can result from computing in an AOS format, several additional instructions, MOVLHPS/MOVHLPS, were added to the extensions. These instructions support emulating a full range of horizontal operations, including addition, subtraction, and logic operations. However, better performance can generally be achieved by computing in an SOA form, and the transpose code used with dynamic reorganization can be effectively hidden behind compiler pragmas or intrinsics.

Conversions

A large number of conversion operations are possible, including integer to/from FP, scalar and packed, source and destination of either register or memory, rounding mode contained implicitly within the instruction, and integer operand sizes (byte, word, double-word). A full set of all permutations is impractical and unnecessary since not all possible cases are common, and many others can be emulated by a sequence of instructions. The factors that motivated the final definition include the following:

- Basic operations between integer and FP are required with both SMID-FP and MMX^{TM} technology for packed data (CVTPI2PS, CVTPS2PI, CVTTPS2PI) and Scalar-FP and IA-32 Integer for scalar data (CVTSS2SI, CVTTSS2SI, CVTSI2SS).
- Destination is a register, since, if needed, the result can be explicitly moved to memory using a store.
- CVTTPS2PI/CVTTSS2SI implicitly encode truncation rounding to eliminate the common serialization performance penalty of changing the control register rounding mode when converting FP to integer.
- Internet SSEs support only conversions to/from double-words. Existing MMX pack and unpack instructions can be used to efficiently translate from double-words to/from words and bytes.

Reciprocal and Reciprocal Square Root

A basic building block operation in geometry involves computing divisions and square roots. For instance, transformation often involves dividing each x, y, z coordinate by the W perspective coordinate. Similarly, specular lighting contains a power function, which is often emulated using an approximation function that requires a division. Also, normalization is another common geometry operation, which requires the computation of 1/square-root. In order to optimize these cases, the new extensions introduce two approximation instructions: RCP and RSQRT. These instructions are implemented via hardware lookup tables and are inherently less precise (12 bits of mantissa) than the full IEEE-compliant DIV and SQRT (24 bits of mantissa). However, these instructions have the advantage of being much faster than the full precision versions. When greater precision is needed, the approximation instructions can be used with a single Newton-Raphson (N-R) iteration to achieve almost the same precision as the IEEE instructions (~22 bits of mantissa). This N-R iteration for the reciprocal operation involves two multiplies and a subtraction, so the overall latency and especially the throughput are lower than the IEEE instructions. For a basic geometry pipeline, these instructions can improve overall performance on the order of 15%.

Unsigned Multiply, Byte Mask Write

Discussions with the D3D team, among others, identified the lack of an unsigned MMX multiply operation as the reason for inefficiency in 3D rasterization performance. This function inherently works with unsigned pixel data, and the existing PMULHW instruction operates only on signed data. Providing an unsigned PMULHUW eliminates fix-up overhead required in using the signed version, creating an application-level performance gain of 8%-10%.

The byte-masked write instruction, MASKMOVQ, is aimed at specific rasterization and image processing applications. The instruction supports several beneficial features:

- A byte mask, either generated by a PCMPEQ/ PCMPGT instruction or loaded from memory, is used to selectively write bytes in the other source operand directly to memory. This mask is effectively transferred in a parallel fashion along with the data throughout the memory subsystem (i.e., write-combining buffers, bus queue entries, and bus byte enables). Alternative implementations with conditional moves or branches did not deliver as much of a performance gain because they introduce significantly more micro-operations into the execution pipeline as well as possible miss-predictions for the branch approach.
- Similar to other non-temporal streaming store cacheability instructions, MASKMOVQ implements a WC memory semantic, which eliminates unnecessary read-for-ownership bandwidth and disturbance of temporal cached data, since the cache is bypassed altogether.

Packed Average

Motion compensation is a key component of the MPEG-2 decode pipeline. It is the process of reconstituting each frame of the output picture stream by interpolating between key frames. This interpolation primarily consists of averaging operations between pixels from different macroblocks where a macroblock is a 16x16 pixel unit. The MPEG-2 specification requires that the result of the averaging operation be rounded to the nearest integer; values precisely at half way should be rounded away from zero. This is equivalent to operations with 9-bit precision. MMX instructions provide either 8 or 16 bits of accuracy, and it is desirable to use the 8-bit versions to increase the data parallelism. The PAVG instruction facilitates the use of 8-bit instructions by performing a 9-bit accurate averaging operation. The word version PAVGW provides higher accuracy for applications that accumulate a result using several computation instructions.

Currently, Motion Compensation of a DVD player on a Pentium® II processor-based system (266MHz) is evenly balanced between memory latency and execution. The PAVG instruction enabled a 25% kernel speedup. Instrumenting the motion compensation code in the player with the PAVG instruction provided a 4%-6% speedup at the application level (depending on the video clip chosen). The application level gain can increase to 10% for higher resolution HDTV digital television formats.

Packed Sum of Absolute Differences

Although the video encode pipeline is quite complex and involves many stages, the bulk of the execution is spent in the motion-estimation function (40%-70% at present). This stage compares a sub-block of the current frame with those in the same neighborhood of the previous and next frames in order to find the best match. Consequently, only a vector representing the position of this match, and the residual difference sub-block, needs to be included in the compressed output stream as opposed to the entire original sub-block.

There are two common comparison metrics that are used in motion-estimation: sum-of-square-differences (SSD) and sum-of-absolute-differences (SAD). Both have benefits and limitations in specific cases, although overall they are roughly comparable metrics in determining the quality of a match.

There is a factoring technique that allows SSD to be implemented using an unsigned multiply-accumulate (byte to word) operation; however, the accumulation range requires 24 bits of precision, which does not map neatly to a general purpose data-type. Instead, the PSADBW instruction retains byte-level parallelism of execution, working on 8 bytes at a time, and the accumulation does not exceed a 16-bit word. This single instruction replaces on the order of seven MMX instructions in the motion-estimation inner loop, largely because MMX technology does not support unsigned byte operations, which need to be emulated by zero extension to words and the use of word operations. Consequently, PSADBW has been found to increase motion-estimation performance by a factor of two.

CONCLUSION

The Internet Streaming SIMD Extensions enable an exciting new level of visual computing on the volume PC platform. The single precision SIMD-FP ISA will deliver the desired performance goal of 2x an increase in FP performance with the Pentium® III processor. This speedup will significantly improve the image quality for real-time 3D applications, and the Pentium III processor systems will enable real-time rendering of complex worlds. This instruction set represents a significant step forward for Intel in improving the performance of visualization on PC platforms.

The addition of SIMD-integer instructions for video will enable real-time video encoding in the MPEG-1 format, as well as the MPEG-2 format, with some trade-offs in visual quality and compression rates. The new extensions will also deliver DVD decode at 30 frames per second within the Pentium III processor timeframe, with good headroom for multitasking other processes. Increasing Pentium III processor frequency will subsequently enable 1M-pixel HDTV format decode. Initially this will require hardware motion compensation support, but with an incremental increase in processor frequency, this decode can be handled entirely in software. These instructions will also enable home video editing similar to that currently available for photographic editing.

A reduction in speech recognition error rates and an increase in dictionary size can be achieved with the use of the prefetching options and the new packed integer instructions. Concurrency of memory accesses and execution have also been enhanced across the range of multimedia applications via the new cacheability instructions.

The definition team delivered the new ISA in record time, working diligently to review all the requested instructions and analyzing the potential improvement in application performance. Intense scrutiny was applied to the definition by the three implementation teams to justify inclusion of instructions. A range of constraints drove the final definition, including performance benefits, die size, timing impact, and code portability. The Internet SSE implementation cost in the Pentium III processor was just around 10% of the die size. This is similar to the cost of including MMX™ technology in the Pentium® and Pentium® II processors.

ACKNOWLEDGMENTS

We acknowledge the contribution of the entire definition team, in particular Srinivas Chennupaty, Patrice Roussel, Vladimir Pentkovski, and Mohammad Abdallah. We also acknowledge the adoption of IA-64 definition and semantics for a number of instructions to maintain consistency. We also acknowledge the guidance of Glenn Hinton, Bob Colwell, and Fred Pollack during the definition period.

REFERENCES

[1] Alex Peleg, et al, "MMX™ Technology Extension to the Intel® Architecture" DTTC Proceedings 1996 (Internal Intel Document).

[2] Millind Mittal, et al, "MMX™ Technology Architecture Overview," Intel Technology Journal, Q3, 1997, http://developer.intel.com/technology/itj/q31997.htm.

AUTHORS' BIOGRAPHIES

Shreekant (Ticky) Thakkar is a principal processor architect in the Microprocessor Products Group. He led the Internet Streaming SIMD Extension development for the Pentium® III processor. Prior to that, Shreekant was responsible for the development of the Pentium® Pro multi-processor. His e-mail is ticky.thakkar@intel.com.

Tom Huff is an architect in the Microprocessor Product Group in Oregon. He was one of the architects in the core team that defined the Internet Streaming SIMD Extensions for the IA-32 architecture. He is currently working on multimedia performance analysis for the Willamette processor project. He holds M.S. and Ph.D. degrees in electrical engineering from the University of Michigan. His email is tom.huff@intel.com.

Pentium® III Processor Implementation Tradeoffs

Jagannath Keshava and Vladimir Pentkovski: Microprocessor Products Group, Intel Corp.

ABSTRACT

This paper discusses the implementation tradeoffs of the Pentium® III processor. The Pentium III processor implements a new extension of the IA-32 instruction set called the Internet Streaming Single-Instruction, Multiple-Data (SIMD) Extensions (Internet SSE). The processor is based on the Pentium® Pro processor microarchitecture.

The initial development goals for the Pentium III processor were to balance performance, cost, and frequency. Descriptions of some of the key aspects of the SIMD Floating Point (FP) architecture and of the memory streaming architecture are given. The utilization and effectiveness of these architectures in decoupling memory accesses from computation, in the context of balancing the 3D pipe, are discussed. Implementation choices in some of the key areas are described. We also give some details of the implementation of Internet SSE execution units, including the development of new FP units, and discuss how we have implemented the 128-bit instructions on the existing 64-bit datapath. We then discuss the details of the memory streaming implementation.

The Pentium III processor is now in production on frequencies of up to 550 MHz. The new instructions in the Internet SSE were added at about a 10% die cost and have enabled the Pentium III processor to offer a richer, more compelling visual experience.

INTRODUCTION

The goal of the Internet SSE development was to enable a better visual experience and to enable new applications such as real-time video encoding and speech recognition [7]. The Pentium® III processor is the first implementation of ISSE. It is based on the P6 microarchitecture, which allows an efficient implementation in terms of die size and effort. The key development goals were the implementation of the Internet SSE while keeping about a 10% larger die size than the Pentium® II processor and achieving a higher frequency by at least one bin.

Two features of these new applications challenge designers of computer systems. First, the algorithms that the applications are based on are inherently parallel in the sense that the same sequence of operations can be applied concurrently to multiple data elements. The Internet SSE allows us to express this parallelism explicitly, but the hardware needs to be able to translate the parallelism into higher performance. The P6 superscalar out-of-order microarchitecture is capable of utilizing explicit as well as extracted implicit parallelism. However, hardware that supports higher computation throughput improves the performance of these algorithms. The development of such hardware and increasing its utilization were key tasks in the development of the Pentium III processor. Second, in order to feed the parallel computations with data, the system needs to supply high memory bandwidth and hide memory latency.

The implementation section of this paper contains details of some of the techniques we used to provide enhanced throughput of computations and memory while meeting aggressive die-size and frequency goals. The primary purpose of this paper, however, is to describe key implementation techniques used in the processor and the rationale for their development.

ARCHITECTURE

The Pentium® III processor is the first implementation of the Internet SSE. The Internet SSE contains 70 new instructions and a new architectural state. It is the second significant extension of the instruction set since the 80386 and the first to add a new architectural state. MMX™ was the first significant instruction extension, but it did not add any new architectural state. The new instructions fall into different categories:

- SIMD FP instructions that operate on four single precision numbers
- scalar FP instructions
- cacheability instructions including prefetches into different levels of the cache hierarchy
- control instructions
- data conversion instructions

- new media extensions that are instructions such as the PSAD and the PAVG that accelerate encoding and decoding respectively

Adding the new state reduced implementation complexity, eased programming model issues, and allowed SIMD-FP and MMX technology or X87 instructions to be used concurrently. It also addressed ISV and OSV requests. All of the SSE State was separated from the X87-FP State; there is a dedicated interrupt vector to handle the numeric exceptions. There is also a new control/status register, MXCSR, which is used to mask/unmask numerical exception handling, to set rounding mode, to set flush to zero mode, and to view status flags. Applications often require both scalar and packed mode of operations. To address this issue, explicit scalar instructions (in the new SIMD-FP mode) were defined, which for the Pentium III processor execute only a single micro-instruction. Support is provided for two modes of FP arithmetic: IEEE compliant mode for applications that need exact single precision computation and portability and a Flush-to-Zero (FTZ) mode for high-performance real-time applications.

(Details of the instruction set are given in other papers in this issue of the *Intel Technology Journal*.)

IMPLEMENTATION

In this section we discuss some of the key features that we developed to increase FP and memory throughput on the Pentium® III processor. We then discuss a couple of techniques we developed to help provide an area-efficient solution. Figure 1 shows the block diagram of the P6 pipeline.

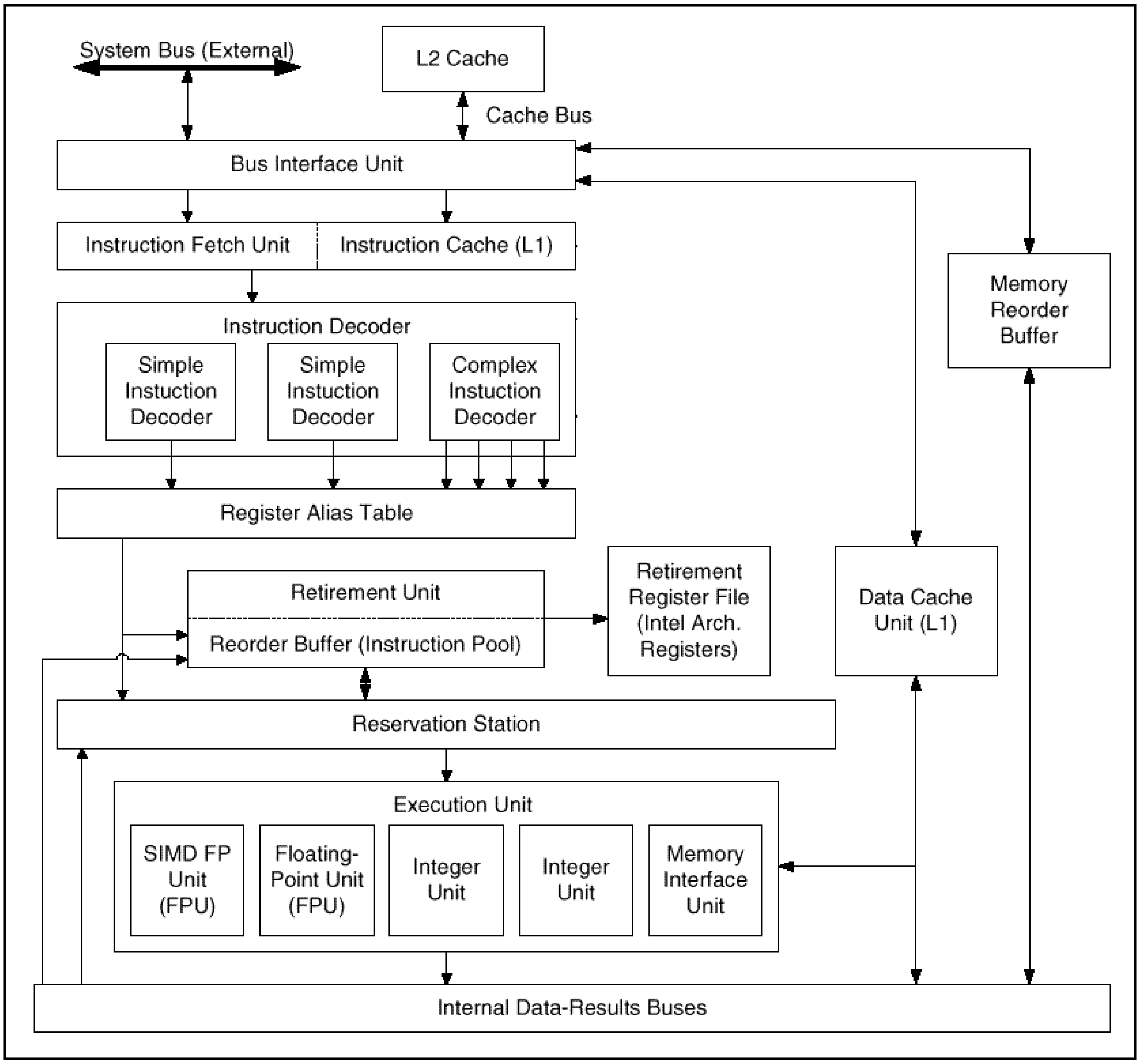


Figure 1. Functional Block Diagram of the P6 Family Processor Microarchitecture

Figure 2 shows the Dispatch/Execute units in the Pentium® II processor. An overview of the P6 architecture and the microarchitecture is given in [5] and [6] where you will also find a description of the blocks shown in Figures 1 and 2.

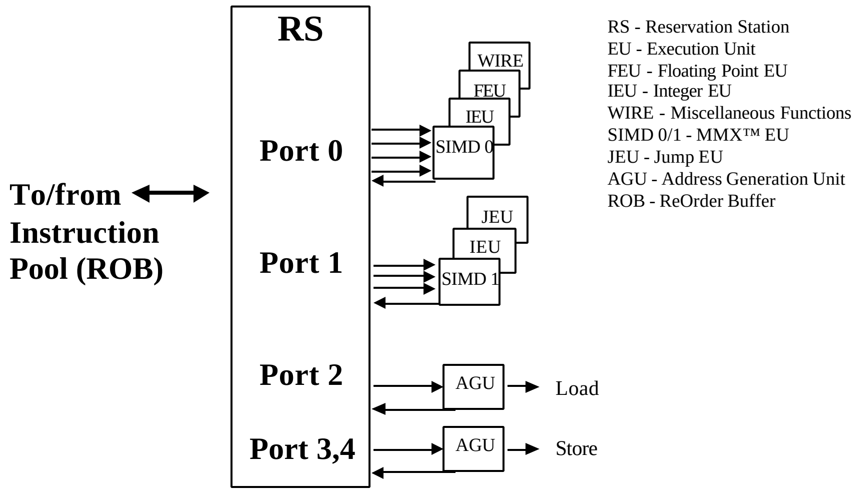


Figure 2: Pentium II Dispatch/Execute units

Implementation of Internet SSE Execution Units

The Internet SSE was implemented in the following way. The instruction decoder translates 4-wide (128-bit) Internet SSE instructions into a pair of 2-wide (64-bit) internal uops. The execution of the 2-wide uops is supported by 2-wide execution units. Some of the FP execution units were developed by extending the functionality of existing P6 FP units. The 2-wide units boost the performance to that of twice the Pentium II processor. Further, implementing the 128-bit instruction set on the 64-bit datapath limits the changes to the decoder and the utilization of existing and new execution units. We also implemented a few other features to improve the utilization of the FP hardware:

1. The adder and multiplier were placed on different ports. This allows for simultaneous dispatch of 2-wide addition and 2-wide multiplication operations. This boosts the peak performance two more times when compared to the Pentium II, and hence, it allows 2.2 GFLOP/sec peak at 550 MHz. The new units developed on the Pentium III and modified P6 units are shown in color in Figure 3.

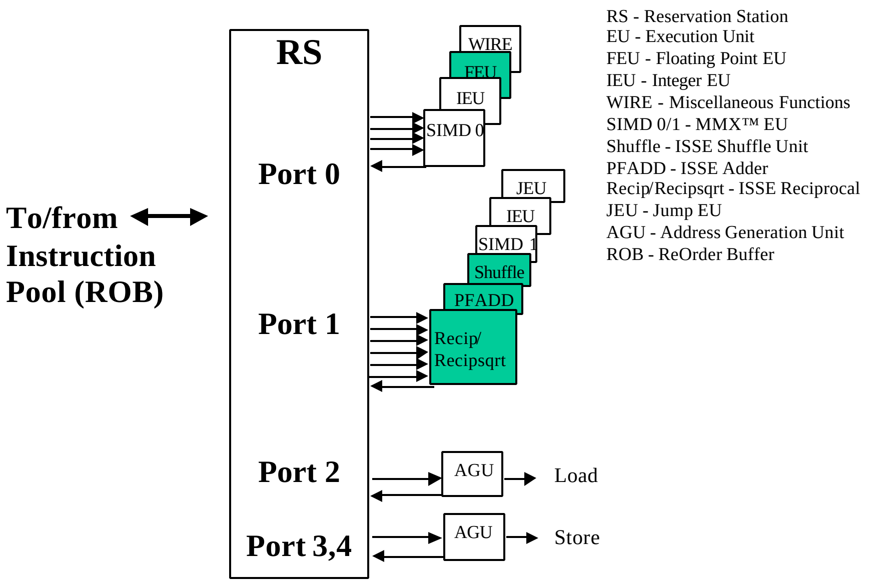


Figure 3: Pentium III Dispatch/Execute units

All the new units have been added on Port 1. The new operations executed on Port 0 have been accomplished by modifying existing units. The multiplier resides on Port 0 and is a modification of the existing FP multiplier. The new packed SP multiplication is performed with a throughput of two SP numbers each cycle with a latency of four cycles. (The X87 SP processor, on the other hand, had a throughput of a single SP number every two cycles and a latency of five cycles.) A new packed SP adder is provided on Port 1. The adder operates on two SP numbers with a throughput of one cycle and a latency of three cycles. The adder also executes all compare, subtract, min/max, and convert instructions. Essentially, we have assigned different units (and therefore different instructions) to Ports 0 and 1 to ensure that a full 4-wide peak execution can be achieved.

2. Hardware support is in place for data reorganization. Effective SIMD computations require adequate SIMD organization of data. For instance, the conventional representation of 3D data has the format of “(x, y, z, w)”, where x, y, and z are the three coordinates of a vertex, and w is the perspective correction factor. In some cases, SIMD computations are more effective if the data are represented as vectors of equivalently named coordinates “(x1, x2, …), (y1, y2,…), (z1, z2,…), (w1, w2,…”). In order to support transformations between these type of data representations, the Internet SSE includes the set of data manipulation instructions. We considered the effective hardware support of these instructions to be an important method to improve the utilization of FP units, since it allows for less time to be spent in data reorganization. The new shuffle/logical unit serves this purpose. It shares Port 1 and executes the unpack high and unpack low, move, and logical uops. The 128-bit shuffle operation is performed through three uops: (1) copy temporary, (2) shuffle low, and (3) shuffle high. The shuffle unit also executes packed integer shuffle, PINSRW and PEXTRW, through sharing of the FP shuffle unit.

3. Data is copied faster. IA-32 instructions overwrite one of the operands of the instruction. We knew that this feature would add more MOVE instructions to the code. For instance, consider the following fragment:

Load memory operand A to register XMM0

Multiply XMM0 by memory operand B

The second instruction overwrites the content of the register XMM0. Hence, if the subsequent code uses the same memory operand A, then we need to either load it again from the memory (thus putting additional pressure on the load port, which is frequently used in multimedia algorithms), or we need to copy XMM0 to another register for later re-use. In order to facilitate the latter method, we implemented two move ports in the Pentium III processor.

Exceptions Handling

The implementation of a 128-bit processor via two 64-bit micro-ops raises the possibility of an exception occurring on either of the two independent micro-ops (uops). For instance, if the first uop retires while the second uop causes an exception, the architecturally visible 128-bit register is updated only partially, and it would cause an inconsistency in the architectural state of the machine. Retirement is the act of committing the results of an operation to architectural state. In order to provide "precise exceptions handling" we implemented the Check Next Micro-Operation (CNU) mechanism that prevents the retirement of the first uop if the second one causes an exception. The mechanism acts as follows. The first uop in a pair of two uops, which need to be treated as an atomic operation and/or data type, is marked with the CNU flow marker. The instruction decoder decodes the CNU marker and sends the CNU bit to the allocator. The allocator informs the ROB to set the CNU bit in the ROB entry allocated for this uop. The ROB is the reorder buffer and stores the micro-ops in the original program order. The ROB will delay retirement of the first uop until the second uop is also guaranteed to retire. Since this mechanism throttles the retirement, we implemented the following optimization. In the case where all exceptions are masked, each uop may be retired individually. Since multimedia software usually masks the exceptions, for all practical purposes, there is no loss of computational throughput.

Moreover, to maintain high computational throughput, we implemented in hardware the fast processing of masked exceptions, which happen routinely during execution of multimedia algorithms, such as overflow, divide by zero, and flush-to-zero underflow. These exceptions are handled by hardware through modifications to the rounder/writeback multiplexers.

CACHEABILITY IMPLEMENTATION

We now discuss the key changes in the memory implementation. These include support for the cacheability control features introduced by the Internet SSE instruction set. Support for byte masked writes, streaming stores, data prefetching, multiple WC buffers, and store fencing operations have been incorporated.

These are some of the aspects of the prefetch implementation on the Pentium III processor. Figure 4 shows the compulsory effect of two stalls that happen in the Pentium II if the load instruction misses the cache.

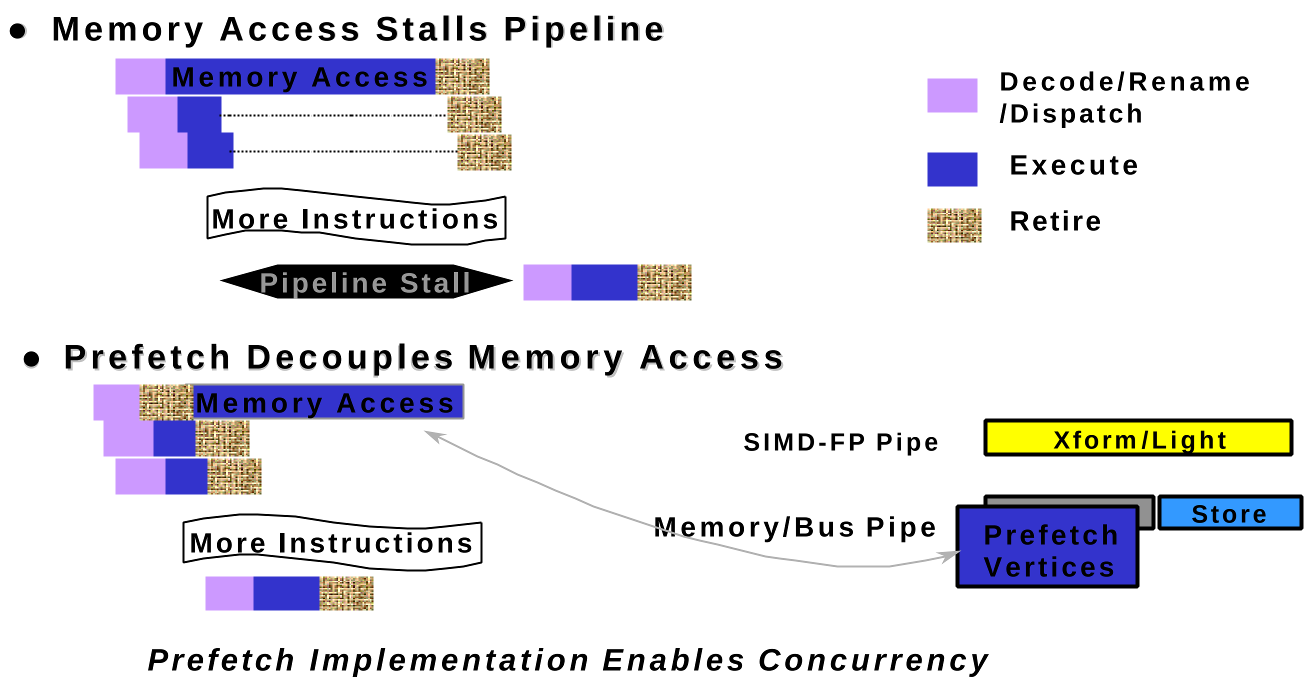


Figure 4: Prefetch implementation

The pipeline stall shown in the "memory access stalls pipeline" portion of Figure 4 is caused by the fact that the load instruction retires much later than the previous instruction. The instructions following the load are executed, but they cannot retire from the machine until the load returns the data. This is illustrated in the "memory access stalls pipeline" portion of the figure: the instructions subsequent to the memory access execute and then wait for the memory access to finish executing before they retire. Therefore, these instructions accumulate inside the machine. Eventually some of the key resources of the processor, such as the ROB that registers non-retired instructions, get saturated. This immediately stalls the front end of the processor since no new instructions can get the resources needed for execution. In the Pentium III processor, we removed this bottleneck for the prefetch instruction. We moved the retirement of the prefetch instruction much earlier in the pipe. This is illustrated in the "prefetch decouples memory access" portion of Figure 4. Here we observe that instructions after the memory access (in this case, Prefetch) are allowed to retire even though the memory access itself has not completed its execution. The prefetch is implemented such that even in the case of a cache miss, it retires almost immediately, and no retirement and resources saturation stalls occur due to memory access. As a result, we get much better concurrency of computations and memory access. This concept is called senior load and is shown in Figure 5.

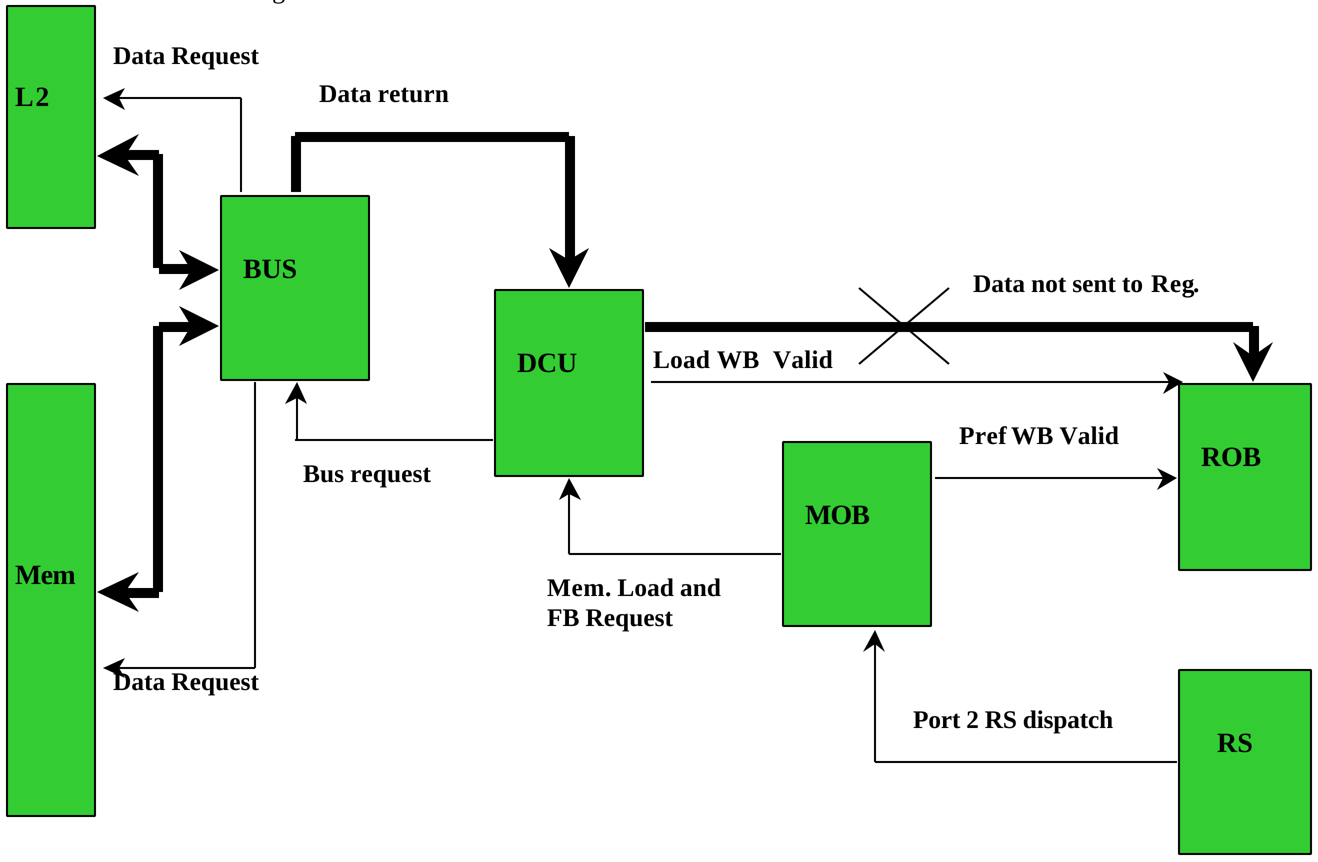


Figure 5: Senior load implementation

Figure 5 shows the differences in how readiness for retirement is signaled in load instructions and prefetch instructions. In the case of a load, the instruction is dispatched into the memory pipe after dispatch from the RS. If it misses the DCU, it is dispatched by the BUS unit. After the data returns from the L2 or the bus, the load is signaled as complete (Load WB valid), and the load and subsequent completed instructions are eligible to retire. In the case of the Prefetch, completion is signaled (by the Pref WB Valid) almost immediately after allocation into the MOB. The completion is not delayed until the data is actually fetched from the memory subsystem. The signaling of early completion permits the retirement of the load, and subsequent instructions occur earlier than in the case of the load, thus removing the resource stalls associated with memory access latency. The prefetch instructions can fetch data into different levels of the cache hierarchy. Also, streaming store instructions are supported. In many instances, applications stream data from memory and use them once without modifications. Using regular cache models will result in eviction of useful data from the cache. In the Discussion section of this paper, you will find details on cache management to increase performance.

We now discuss writes. The main issue with writes is that many applications such as video and 3D have a large working set. This working set doesn't fit into the cache. In such a situation, additional performance may actually be gained if the application bypasses the cache. In other words, the application should keep data in the memory. Hence it should write directly to the memory. This is what streaming stores are for.

The implementation of high write throughput is as important as high bandwidth memory reads. What we have done in the Pentium III processor is mainly two things: we have improved the write bandwidth and the write combining allocation and eviction. The bus write bandwidth was improved by 20%. The Pentium III processor now can saturate a 100 MHz bus with 800 MBs of writes. This was done by removing the dead cycle between back to back write combining writes.

We also improved the write buffers allocation mechanism in order to support this large write bandwidth. Since we re-use Pentium II fill buffers to do this, some further clarifications on the difference in buffers in the Pentium III and the Pentium II processors are in order. The differences are based on the very nature of SSE. Before the Pentium III processor, the architecture was mainly oriented to scalar applications. The purpose of fill buffers was to provide high instantaneous throughput caused by bursts of misses in scalar applications. The average bandwidth requirements were comparatively small, about 100 MB per second, but instantaneous requirements were high. SSE applications are streaming applications such as vector algorithms. Hence, the purpose of SSE buffers is to sustain high average throughput. In terms of overall (read+write) throughput requirements, the capacity of the Pentium II processor's fill buffers is enough for the Pentium III processor timeframe. But the allocation policy had to be improved in order to increase the efficiency with which this capacity is used. We therefore allowed a few write buffers at a time, and we provided fast draining of the buffers to reduce the occupancy time. The faster draining of the buffers and the efficient utilization techniques for multiple buffers are described below.

The Pentium III processor's write combining has been implemented in such a way that its memory cluster allows all four fill buffers to be utilized as write-combining fill buffers at the same time, as opposed to the Pentium II processor which allows just one. To support this enhancement, the following WC eviction conditions, as well as all Pentium™ Pro WC eviction policies, are implemented in the Pentium III processor:

- A buffer is evicted when all bytes are written (all dirty) to the fill buffer. Previously the buffer eviction policy was "resource Demand" drivem, i.e. a buffer gets evicted when DCU requests the allocation of new buffer.
- When all fill buffers are busy a DCU fill buffer allocation request, such as regular loads, stores, or prefetches requiring a fill buffer can evict a WC buffer even if it is not full yet.

Die-Frequency Efficient Implementation

In the Pentium III processor, a number of tradeoffs were made to remain within tight die-size constraints and to reach the frequency goals. Two of these tradeoffs are mentioned below:

- We merged the x87 multiplier with the packed FP multiplier. This helped significantly with die size and kept the loading on the ports down. Loading on the writeback busses was an important factor for us. The writeback busses have been significant speed paths in past implementations, and the addition of new units and logic for implementation of the Internet SSE would have made the situation worse. This was an area of focus from the very inception of the project. We also considered merging the x87 adder with the packed FP adder, but we did not follow through with this because of schedule tradeoffs.
- We reused the multiplier's Wallace tree to do the PSAD. The PSAD, computing the absolute difference of packed MMX values, was implemented with three uops: computation of the difference, computation of the absolute value, and the sum of the absolutes. The sum of the absolutes was computed in the multiplier's Wallace tree. The bytes that needed to be added were fed into the tree that normally sums the multiplication partial products. The reuse of this logic enabled us to implement the instruction with a very small die and frequency impact. Alternatives to execute the instruction with reasonable performance were significantly more expensive.

RESULTS

We would like to outline two main results: the method of implementation of 4-wide ISA via concurrent dispatch of two 2-wide streams of computations, and decoupled execution of the streams of computations and memory.

Implementation of a 4-wide ISA based on a 2-wide data path provides a good tradeoff between die size and performance. From the performance standpoint, this approach may raise the question of how is 2-wide implementation of a 4-wide ISA different from a 2-wide implementation of a 2-wide ISA The difference comes from the fact that 4-wide ISA has twice as many registers. Consider a loop of instructions that uses eight registers. The loop coded in 4-wide ISA can be viewed as the loop coded in 2-wide ISA, which is then twice unrolled and software-pipelined. In general, the explicit loop unrolling improves performance. In particular, it delivers additional improvement even in the out-of-order architecture, since it exposes more parallelism to the machine. The same loop in a 2-wide ISA cannot be unrolled since the loop uses all the available registers. Hence, in the case of a 4-wide ISA, the performance benefits come from two sources: internal out-of-order scheduling plus the explicit loop unrolling. In the case of 2-wide ISA, the benefits come from the internal out-of-order execution only.

The main reason behind the streaming architecture is to meet the requirements of multimedia performance by providing concurrent processing of data streams. From the implementation standpoint, it means that the processor should provide concurrent execution of the computational stream and stream of memory accesses.

The P6 microarchitecture extracts concurrency of memory accesses and computations. However, the explicit prefetch instructions allowed us to completely decouple the data fetch and retirement of subsequent instructions. Hence, the throughput of each of these streams can reach almost the theoretical maximum possible for the given task. As a result, the maximum throughput that can be reached with the Pentium III processor for the given tasks is equal to the lowest of maximum memory throughput and maximum computational throughput of this task.

Since we increased the effective memory throughput, we had to balance the throughput of the processor buffering subsystem and bus throughput. We did not implement new buffers but rather we implemented a few methods to improve the utilization of the existing buffers and improve the write throughput of the external bus. This allowed us to pay a negligible die-size price for performance balancing the memory datapath.

DISCUSSION

In parallel with the development of the Pentium III processor, we developed programming models that allow us to utilize the potential gain of this implementation in real-world applications. In order to outline the details of these models, we discuss three types of multimedia applications:

1. *Compute bound applications such as AC3 Audio*. These applications exhibit fairly small memory bandwidth requirements, but need large computational throughput. In the Pentium III processor these applications are supported by high throughput FP units. In order to utilize the computational power of these units, programmers are supposed to use SSE optimization tools described in [2].

2. *Memory bound applications such as 3D imaging*. The distinct feature of these applications is a fairly large working set. Because of this, the data of these applications usually are in the memory, and the cache doesn't work as well as it does for compute-bound applications. Moreover, in some cases, it is even better to bypass the cache. In these cases, the software can keep data in the memory and utilize the high memory throughput and concurrency described above. In order to utilize these features, it is recommended that a software developer identify incoming and outgoing streams, program these streams using prefetch and streaming store instructions in order to ensure that these streams are fetched/stored directly from/to memory without excessive internal caching. The paper in [3] describes the details of some of these techniques by describing an on-line driver approach for 3D geometry.

   Additional techniques for prefetching include optimizing the length of the data streams to reduce the degree of memory access de-pipelining. This may happen in the beginning of a data stream due to unutilized prefetches. Reference [3] describes a DrawMultiPrimitive technique that demonstrates the details of this programming model.

   The details of these methods are described in [3]. 3D processing is an example. Software implementation of this model allowed us to achieve twice the speedup at the application level.

3. *Mixed class such as video encoding*. These applications usually have few working sets; some of them fit into cache, some of them do not. The strategy of implementation support and programming model for these applications is based on the combination of the above methods. For these types of applications, it is important to separate frequently reused working sets from ones that are used less frequently, and to build a caching strategy based on the frequency of reuse.

   For instance, the MPEG2 Encoder [4] processes the data shown in Figure.6: color and brightness data of Intrinsics frames (I-frame), color and brightness data of Bi-directional frames (B-frames), and downsampled data.

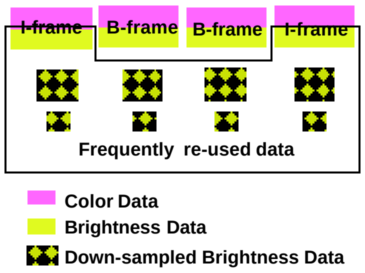


Figure 6: Data reuse in MPEG-2 encoder

According to the encoding algorithm, the I-frame brightness data and downsampled data are processed a few times more frequently than I-frame color data and B-frame data. Hence, the caching strategy for this application is to keep the former data in the cache, and the latter in memory. Figure. 7 shows the difference between two methods of data placement in the cache/memory hierarchy: non-controlled caching in the case of regular IA-32 caching, and software controlled caching that can be achieved in the Pentium III using Internet SSE streaming store and prefetch instructions. Though the I-frame color data and B-frame data are in the memory, the high throughput memory prefetch/store instructions allow us to hide the latency of the data fetch.

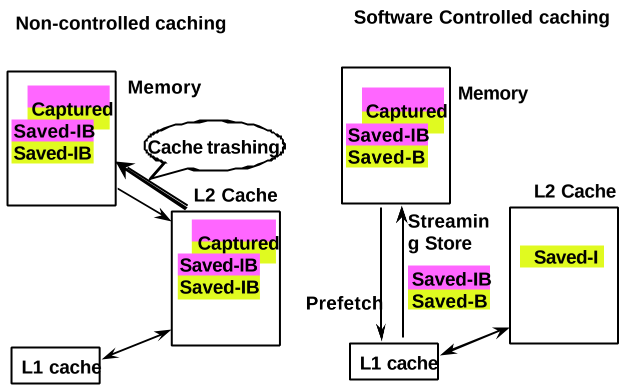


Figure 7: Software controlled caching vs. non-controlled caching in the MPEG-2 encoder

The combination of this model with the application of the PSAD instruction (in the motion estimation algorithm) allowed us to reach MPEG2 real-time software encoding in the Pentium III processor.

CONCLUSION

The Pentium III processor is now in production on frequencies of up to 550 MHz. The 70 new instructions that were added were done at a cost of an additional ~10% in die size. The features that we have described have enabled the Pentium III processor to achieve superior multimedia performance. One more important feature is that our fairly straightforward implementation of the 4-wide Internet SSE and concurrency of the computational and memory streams allows for further performance scalability of SSE applications moving toward higher frequencies.

ACKNOWLEDGMENTS

The development of the Pentium III processor is the result of work done by a number of people in MPG-FM architecture, design, and validation. A lot of the work described in this paper was done by the MPG-FM architecture group. We acknowledge the guidance of Srinivas Raman during the definition and execution of the product. We also thank Dave Papworth, Glenn Hinton, Pete Mcwilliams, Bob Colwell, Ticky Thakkar, and Tom Huff for their input during the process.

REFERENCES

[1] Narayanan Iyer, Mohammad Abdallah, S.Maiyuran, Vivek Garg, S.Spangler, “Katmai Features POR,” Intel internal document.

[2] Joe Wolf, “Programming Methods for the Pentium® III Processor’s Streaming SIMD Extensions Using the Vtune^{TM} Performance Enhancement Environment,” *Intel Technology Journal*, Q2, 1999.

[3] Paul Zagacki, Deep Buch, Emile Hsieh, Hsien-Hsin Lee, Daniel Melaku, Vladimir Pentkovski, “Architecture of a 3D Software Stack for Peak Pentium® III Processor Performance,” *Intel Technology Journal*, Q2, 1999.

[4] James Abel et al, “Applications Tuning for Streaming SIMD Extensions,” *Intel Technology Journal*, Q2, 1999.

[5] David B. Papworth, “Tuning the Pentium Pro Microarchitecture,” IEEE Micro 1996.

[6] Intel, *Pentium Pro Family Developer’s Manual*, Intel literature.

[7] Ticky Thakkar et al, “The Internet Streaming SIMD Extensions,” *Intel Technology Journal*, Q2, 1999.

AUTHORS’ BIOGRAPHIES

Jagannath Keshava has been with Intel since 1990. He is currently working in the MPG-Folsom Architecture Group

on the definition of integrated microprocessors for the Value PC segment. He led the Pentium III processor definition and microarchitecture validation teams. In the past, he has held lead positions in design, microarchitecture, and validation on the Pentium II and i960® microprocessor groups. Jagannath has an M.S. degree in computer engineering from the University of Texas, Austin.

His e-mail is jagannath.keshava@intel.com

Vladimir Pentkovski is a Principal Engineer in the Microprocessor Product Group in Folsom. He was one of the architects in the core team, which defined the Internet Streaming SIMD Extensions of IA-32 architecture. Vladimir led the development of Pentium III processor architecture and performance analysis. Previously he led the development of compilers and software and hardware support for programming languages for Elbrus multi-processor computers in Russia. Vladimir holds a Doctor of Science degree and Ph.D. degree in computer science and engineering from Russia. His e-mail is vladimir.m.pentkovski@intel.com.

Pentium® III Processor Serial Number Feature and Applications

Stephen Fischer, BMD-FM Design, Intel Corp.
James Mi and Albert Teng, Content Group, Intel Corp.

Index words: Pentium® III, Internet, Java*, asset management, information management

ABSTRACT

With the ever-growing importance of the Internet in the everyday life of an individual or a business user of a personal computer, the ability to have some form of unique identifier for that computer has become increasingly important. Applications ranging from system management for reducing Total Cost of Ownership (TCO) and electronic commerce to information management can expect to benefit from such a capability.

In response to this need, the Pentium® III processor has incorporated a serial number capability into the existing instruction set. The serial number feature makes use of information programmed onto the die during manufacturing, designed to create a unique number that is readable by external software.

Intel concerns about user privacy led to the incorporation of a user hardware disable feature for the processor serial number.

INTRODUCTION

The Intel® processor serial number (which for brevity will be referred to as ps#) refers to a new feature introduced with the Pentium® III processor, namely a unique numeric identifier. This serial number can be read by external software.

With the availability of a processor-based serial number, new classes of software applications are more easily enabled. Moreover, electronic transactions via the Internet can be more easily enabled by using the ps# as an added level of support for authentication. Corporations can make use of the ps# to facilitate system configuration and tracking, thereby improving manageability. Sensitive data can be closely controlled by binding the ps# information to the accessibility of the data.

This paper defines the processor serial number feature and explains how it is implemented on the Pentium III processor. We also describe some of the applications that are enabled with this capability, and give an overview of an application framework that provides CPU identification, based on the ps# in an open network environment, and limits cross-correlation of information across Web sites.

ARCHITECTURE AND IMPLEMENTATION

The ps# capability introduced in the Pentium® III processor is communicated through an extension of the existing CPUID instruction [1]. The CPUID instruction is responsible for returning specific parameters of the processor to external software. The type of parameters include items such as the product family, model, and stepping, as well as feature-specific attribute information such as a feature flags field for indicating what functions are available for this processor. Including the processor serial number information in the list of possible parameters that can be returned was therefore a natural extension of the CPUID instruction. Since this instruction can be executed at all privilege levels (PL0 – PL3), it is available for execution at the application level as well as the OS level, an important distinction that enables a much wider range of uses of the ps# feature.

Parameter information such as the stepping number or feature flag bits are returned in the general purpose registers EAX, EBX, ECX, and EDX when the CPUID instruction is executed with a specified input index value held in EAX.

```
Case EAX=0;
{
       EAX           = maximum index supported
       EBX:EDX:ECX   = “GenuineIntel”
}
Case EAX=1;
{
       EAX           = Family:Model:Stepping
       EBX:ECX       = reserved
       EDX           = feature flags (New ps# feature flag bit 18)
}
Case EAX=2;
{
       EAX:EBX:ECX:EDX  = cache and TLB parameters
}
Case EAX=3;
{
       EAX:EBX       = reserved
       ECX:EDX       = processor serial number  data
}
```

Figure 1: CPUID instruction definition

The ps# information is returned by the addition of a new index (EAX=3). The presence of the ps# feature is also indicated by the setting of a new feature flag, bit 18. This allows external software to determine if the ps# feature is supported and enabled in the processor. Figure 1 summarizes the definition of the CPUID instruction supported by the Pentium III processor.

To address potential concerns about compromising user privacy through the visibility of the processor serial number, a capability is incorporated that allows a user to choose whether to enable or disable this feature. This capability is intended to be under the user or system manager’s control. This is implemented through the addition of a new read/write control register disable bit with a “sticky” property. During execution of the CPUID instruction, the internal microcode of the processor polls this bit to determine if ps# information should be reflected back to the CPUID instruction-level functionality. Once the bit is set to a ‘1’ (ps# disable), it cannot be cleared back to a ‘0’ through software means; only a hardware reset of the processor can clear the disable bit, thus preventing subsequent software from overriding the user preference setting after a potential software disable action has been performed. It should also be noted that the ps# disable bit is accessible only at the highest privilege level (PL0). This level of accessibility keeps the user or system manager preference setting decision with supervisory or initialization software such as the system BIOS.

The ps# information returned by the Pentium III processor is derived from on-die polysilicon fuse bits programmed at wafer sort. The underlying microcode supporting the CPUID instructions reads the logical programmed values of these internal fuse bits and concatenates them to form a 64-bit value returned in the general purpose registers EDX and ECX.

The underlying fuse technology is based on a novel silicon approach that uses a Ti-silicide layer on top of a polysilicon line [2]. Programming occurs by a current stress that is high enough to cause agglomeration of the Ti-silicide. A current mirror sensing circuit is used to measure the programmed fuse resistance relative to an unprogrammed reference fuse and return a logical value. The technology has yielded near 100% programming success and maintains this reliability under thermo-mechanical and bias-temperature stress conditions. Redundant fuse elements for each logical fuse bit are incorporated to further increase the reliability for a successful programmed value, yielding a robust process for deriving and programming the serialized value for the ps#, in manufacturing of the Pentium III processor.

APPLICATION USAGE MODELS

Example 1: Improving Manageability, Reducing TCO

In large enterprise environments, IT managers face daily challenges to ensure a well-managed and smoothly running computing infrastructure. The Intel® Pentium® III processor and its ps# give IT departments a new tool to improve manageability and lower the total cost of ownership of PCs.

In the past, IT departments utilized a variety of methods, including user name, MAC address, IP address, and GUID to identify hardware. However, none of these methods are as consistent and reliable as the ps#, which cannot be erased or changed. With a ps#, it is easy to identify a specific PC, even if the system changes users, network cards are swapped out, or the system software and BIOS are reloaded. The ps# also makes it possible to report more reliably on software asset management: IT managers can know with a higher level of certainty which software is running on each system and who the users of the software program are. It can also assist Help Desk personnel in troubleshooting problems even when a PC's hard disk has crashed.

For system configuration and software updates, companies can use the ps# as a way of reliably identifying PCs pre-boot and post installation remotely. If support technicians know the processor serial number ahead of time, they can enter the number in the database and pre-program the software to be delivered when the PC is placed on the network. This reduces on-site engineering visits and automates the configuration process, saving time and reducing support expenses.

When a processor fails in a multiprocessor or clustering environment, it is difficult to determine which specific processor failed. With the Windows NT* operating system, the logical processor can be identified but not mapped to the physical processor. The ps#, however, allows IT staff to determine the exact point of failure, thereby enabling them to route work around the problem processor. This can significantly improve load balancing and fault tolerance, and it can increase the system's availability to the user.

Example 2: Enhancing Management for E-Business

Internet-based Electronic-Business (E-Business) gives companies new freedom to push and pull information to and from one another, but also increases the need to ensure that the information reaches only its intended recipients. The ps# can be invaluable in this regard.

Using present technology, individuals and businesses can authenticate who is accessing the information on their personal computers and their company network by combining any two or three variables: the traditional *something you know* mechanisms such as login names and passwords; *something you have* items such as hardware keys (dongles) and smartcards, and *something you are* aspects such as biometric measures.

With the launch of the Intel Pentium III processor and its ps# technology, the PC now has another *something you have* item. It is an access token that can be used in conjunction with passwords to help ensure that only the intended platform receives sensitive corporate information. For example, an Internet-based travel agency network can validate a system's processor serial number to make sure that sensitive pricing information is pushed only to authorized travel agents' machines. The increased identification offered by the ps# also helps corporate intranets extend information to employee desktops, offering employees greater real-time access to their 401(k) plans, payroll, and other personal data once their ps# is validated. The ps# also allows businesses to broadcast sensitive video with synchronized presentations by adding another layer of authentication prior to pushing the presentation out to the user.

In business-to-business transactions, corporations can bind the ps# to their digital certificates and internal or external certificate authorities. Business partners can then gain access to private information only if they have their corporate certificate and are accessing the data from a validated platform. (For more information on how to validate system identity, see the section entitled "System Verification Based on Processor Serial Number" in this paper.)

Example 3: Information Management

As the flood of information rises and the PC becomes the primary vehicle for processing, storing, and accessing information, the management of information poses a greater challenge. The ps# provides a non-intrusive identifier that enables information service providers to customize the data and services that are delivered to the end user. The ps# also provides a better way to track and protect important or sensitive information, and it can improve applications such as data backup and restore protection, removable storage data protection, managed access to files, and confirmation of document exchange between appropriate users.

SYSTEM VERIFICATION

It is a challenging task to design a software system that can reliably identify a system in an open network environment, based solely on the serial number of a processor. Because the client system could be a hostile system controlled by a potential hacker, it is difficult for the server to determine whether the returned ps# information from the client system is valid or spurious. The Processor Serial Number Verification Reference Implementation (RI) offers a basis to solve this problem. The RI was a joint effort by Intel and Independent Software Vendors (ISV) to provide a way to extract the ps# from a client system in an open network environment and limit cross-correlation of information across Web sites.

Tamper Resistance

The RI's software agents are downloaded over an open network and are therefore exposed to attacks by hackers. To protect against these attacks, the software agents are designed using special tamper resistant techniques, appropriately called Tamper Resistant Software (TRS) agents. The TRS agents automatically detect and protect against potential hacker attacks. Typically non-processor-based hardware authentication solutions imply the use of privileged instructions; it is not available at the application level directly. The additional software layers such as device drivers are exposed to more hacker attacks. Processor serial number instruction can be executed at the application level directly and does not require special drivers. It makes the tamper resistant protection stronger. These TRS agents are available from different ISVs. A common set of API functions has been defined and made available to these vendors so that developers can easily use agents from a number of vendors interchangeably.

To provide safety against impostors, the framework adopts a protocol whereby the authentication or verification of the client happens on the server. Agents are used once for a short time and then are discarded, thus enhancing protection.

Privacy Protection

Authentication mechanisms play an important role in Web-based applications. However, some users or businesses may not honor a consumer's right to privacy, and they gather personal information about the consumer without the consumer's consent. Intel has taken several measures to address consumers' privacy concerns not only in the design of the processor serial number feature, but also in providing certain utility tools. Several RI features work to further enhance the protection of a user's data:

- Software agents that gather the ps# are packaged in a digital container (a cabinet file for Internet Explorer[*], and a Jar file for Netscape Navigator*) that is then digitally signed by the Web service provider and delivered to the client system. When the Web browser sees the container, it prompts the user to grant access rights to the software, ensuring that the ps# cannot be collected without the user's consent.
- The ps#, once read, is transformed into another unique identifier by hashing it with a service ID. The hashed value is then sent by the client agent to the server to be stored in the user database.
- The service ID is unique to each service provider. This precludes different Web sites from correlating user profiles due to the non-communicative characteristic of the hashing algorithm.

The hashing algorithm is designed to be a one-way, collision-free algorithm, which means one cannot infer the processor serial number given the hashed value and the service ID. Intel also recommends that Web service providers make their privacy policies available to the consumer.

Performance Considerations

Another important attribute of the RI design is the short download time for the agents. If the size of the software agents is large, the user might have to wait for a long time before getting access. To reduce download times, agents used in the RI are limited to about 35 Kbytes. However, the quality of protection is proportional to the size of agents: the larger the size, the better the protection provided. A balance was reached with a small agent that can protect against attacks for a sufficient time. Protection is augmented by dynamically renewing the agents and by using a time-out mechanism on the server.

REFERENCE IMPLEMENTATION ARCHITECTURE

The RI framework consists of a client module, a server module, and a protocol for communication between the client and the server.

The client module consists of two types of client agents: a registration agent, which is a non-armored module for client registration; and a verification agent, which is a TRS armored module for client verification. Each agent consists of a Java* applet and native code DLL that are packaged together in digitally signed containers.

The server module manages client sessions and authenticates the client system in addition to providing access to the Web site. The Web server also stores the registration code with the user name and password in a backend database.

[*]Other brands and names are the property of their respective owners.

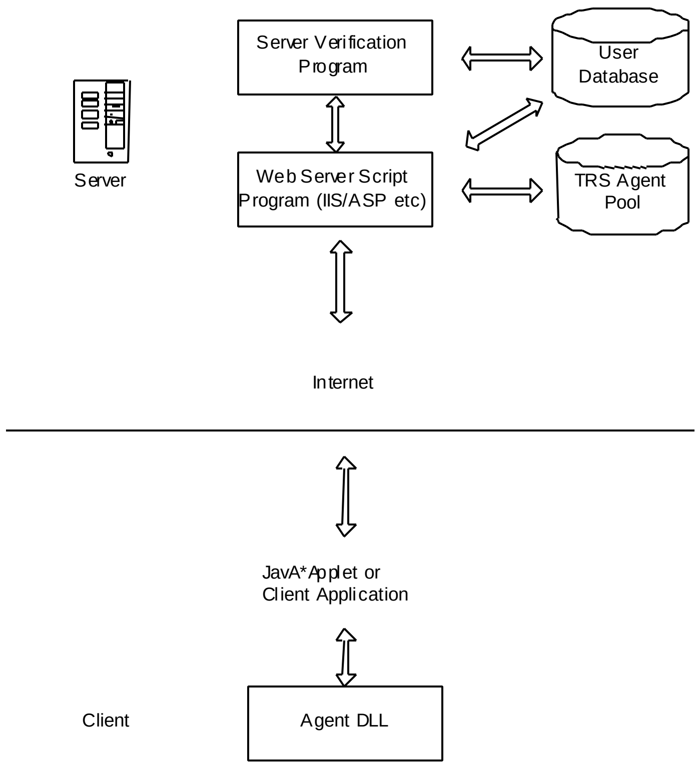

Server Verification Program
User Database
Server
Web Server Script Program (IIS/ASP etc)
TRS Agent Pool
Internet
JavA*Applet or Client Application
Client
Agent DLL

Figure 2: Reference implementation architecture

System Verification Flow

When a user first logs on to the Web server, the server asks for a user ID and password and then downloads a registration agent to the client system. The registration agent computes a registration code that is a hash of the ps# and service ID, and then returns it back to the server to be stored in the user database with the user's name and password. The server then sends a randomly selected verification agent from a pool of agents. Each agent is tamper resistant and embeds a unique secret value. It is only used once during a pre-defined time duration (usually about ten minutes). The agent is designed to sustain an attack by a very experienced hacker during that time period. Once the verification agent is downloaded to the client system, it calculates the verification code and sends the verification code to the server. As a measure of extra protection, the server times out during these sessions if it does not receive a valid verification code from the client within a pre-determined time.

The verification code is computed by first hashing the ps# and service ID (the same as the one used for registration). The resulting registration code is again hashed with additional unique secret values embedded within the verification agent. This results in a verification code, which is sent back to the Web server for verification.

After the server receives a valid verification code from the client, it first stores the returned value temporarily. Then the server calculates an authentication code by hashing the previously stored registration code with the unique secret value embedded in the particular verification agent sent to the client. The server then compares the authentication code with the verification code. If the two values match, the client system is authenticated, and the user can access content or obtain the requested service.

Supported Environment

Client agents support Microsoft[*] Windows* platforms (Windows NT*, Windows* 98 and Windows 95*). The RI supports Microsoft Internet Explorer* 4.0, Netscape Navigator* 4.x, and AOL* browsers. Both uniprocessor and multiprocessor client systems are supported. For multiprocessor systems, the ps# is gathered consistently from the same processor selected from the set of available processors. Similar software can be developed to support other client operating systems, such as Linux.* For server environments, the Reference Implementation supports both Windows NT and Unix.*

CONCLUSION

The serial number feature of the Pentium® III processor is designed to provide a unique identifier for each processor shipped by Intel with this feature to be visibly retrieved using application software. The CPUID instruction enables a natural method for providing this information with minimal impact to the processor design or to future implementations.

As mentioned in this paper, several different categories of applications can greatly benefit from a processor-based serial number capability. The most obvious areas are enterprise asset management, information management, and management for E-Business.

Unlike other means of deriving unique identifiers, the ps# feature implementation of the Pentium III processor is not impacted by a change of system hardware or software configuration (i.e., network card, IP address, etc.). Embedding this feature in the processor provides multiple benefits:

[*]Other brands and names are the property of their respective owners.

- Consumers and service/content providers have greater confidence in this feature due to the increased tamper resistance of the unique identifier.
- Visibility of the ps# to application-level software. Typical non-processor based solutions imply the use of privileged instructions (PL0) not available to application-level code or common across platforms.

As a result, we expect the ps# feature of the Pentium III processor to be of particular value to groups such as information providers and IT managers, and also to consumers as new applications take advantage of this feature.

ACKNOWLEDGMENTS

We thank those who helped to initiate, define, and refine the processor serial number feature and application program. Among the most active were Rob Sullivan, Ticky Thakkar, Natasha Oza, Vishesh Parikh, Jim Kolotorous, Susan Wojcicki, and Peter Ruscito.

REFERENCES

[1] *Pentium® Pro Family Developer's Manual, Volume 2: Programmers Reference Manual*, Order Number 000900-001, Intel Literature Sales, Mt. Prospect, IL, 1996, pp. 11-73 to 11-79.

[2] Mohsen Alavi, Mark Bohr, Jeff Hicks, Martin Denham, Allen Cassens, Dave Douglas, Min-Chun Tsai, "A PROM Element Based on Salicide Agglomeration of Poly Fuses in a CMOS Logic Process," 1997 IEEE International Electron Devices Tech Digest, December 1997, pp. 855-858.

AUTHORS' BIOGRAPHIES

Stephen Fischer is a staff design engineer with Intel Corporation's Folsom Design Center, which is responsible for the microcode and microarchitecture related design of the Pentium® III processor. Prior to that, Stephen was involved in various programs including EISA chipset definition, PCI bus and chipset definition, and the Intel MMX™ technology definition. He received a B.S. degree in computer engineering from CSU-Sacramento in 1985 and currently holds six U.S. patents. His e-mail is sfischer@pcocd2.intel.com.

James Mi is manager of Enabling Technology with Intel's Content Group, which is responsible for application architecture and development. Prior to that, he worked in marketing, software and hardware development at Intel's Content Group, TCAD, and Flash TD. James received a B.S. degree in physics from Fudan University, China, in 1989 and an M.S. degree in EE from Princeton University in 1991. He joined Intel in 1992. He holds seven U.S. patents. His e-mail is james.mi@intel.com.

Albert Teng is director of New Technologies with a focus on client/server applications for enterprise/e-commerce solutions, security, and knowledge management. Previously he was the general manager of Intel China and held an engineering management position in the Microcomputer Labs. Before joining Intel in 1985, Albert worked at AT&T Bell Labs and at the Illinois Institute of Technology. He received his Ph.D. from Ohio State University in 1979. His e-mail is albert.y.teng@intel.com.

Architecture of a 3D Software Stack for Peak Pentium® III Processor Performance

Paul M. Zagacki, Deep Buch,
Emile Hsieh, Daniel Melaku, Vladimir Pentkovski, Microprocessor Products Group, Intel Corp.
Hsien-Hsin Lee, EECS-ACAL, University of Michigan, Ann Arbor

Index words: 3D, Graphics, Performance, Pentium® III, Driver

ABSTRACT

In this paper, we analyze the benefits of key architectural modifications to a conventional 3D graphics software stack (application, library, and graphics driver). We do not propose a new 3D pipeline architecture; rather, we focus on improving the efficiency with which it is practically implemented. It is certainly possible to target specific layers of a 3D software stack for optimization and to realize significant performance gains with the Pentium® III processor and Internet Streaming SIMD Extensions. However, we will show that optimizing the kernel layers of the 3D software stack enables the user to take maximum advantage of the latent capabilities of the Pentium III processor. We use, as a case study, a geometry pipeline implementation, the Architecture Geometry Engine, developed by the Pentium III Architecture team (referred to as ArchGE) and a 3D scene manager. In this paper, we present performance data, based on our measurements, to demonstrate the benefit of the architectural enhancements.

INTRODUCTION

The prohibitive cost of applying the algorithms necessary to compute geometry and lighting in a conventional 3D pipeline has long kept 3D in the realm of high-end workstations. Figure 1 illustrates the classic 3D pipeline structure, which consists of several key components. First, the geometry and lighting calculations are performed on the system's host processor.[1] The application's 3D models are transformed into their virtual worlds, and lighting information is generated. These calculations are done in either a popular 3D library (OpenGL* or Microsoft's Direct3D* for example) or by the application. The generated information is then handed to another component, a 3D graphics controller, for rasterization (conversion into a 2D pixel representation of the image) on the computer screen. Keeping these two components in balance is one of the fundamental challenges that high-performance 3D engine development must address.

[1]This paper assumes a basic understanding of 3D graphics. For an in-depth review of this material refer to [1].

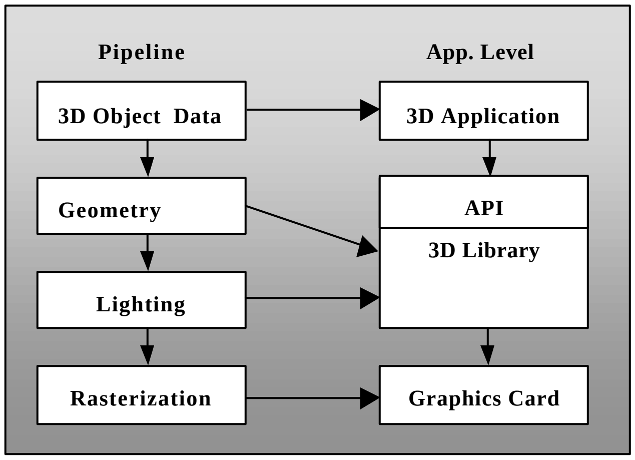


Figure 1: Typical 3D pipeline structure and its associated application-level components

An increase in graphics' controller performance means the 3D libraries built to deliver the geometry information to the cards must also increase in performance to keep the division of work in balance. The Internet Streaming SIMD Extensions were developed, in part, to increase the efficiency and throughput of the geometry and lighting calculations thus realizing higher system performance. However, to achieve peak 3D performance with the Pentium® III processor, components beyond the kernel level should also be optimized.

In order to demonstrate the peak 3D performance and usage models for the Pentium III processor, we developed the Architecture Geometry Engine (ArchGE) to fit into the 3D library layer (Figure 1). ArchGE incorporates the Internet Streaming SIMD Extensions to boost the geometry and lighting performance, but also adds two new architectural extensions to a general purpose 3D library:

1. MultiPrimitive API extension
2. "Online" driver model

The exact kernel-level speedup achieved with Internet Streaming SIMD Extensions over x87 code varies depending on the type and number of lights (infinite, local, specular, etc.), amount of clipping, primitive type, and other content variables.[2] When used in typical transformation and lighting kernels, we expect to see 1.4-2.0x the kernel-level performance over optimized x87 floating-point code for single light workloads. Workloads with multiple lights and larger primitive sizes are expected to see in excess of 2.0x the performance over optimized x87 implementations. Additionally, we have measured highly tuned, custom engines that see 2.5x–2.75x kernel-level performance. However, only a percentage of this kernel-level performance translates into application-level performance, the important measure for the consumer. The application performance increase is governed by *Amdahl's Law* and is typically less at the application level than at the kernel level unless additional optimizations to the software stack are made [2].

The MultiPrimitive API extension allows an application to gather all the primitives (e.g., strips, fans, vertex buffers) that share identical render state information and submit them in a single API call to the library. The MultiPrimitive optimization has been shown to provide 17%-40% of additional performance over conventional (single primitive per call) methods for drawing primitives. The additional performance is a result of the amortization of call overhead and vertex prefetch costs over a greater number of vertices being processed. The reduction of time spent in "startup" cost translates to more time spent in useful geometry and lighting computations that are accelerated by Streaming SIMD Extensions.

The second key extension introduced in ArchGE is an "online" driver (OLD). The OLD mechanism allows the graphics controller's driver to present the final destination buffer for the transformed and lit vertices directly to the geometry pipeline. Typically, in a general purpose API, all the vertices are transformed and lit, then placed in a buffer controlled by the library. The library signals the graphics controller when it is safe to take the buffer and render the information. When the buffer is ready, the device driver must copy the data from the library's buffer into the controller's memory (typically allocated in AGP memory). There are three issues with this methodology. First, moving large batches of transformed and lit vertices between library and graphics memory exercises the processor bus but not its computational throughput, thus leading to inefficient use of available resources. Second, this process typically generates excessive cache write-back activity (moving modified lines from a smaller, faster cache level to a larger and slower cache level or memory). This tends to aggravate the loading of the processor buses, reduce the efficiency of the cache hierarchy and prefetch instructions, and reduce the throughput of geometry computations. Third, the additional copy and formatting of the data by a typical device driver can increase driver execution times by up to 10x that of an OLD approach. This time spent in additional data movement is not time spent doing meaningful computations. OLD solves each of these issues by allowing the graphics pipeline to deposit transformed and lit vertices into the graphics controller's local memory, as they are calculated. The "direct deposit" of vertex information increases the concurrency between the geometry and lighting computations (computation intensive) with the storage of the results (bus intensive). This increased concurrency has been demonstrated to provide an additional application-level performance speedup of 30%-80% relative to a typical offline driver implementation.

The remainder of this paper discusses the following methods of 3D software stack optimizations (see Figure 1) and how these optimizations affect application-level performance:

- *3D Library/API Layer:* batch multiple primitives per drawing command, single pass vs. multiple pass geometry pipeline
- *Device Driver/Graphics Controller Layer:* online driver delivery of processed vertices
- *3D Application Layer:* object-level clipping and render state sorting

With all of these optimizations in place, ArchGE is able to display nearly 2x the peak application-level speedups of optimized x87 floating-point pipelines on similarly configured machines running identical workloads. While existing 3D libraries and device drivers are able to perform

[2]Kernel-level performance for our study is defined as including transformation, culling, specular lighting, transposition into graphics controller vertex order from a SIMD format, and storing the processed vertices to AGP memory.

the computations necessary for real-time 3D graphics, the techniques described in this paper add significantly to the overall performance for such implementations.

AMDAHL'S LAW

Amdahl's Law governs how much kernel-level speedup translates into application-level performance. Simply stated (using a 3D pipeline as an example), *Amdahl's Law* states that the amount of application-level speedup that optimizing transformation and lighting produces is limited to the percentage of time the software spends in this optimized code.

The example in Figure 2 shows an application of *Amdahl's Law* to 3D. Here we apply the Pentium® III Internet Streaming SIMD Extension instructions to transformation and lighting within a 3D application stack. We show a 2x kernel-level speedup and spend 50% of our time in these 3D geometry routines.

$$Speedup_{overall} = \frac{ExecutionTime_{old}}{ExecutionTime_{new}}$$

$$= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$= \frac{1}{(1 - .5) + \frac{.5}{2.0}} = 1.33x \text{ Speedup}$$

Figure 2: Amdahl's Law for predicting application-level performance applied to sample Xform and lighting optimizations

Based on Figure 2, if the performance of transform and lighting (3D library layer in Figure 1) increases 2x, yet only 50% of the total time is spent in this code, this translates to an overall application speedup of 1.33x. While a 1.33x performance increase is impressive, it is not quite the 2x we saw at the kernel level. It is clear, that in addition to porting transform and lighting routines to Internet Streaming SIMD Extensions, it may also pay off significantly to optimize other portions of the application stack to achieve peak Pentium III processor performance.

Many of the optimization techniques described in the following sections of this paper are designed to help defeat the performance-limiting affects of *Amdahl's Law* by increasing the time spent in the "enhanced" code segments.

3D LIBRARY AND API OPTIMIZATIONS

There are several popular 3D libraries and countless custom engines available to handle most of the details behind manipulating objects in three dimensions and displaying them on a 2D monitor. Existing 3D libraries typically have architectures that may potentially limit the performance an application can realize on a processor like the Pentium® III processor.

Multi-Pass Vertex Processing

Current 3D libraries normally have a multiple-pass structure for operating on input vertex information. In a multi-pass geometry pipeline, the vertices are processed through several individual loops. Each loop processes all the vertices submitted to the pipeline through transformation, backfaced culling (removal of non-forward facing triangles) and then lighting (MP half of Figure 3). There are two issues with this approach:

1. Complicated cache management code
2. Small basic code block sizes with which to interleave memory and computation instructions

Multi-pass processing is heavily dependent on cache management and potentially breaking vertex blocks, submitted for transformation and lighting, into cache sized increments. After absorbing all of the cache misses incurred during the transformation phase, the pipeline should not also have to service misses during the culling and lighting portions (even if the data stays in the L2 cache, there is still a small penalty to access it).

In addition to the extra programming efforts to directly manage cache usage for a multi-pass implementation, a small basic code block size makes it difficult to effectively interleave memory accesses and computation. Ideally, the memory leadoff/latency times for a loop should be balanced by the computation time within that loop. In a multi-pass pipe there is rarely enough computation per loop iteration to balance the load/store requirements. This makes our critical code sections memory bound and not very scalable as processor core frequency increases. (System memory performance historically lags behind processor performance.)

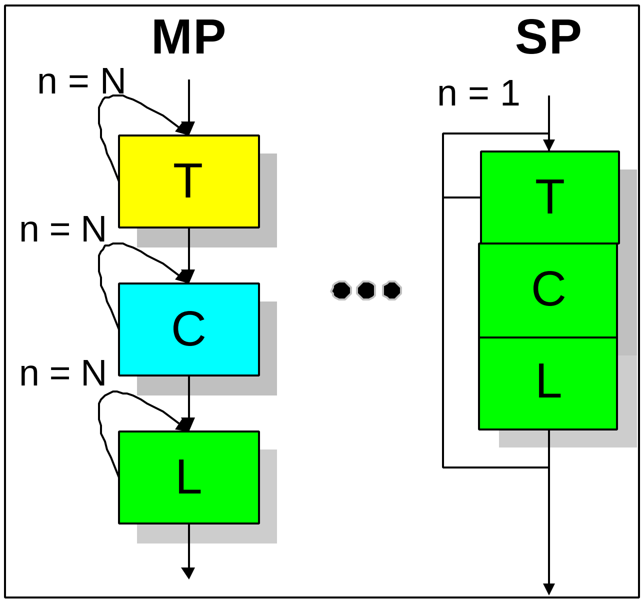


Figure 3: Multi-pass (MP) vs. single-pass (SP) geometry pipe (T = transform, C = backface cull, L = light, n= number of iterations, N = number of vertices)

Single-Pass Vertex Processing

In order to address the issues of a multi-pass pipeline structure, ArchGE implements the single-pass (SP) methodology shown in Figure 3. The key difference in this approach is that a few vertices are processed through transform, culling, lighting, and writing to the graphics controller's memory in a single loop.[3] This eliminates the need to add code to carefully manage cache utilization and also increases the basic block size significantly. The Internet Streaming SIMD Extension PREFETCH instruction is used to hide memory latency behind the computation performed in the pipe. Data, which will be transformed (x, y, z coordinate information) during the next iteration of the loop, is brought into the cache while transforming the current vertex. The same methodology applies to normals and texture coordinate(s) values during lighting computation.

By using PREFETCH instructions and implementing a large basic block, ArchGE is able to significantly increase the concurrency between the memory and computation. Our studies have shown that, as a result of this increase in coherency, a single-pass pipeline is 20%-30% faster then an optimized multi-pass pipeline. Since the ArchGE pipe tends to be more compute bound than its multi-pass counterpart, it should also scale more effectively with processor frequency.

MultiPrimitive API Extension

How vertices are submitted to a geometry pipe is almost as important as how they are processed. Most 3D libraries support many different ways to pass the application vertex information through the application programmer interface (API[4]). Vertices are grouped together into primitives (typically triangle-based) by the application and then passed to the library for transformation, lighting, and then rasterization by the graphics controller. OpenGL*, for instance, supports ten types of these primitives ranging in complexity from individual points to quadrilateral strips [4]. Since most graphics controllers accept information in triangle-based format, these are currently the most popular primitive types. Figure 4 demonstrates three such primitives.

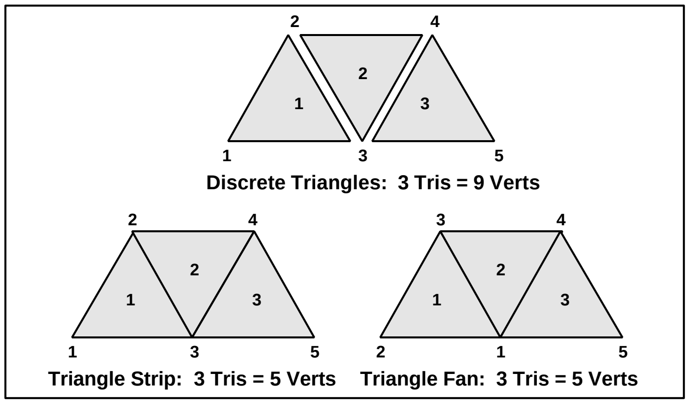

2
4
1
2
3
1
3
5
Discrete Triangles: 3 Tris = 9 Verts
2
4
1
2
3
1
3
5
Triangle Strip: 3 Tris = 5 Verts
3
4
1
2
3
2
1
5
Triangle Fan: 3 Tris = 5 Verts

Figure 4: Different triangle primitive types and the vertices necessary to draw them

Most existing graphics libraries allow an application to submit only one primitive at a time for processing. This means that an application can only process one triangle strip, one triangle fan, or one indexed list of vertices per function call (or whatever primitives are supported by the library). Since most primitives are comprised of relatively few vertices, the overhead involved in just making the function call to process each individual primitive becomes significant.[5]

[3]In the case of ArchGE a few vertices is actually four, which nicely correlates to the Pentium® III processor's internet S.S.E. register width.

[4] The API is the set of function calls a program can make to interact with a library.

[5] This observation is based on a study of several current games and benchmarks. A similar observation was made

The overhead for processing a single primitive can be broken into two parts: additional instructions outside of geometry computations and memory de-pipelining. The obvious source of additional work is the added instructions and cycles necessary to push and pop parameters, set up transform matrices and lighting information, validate parameters, etc. This was measured to be on the order of a thousand cycles per call in some popular libraries. This is a very significant amount of time if the application is submitting a small number of vertices per call.

In the ideal case, the Pentium III processor with Internet Streaming SIMD Extensions allows for almost complete overlap of memory accesses and computation. This is achieved by fully pipelining memory accesses using the PREFETCH instruction (lower portion of Figure 5).

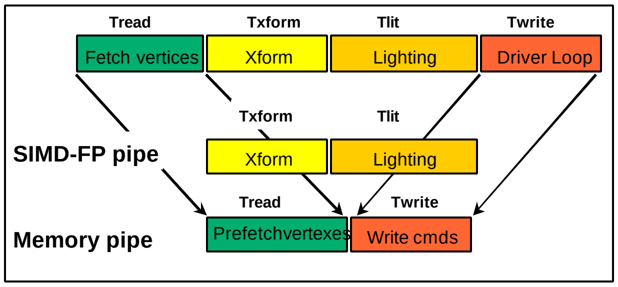


Figure 5: Ideal picture of increased memory and computational concurrency within a 3D pipeline

Each box in Figure 5 represents a block of processing time in a simplified 3D pipeline. The top portion of the figure is a conventional pipe, with serial memory and computation (a simplification since even older processor families allow for a small amount of concurrency between memory and computation). The bottom portion of Figure 5 shows what can be achieved by utilizing the PREFETCH and streaming store features of the Pentium III processor. Practically, however, an effect we refer to as "memory de-pipelining" occurs at primitive boundaries causing the total time in our ideal case to stretch somewhat [8]. For example, there can be "startup costs" associated with prefetching the first several vertices of a primitive during which computation is effectively stalled waiting for the data. For nested loops, memory de-pipelining can occur during the interval between the last iteration of an inner loop and the next iteration of its associated outer loop.

---

by [11] regarding some of the ViewPerf benchmark datasets.

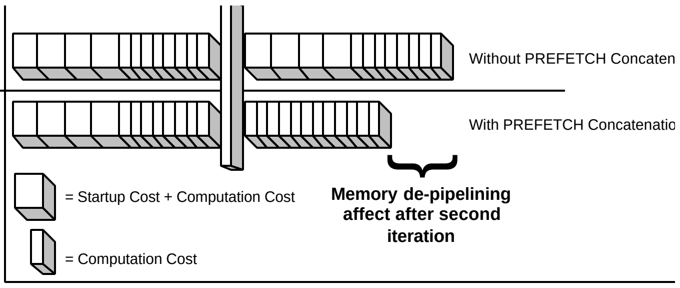

Without PREFETCH Concaten
With PREFETCH Concatenatio
= Startup Cost + Computation Cost
= Computation Cost
Memory de-pipelining affect after second iteration

Figure 6: Memory de-pipelining between two short primitives

Figure 6 shows a graphical example of the effects of memory de-pipelining. In the figure, the large boxes represent the amount of time to do normal computation plus the time spent waiting for initial PREFETCH instructions to return data to the cache (which delays completion for several of the initial iterations of geometry processing). The smaller boxes represent the amount of time necessary to complete computation in the steady-state.

The recommended technique to alleviate the performance issue of memory de-pipelining is "prefetch concatenation." Concatenation can bridge the execution pipeline bubbles between the boundary of an inner loop and its associated outer loop by using the PREFETCH instruction to “look ahead” to the next outer loop iteration. In the example outlined in Figure 6, the geometry pipeline “looks ahead” across primitive boundaries. It is clear that if an API only allows an application to submit a single primitive per call, this technique cannot be used at primitive boundaries to amortize the memory start-up costs for each primitive submitted for processing. This is especially important when dealing with primitives containing relatively few vertices (less then 100).

In order to reduce both the impact of an application calling through the API layer for every primitive and the memory de-pipelining effects, ArchGE implements a MultiPrimitive method for passing primitives to the geometry engine. This allows an application to pass a list of primitives and a corresponding list of primitive lengths to ArchGE with one call. MultiPrimitive generates a 40% increase in application-level performance for the ArchGE/Scene Manager software stack. Figure 7 shows details of the sensitivity of MultiPrimitive to primitive size and the number of primitives in a batch. In the best case of small primitives with many primitives in each call, MultiPrimitive achieves over 400% the performance of a single primitive API. At the low end of the spectrum, very large primitives (65 – 120 vertices per primitive) with only two per call, MultiPrimitive is still able to achieve a 20% increase in application-level performance. Based on studies of existing games and benchmarks, we anticipate that this feature could potentially generate a 30%-40% application-level speedup for typical workloads.

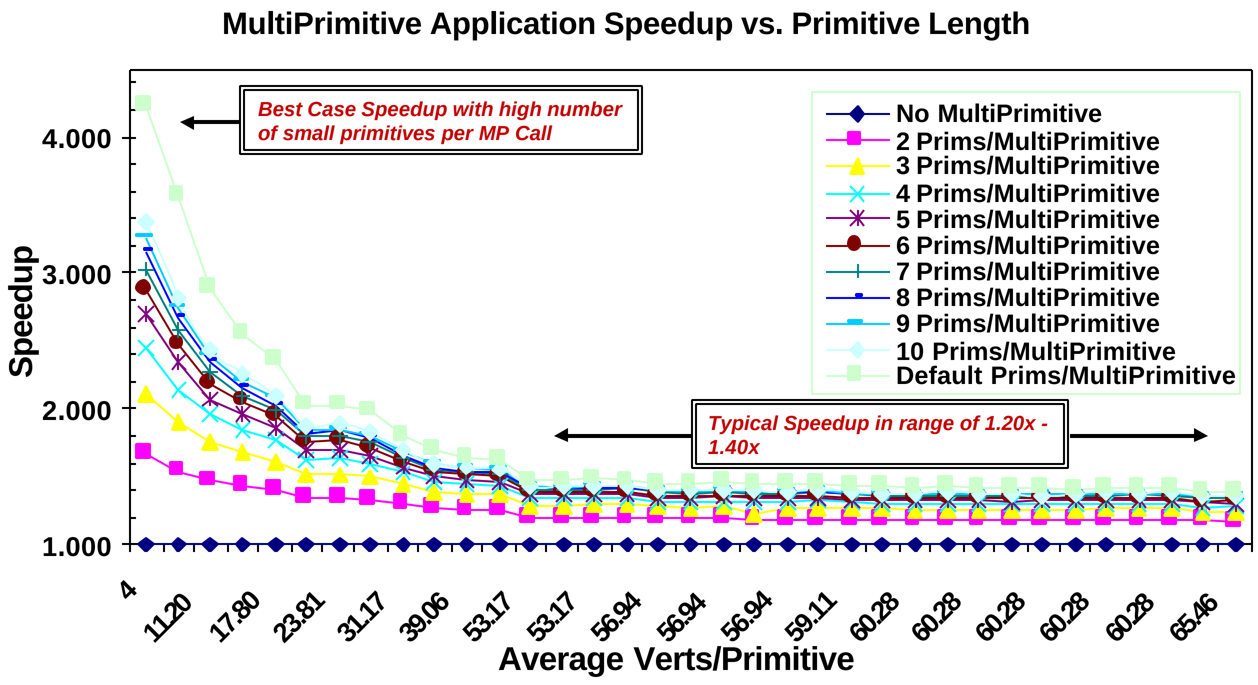


Figure 7: MultiPrimitive speedup sensitivity to primitive size and the number of primitives batched with each call (x-axis is non-linear)

The results in Figure 7 were generated by varying two variables: the maximum vertices per primitive and the number of primitives per batch. As we increase the maximum number of vertices allowed per primitive, the average number of vertices per primitive does not necessarily increase in a linear fashion (along the x-axis). The scene used for the experiment documented by Figure 7 had an original structure of 66 vertices per primitive on average. As we get closer to the original maximum average primitive size, moving right along the x-axis in Figure 7, a compression in the x-axis result occurs because additional vertices on a per-primitive basis do not affect the end average to any great extent.

DEVICE DRIVER OPTIMIZATION

Some conventional 3D libraries implement an *offline* driver model. Vertices are transformed, lit, and then stored in a temporary location within cacheable memory. The geometry engine then signals the graphics controller's driver that the buffer is ready, and the device driver begins to move the information from the temporary, cacheable memory to local memory controller memory (typically in a write-combinable and uncacheable memory range[6]). Looking back at the top portion of Figure 5, we can easily see that this will hurt the concurrency we are trying to build between memory access and computations. The conventional driver portion of the time is indicated by the "Driver Loop" time bar.

[6] See [9] for more information on Pentium® III processor memory type definitions.

In addition to reduced concurrency within a typical geometry pipeline, *offline* driver models also have a tendency to upset the utilization of the external bus (the bus between any CPU core and memory). Many cache lines are modified in the process of storing all of the command and vertex information for a primitive (post transform and lighting information). This can easily evaporate all the careful cache utilization work done in the application and the transform and lighting routines by writing unanticipated data to the caches. Quickly, the application finds itself faced with modified cache lines that need eviction prior to pulling fresh cache lines that contain the current data necessary for computation. The modified line evictions cause an unnecessary load on internal and external busses and can significantly hurt algorithm performance.

ArchGE solved both the problem of decreased concurrency between memory and computation and the issue of inefficient cache management by implementing a different driver model. OLD differs from a conventional driver model in one simple area: a large temporary buffer for transformed and lit vertex information is not required. With OLD, vertices are transformed and lit (four at a time in our single pass implementation) and then stored immediately to memory presented by the graphics controller. ArchGE implemented an OLD mechanism for a commercially available high-performance graphics controller. In the ArchGE geometry pipeline, four vertices are transformed, lit (if visible), and deposited directly in the graphics controller's memory. Since only small blocks of vertex and command information are stored directly to memory, we have increased the concurrency between the computation in transform and lighting and have also decreased the effects of excessive cacheable writes.

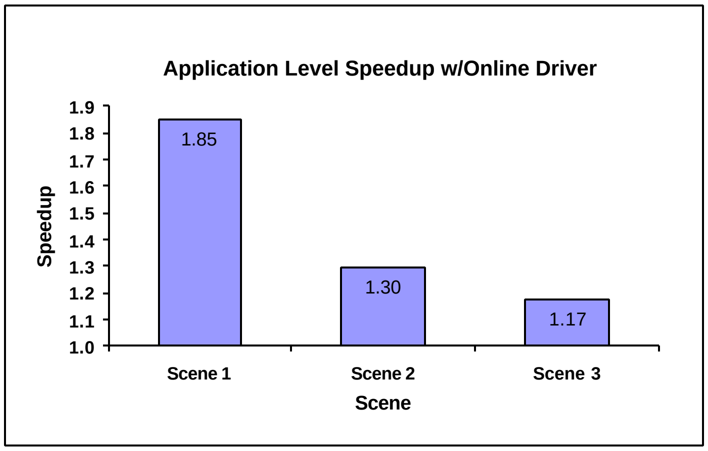


Figure 8: Application-level performance achievable with online vs. conventional driver models

Two very significant results were achieved with the implementation of an OLD in ArchGE. The first was the level of application speedup shown by this device driver model. Figure 8 demonstrates some of the possibilities of the *online* driver methodology. We measured three different scenes of varying complexity. The first scene in the chart, Scene 1, generates a 1.8x speedup over the same scene run with an offline driver in ArchGE. The high level of performance increase is attributable to the fact that the scene is very geometry intensive (approximately 82,000 vertices submitted per frame for processing) and is not bound by graphics controller fill rates (mostly small triangles). Thus, Scene 1 is more sensitive to processor capabilities and available bus bandwidth. With no external limitations on the performance of this workload, OLD is able to show close to peak performance.

The second scene in the chart, labeled Scene 2, is a close-up view of the first one and is much more sensitive to graphics controller fill rates. The somewhat lower speedup, 1.30x over a conventional driver model (vs. 1.8x for Scene 1) reflects the scene's sensitivity to fill rate. Finally, the third scene measured, Scene 3, shows a 1.17x performance delta over an offline driver. The third scene represents what we feel to be the worst-case content for ArchGE, since the geometric complexity of the content is relatively low, and the fill rate requirements are quite high, which make the graphics controller the performance bottleneck.

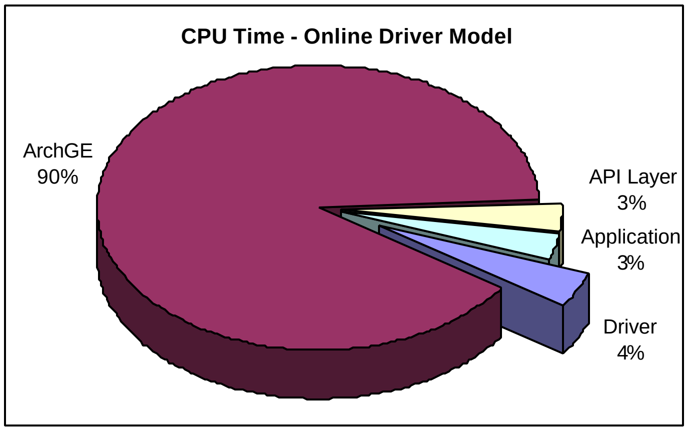
Figure 8 shows the large range of performance that is possible by implementing an online driver model. Our studies on the content of current games and benchmarks have indicated that results achievable fall between the peak of 1.8x and 1.3x. Tuning of the content for the third scene should yield results that fall into this range.

Increasing the amount of time spent transforming and lighting vertices is an additional effect of an online driver. With all of the additional time spent in code optimized with the Pentium® III processor's Internet Streaming SIMD Extension instructions, we are able to get much closer to the theoretical speedup generated by kernel-level optimizations of transformation and lighting (according to *Amdahl's Law* described previously). Figure 9 clearly displays the additional amount of time spent in meaningful computation in the case of the online driver. The pie-chart on the left of Figure 9 shows that 90% of our time is spent in the ArchGE library transforming and lighting vertices. In contrast, the pie-chart on the right of Figure 9 shows only 50% of our time in transformation and lighting, while we spend 46% of our time in device driver code (copying vertices to the graphics controller's local memory). Both of the profiles shown were generated using ArchGE on a very complex scene, which is not limited by graphics card fill rates.

The online driver feature translates into more time spent transforming and lighting vertices and less time moving data in and out of the cache hierarchy. The increase in focus on transformation and lighting (coupled with the optimizations possible with the Pentium III processor) allows an application to increase the level of content and generate a more realistic user experience. Our measurements have shown realistic speedups in the range of 1.8x to 1.3x with an online driver.

3D APPLICATION LAYER OPTIMIZATIONS

Optimizing a 3D application stack starts at the very top, with the application code and the content itself. The structure of the content (type of primitive, number of vertices per scene, amount of textures, etc.) has a huge impact on the performance of an application. The manner with which this content is presented to the 3D library layer is also very important. The scene manager used in our study implements a few key optimizations that generate significant benefits.



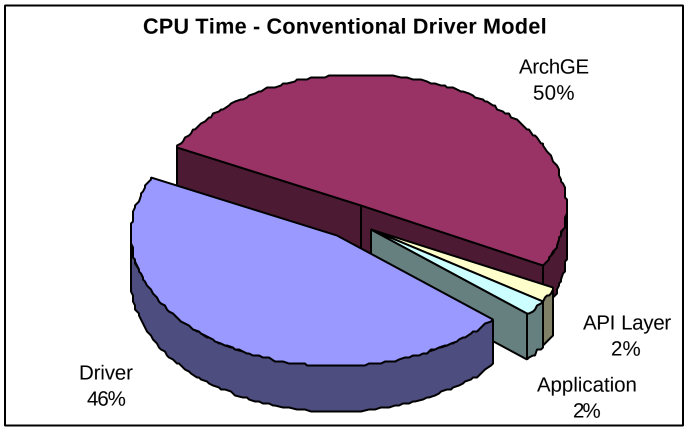


Figure 9: Effects of an online driver model to time spent in computation vs. data formating and movement

Render State Sorting

In order to convert the scene manager's 3D models into pictures on the screen, they must maintain a render state. The render state is a collection of information that tells the geometry engine and the graphics controller how to process incoming information (and display it on the screen). A render state contains information ranging from various transform matrix values to texture map addresses, which should be selected by the graphics card for rasterization.[7] Switching any portion of the render state, within a standard 3D library, is very expensive for the application.

The first cost associated with render state changes is in the 3D library itself. For the most part, a call into the 3D library to alter the current state involves a large number of processor cycles. Some of this time is spent in the library routine validating the new state and manipulating state variables. Another chunk of time goes to the device driver, where it typically goes through its own process of validation and setup.

The second cost of frequent render state changes is exhibited by the 3D graphics card. As graphics card frequencies and performance increase, so do the depths of their rasterization pipelines. It is typically necessary to flush the raster pipe of existing primitives and only restart after this has completed. This flushing leads to excessive bubbles in the rasterization pipeline and a less than effective utilization of precious pixel fill and triangle setup rate bandwidth.

By identifying the types of render states utilized and grouping primitives by distinct state setting at the time of the creation of the model/scene graph, our application was able to eliminate much of this overhead.

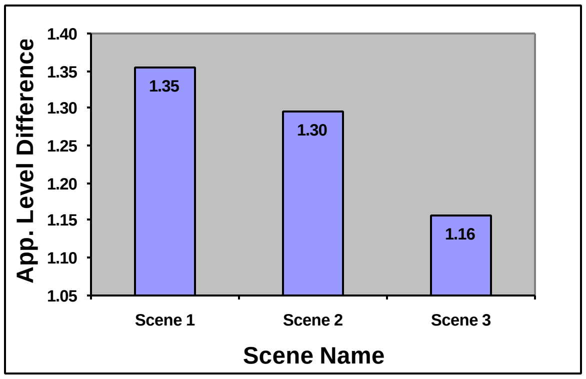
Object-Level Clipping

There are various ways a 3D application can avoid processing non-visible geometry. Our application uses bounding boxes around portions of the scene being processed to trivially accept or reject the primitives for further processing. The scene manager compares the points, which define the corners of the bounding box, with the dimensions of the viewport. There are three possible results of this comparison:

1. Completely outside of the viewing frustum; reject from further processing
2. Completely inside of the viewing frustum; submit for processing and indicate that no clipping is necessary
3. Points on the bounding box straddle the viewing frustum; submit for processing and indicate that clipping may be necessary[8]

[7]Rasterization is the process by which a graphics controller converts geometry information into pixel position and color information on a computer monitor.

[8] For more information regarding clipping primitives, please see [5] and [6].



Scene Name	Perf. Diff.	FPS w/BBoxes	FPS wout/BBoxes
Scene 1	1.35	43.6	32.19
Scene 2	1.30	47.6	36.71
Scene 3	1.16	38	32.86

Figure 10: Application-level performance difference with bounding boxes turned on

Using a bounding box with eight corner points for this comparison can save much processing time. The comparison involves transforming all the vertices for the primitive from their own object space into world or screen space. With a bounding box, only the eight corner points need to be checked. To check an individual point against an arbitrary bounding plane requires a minimum of four multiplication and three addition operations. These operations need to happen for the top, bottom, right, left, front, and back planes on the viewing frustum (a total of six planes). This becomes a total of 24 multiplication and 18 addition operations per point to account for each plane [3].

Implementing bounding boxes around 3D models minimizes the amount of computation necessary to trivially accept or reject vertices for further processing. Examples of the application-level performance impact of this optimization can be seen in Figure 10. This chart demonstrates a significant, application-level performance impact with bounding boxes enabled for object-level clipping. This effect was measured by turning the feature on or off within the application on three different scenes of varying geometric complexity. (Scene 1 contains the greatest number of vertices and Scene 3 the least.)

During our study on bounding box usage, we discovered that you can actually use too many bounding boxes around elements of a scene. The minimum number of vertices to include in a bounding box depends on many different factors and should be experimented with to determine what will work most effectively in any particular 3D software stack.

Implementing render state sorting and object-level clipping in our application layer has the potential to significantly boost the performance of the optimized ArchGE engine. The object-level clipping shows a 16%-35% application-level performance increase, and it brings ArchGE closer to peak Pentium® III processor performance by at least that much.

CONCLUSION

Software applications are exploiting more 3D graphics than ever before. The Pentium® III processor, with its Internet Streaming SIMD Extension instructions, can boost performance on 3D transformation and lighting over 2x that of optimized floating-point instructions. However, as shown in this paper, all of this kernel-level performance does not translate directly into application-level performance.

What we have outlined in this paper is a series of architectural optimizations for various levels of the 3D application software stack. Such optimizations can bring applications closer to realizing peak Pentium III performance for typical 3D graphics workloads. Utilizing a tuned scene manager and our ArchGE geometry engine, we are able to demonstrate close to 2x the application-level speedup of the Pentium® II processor at the same frequency on the Pentium III processor on a general purpose 3D software stack.

ACKNOWLEDGMENTS

All of the work and results outlined in this paper could not have been achieved without the significant help of the following people and organizations:

- **BMD Architecture:** We acknowledge Ken Castro for keeping our development and measurement process smooth with outstanding lab support.
- **MAP-PBA**: We acknowledge Shervin Kheradpir and Jeff Ma for developing the scene management software to support ArchGE and various experiments.
- **PMD Architecture**: We acknowledge Tom Huff for participating in discussions on ArchGE features, code reviews, and recommended improvements.
- **GCD:** We acknowledge Peter Doyle for his work on the definition of the i740™ definition of the online driver.
- **PDD:** We acknowledge Gerry Blank and Brandon Fliflet for prototyping and implementing the i740™ version of the online driver.

- **3dfx Interactive, Inc.**: We acknowledge Colyn Case and Andrew Hanson for helping to define and develop a Voodoo2^{TM*} version of the online driver for ArchGE that was used to generate much of the experimental data in this paper.

REFERENCES

[1] James D Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Morgan Kaufmann, San Francisco, CA, pp. 29-31.

[2] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, Addison-Wesley, Menlo Park, CA, pp. 201-283.

[3] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, Addison-Wesley, Menlo Park, CA, pp. 868.

[4] Mason Woo, Jackie Neider, and Tom Davis, *OpenGL® Programming Guide: Second Edition*, Addison-Wesley, Menlo Park, CA, pp. 42-45.

[5] Jim F. Blinn. and Martin E. Newell, "Clipping Using Homogeneous Coordinates," *SigGraph 1978 Proceedings*, pp. 245-251.

[6] Jim F. Blinn, *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann, San Francisco, CA, pp. 122-134.

[7] *Intel® Architecture Optimization Reference Manual*, available at http://developer.intel.com/design/PentiumIII/manuals/

[8] *Intel® Architecture Optimization Reference Manual*, pp. 6-13 – 6-15.

[9] *Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide*, pp. 9-4 – 9-13. Available at http://developer.intel.com/design/PentiumIII/manuals/

[10] *Intel® Architecture Optimization Reference Manual*, pp. 6-6 – 6-9.

[11] Chia-Lin Yang, Barton Sano, and Alvin R. Lebeck, "Exploiting Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications," Technical Report CS-1998-14, Computer Science Department, Duke University, September 1998.

AUTHORS' BIOGRAPHIES

Paul Zagacki is a senior processor architect for the Microprocessor Products Group in Folsom, CA. He holds a B.S. degree in computer science from the University of Michigan, Ann Arbor. He has worked for Intel since 1994 in the areas of high-level performance modeling for microprocessor architectures, Pentium® III processor software and benchmark analysis and optimization, and 3D graphics implementation, performance analysis, and tuning. His professional interests include computer architecture/microarchitecture, 3D graphics, compiler performance, and software/hardware performance analysis. His e-mail is paul.zagacki@intel.com.

Deep Buch is a staff processor architect in the Microprocessor Products Group in Folsom, CA. He received an M.Tech degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1989. He has been working for Intel since 1993 in the areas of processor architecture, platform technologies, and 3D graphics. Prior to joining Intel, Deep was a hardware specialist at Wipro Infotech R&D in Bangalore, India, working on ASIC and system level design. His interests are computer architecture, multimedia and communications. His e-mail is deep.k.buch@intel.com.

Emile Hsieh is a senior processor architect in the Microprocessor Product Group in Folsom, CA. He holds a B.S. degree from the National Taiwan University, Taipei, Taiwan, and a M.S. degree from Purdue University, West Lafayettte, IN, all in electrical engineering. His research interests include computer architecture, performance modeling and analysis, compilers, graphics, signal processing, and communications. His e-mail is emile.hsieh@intel.com.

Hsien-Hsin Lee is presently a Ph.D. candidate in computer science and engineering at the University of Michigan. From 1995 to 1998, Hsien-Hsin was a senior processor architect for the Microprocessor Products Group in Folsom, CA. While there he worked on design and performance modeling for the Pentium® Pro, Pentium® II and Pentium III processors. He holds a B.S.E.E. degree from the National Tsinghua University, Taiwan and an M.S.E. degree from the University of Michigan. His research interests include microarchitecture, memory system design, ILP optimization, and graphics architectures. His e-mail is linear@eecs.umich.edu.

Daniel Melaku is a processor architect for the Microprocessor Products Group in Folsom, CA. He holds a B.S. degree in computer engineering from California State University, Sacramento. Daniel has been with Intel since 1997, and has worked in the areas of performance projection, validation, and tool development. His interests include digital signal processing, computer animation, voice and image recognition, and artificial intelligence. His e-mail is daniel.melaku@intel.com.

Vladimir Pentkovski is a Principal Engineer in the Microprocessor Product Group in Folsom. He was one of the architects in the core team that defined the Internet Streaming SIMD Extensions for the IA-32 architecture. Vladimir led the development of the Pentium III processor architecture and performance analysis. Previously he led the development of compilers and software and hardware support for programming languages for Elbrus multi-processor computers in Russia. Vladimir holds a Doctor of Science degree and a Ph.D. in computer science and engineering from Russia. His e-mail is vladimir.m.pentkovski@intel.com.

Applications Tuning for Streaming SIMD Extensions

James Abel, Kumar Balasubramanian,
Mike Bargeron, Tom Craver, Mike Phlipot, Microprocessor Products Group, Intel Corp.

Index words: SIMD, streaming, MMX™ instructions, 3D, video, imaging

ABSTRACT

In early 1997, Intel formed an engineering lab whose charter was to apply a new set of instructions to the optimization of software applications. This lab worked with commercial software companies to increase the performance of their applications by using these new instructions. Two years later, this new instruction set has been made public as a principal new feature of the Pentium® III processor, the Streaming SIMD Extensions. Many of the commercial software companies' applications on which the lab consulted have been brought to market, demonstrating significant performance improvements by using the Streaming SIMD Extensions. This paper describes many of the principles and concepts developed as a result of that activity.

The Streaming SIMD Extensions expand the Single Instruction/Multiple Data (SIMD) capabilities of the Intel® Architecture. Previously, Intel® MMX™ instructions allowed SIMD integer operations. These new instructions implement floating-point operations on a set of eight new SIMD registers. Additionally, the Streaming SIMD Extensions provide new integer instructions operating on the MMX registers as well as cache control instructions to optimize memory access. Applications using 3D graphics, digital imaging, and motion video are generally well suited for optimization with these new instructions.

Data organization plays a pivotal role in the performance of applications in the above areas. This paper explores three data organizations (Array of Structure, Structure of Array, and Hybrid data orders) and their impact on SIMD processing performance. The impact of cache control instructions, such as the prefetch instructions, is also examined.

Examples of applying the Streaming SIMD Extensions to 3D transform and lighting, bilinear interpolation, video block matching, and motion compensation are considered. The principles applied in these examples can be extended to many other algorithms and applications.

INTRODUCTION

It is desirable to have many products available at the initial launch of a processor to help establish consumer interest. The development of these products begins with understanding the full potential of the new processor. This process requires optimizing select algorithms to achieve maximum performance. For the Pentium® III processor, that activity started in 1997 with a focus on the Streaming SIMD Extensions.

Although applicable to a wide variety of programs, the extended instruction set is designed to be especially effective in applications involving 3D graphics, digital imaging, and digital motion video. The purpose of this paper is to describe how those particular applications are best optimized with the new SIMD instructions.

Rather than optimize an entire application, specific algorithms or components were selected that would offer the best speedup. Analysis tools such as the VTune™ Performance Enhancement Environment [1] identified the most processor-intensive components of an application. The identified components were further examined for algorithms that execute similar operations on large data sets with a minimal amount of branching.

Data flow in and out of the processor is an important element in optimization so various data organization strategies were tested, including the impact of prefetch.

All algorithms were coded with and without the Streaming SIMD Extensions. The two versions of the algorithms were run on the same Pentium III processor platform to determine the relative performance difference.

DATA AND THE STREAMING SIMD EXTENSIONS

This section examines issues that must be taken into account to achieve the best possible performance with the Streaming SIMD Extensions. The order of data in memory and the methods by which such data are moved to and from the processor can have a significant impact.

SIMD and Memory Interactions

The floating-point instructions in Streaming SIMD Extensions generally operate "vertically"; that is, they operate between corresponding positions of the SIMD registers, or the equivalent positions of data being loaded directly from memory. Since the same operation must be done to all four floating-point values in a register, typically the best approach is to use each of the four positions of an SIMD register to store the same variable of an algorithm, but from different iterations. For example, if the algorithm is A[j] = B[j] + C[j], one would want to put four B's in one register, four C's in another register, and use a single SIMD add operation to create four resulting A's.

Each SIMD register can be thought of as a small array of data. A set of these registers can be thought of as a structure of arrays (SoA for short):

```
struct { float A[4], B[4], C[4]; } SoA;
```

This SoA approach is not always applicable. In the equation B[j] = B[j-1] + C[j], the dependency between iterations would require a different approach.

The Pentium® III processor typically loads data from memory 32 bytes at a time, from 32-byte aligned addresses. Each 32 bytes is stored to one "cache line" of the L1 and/or L2 caches. Frequent use of instructions that load or store data that is split over two cache lines will cause a significant performance penalty.

Most of the Streaming SIMD Extensions floating-point instructions that access memory require a 16-byte aligned address, thus avoiding the penalty. The `movups`, `movlps`, and `movhps` instructions were included to support unaligned accesses, at the risk of incurring the penalty. The `movlps` and `movhps` instructions can access 8-byte aligned addresses without penalty, since they move only 8 bytes at a time (compared to `movups` which move 16 bytes).

Using the Prefetch Instructions

Prefetch instructions can be useful for algorithms limited by CPU processing speed, ensuring that data is always ready as soon as they can be used. The prefetch instructions *load data ahead of use*, thereby hiding load latency so that the CPU can take full advantage of memory bandwidth. The Pentium III processor loads data to cache when a cache line is written to, so prefetches can also reduce latency for storing data.

Prefetch instructions can be useful for memory bandwidth-limited algorithms as well. For example, the `prefetchnta` instruction fetches data only to the L1 cache, avoiding some overhead incurred when data is also loaded to the L2 cache (as occurs with normal load and store).

Loading data only to the L1 cache, if it is not going to be needed again soon, also avoids unnecessarily evicting data from the L2 cache. When data is unnecessarily evicted, it can impose a double penalty. Modified cache lines that are evicted must be written back to memory and reloaded later when they are again needed.

The `prefetcht2` instruction might be used to load the L2 cache with a data set larger than can fit in the L1 cache. Meanwhile a CPU speed-limited algorithm could be executing and randomly accessing data. As it proceeds, it would find more and more of its data in the L2 cache.

To get the most efficient use of prefetch, loops should be unrolled (i.e., multiple passes of the algorithm should be in each loop iteration) so that each iteration prefetches and uses one cache line worth of each variable of the algorithm.

Data Order and SIMD Algorithm Performance

The SoA order is the most natural order for SIMD operations, so it would seem equally natural to use it as an order for data in memory:

```
struct
{
      float A[1000], B[1000], C[1000];
} SoA_data;
```

In some cases, this approach can work fine. But for a larger number of structure members, SoA can have memory access performance penalties. PC memory systems can only keep a limited number of pages (typically 4KB blocks) of memory "open" for fast access. If the number of members exceeds that number, so that each set of four values used in a SIMD computation must come from a different area of memory, the memory subsystem may spend inordinate amounts of time "re-opening" pages.

An Array of Structures (AoS) data order is more conventional in non-SIMD programming:

```
struct
{
      float A, B, C;
} AoS_data[1000];
```

Sequential processing through an AoS data set will find needed data close together in memory, thus avoiding the "open pages" limitation of SoA. However, the data are clearly not well ordered for SIMD computations.

The solution to this is generally to load the AoS data into SIMD registers and convert them to the SoA format via data reordering ("shuffling") instructions. This process can be thought of as "transposing" the data order. See Figure 1.

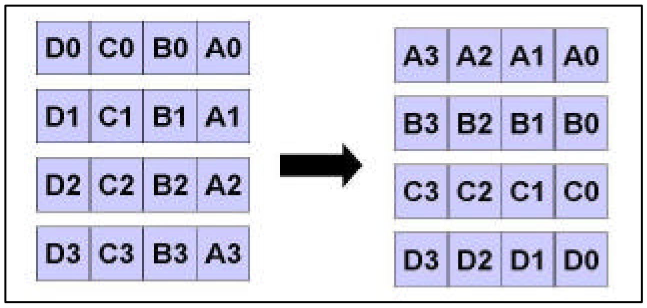


Figure 1: Transposing from AoS to SoA

While there is some performance cost due to this transposition, this approach generally works reasonably well, and may be the only viable solution if other factors mandate an AoS data order in memory. Existing data may be in AoS format; existing programs may have interface specifications that require AoS data; or data may be randomly accessed rather than sequentially accessed.

The Pentium III processor loads 32 bytes at a time from memory to cache. If there are members in an AoS structure that are not needed in the current computation, they will nonetheless be pulled across the memory bus, incurring unnecessary bus overhead, and limiting performance.

Data caching can sometimes offset this overhead by keeping data in cache until they are needed in a subsequent processing step. But for large data sets, the cache may not be large enough, and data may be evicted before they can be used. In general, it is a good idea to limit AoS structures to just members that will often be used at the same time. (This applies to non-SIMD code as well.)

It would be preferable, when AoS is not forced on us by external factors, to find a data order that preserves the AoS data adjacency, while supporting the SoA load order. An example of this "hybrid" data order is

```
struct
{
      float A[8], B[8], C[8];
} Hybrid_data[125];
```

As with SoA, this order allows the processor to load four values at a time (e.g., with `movaps`) from any member array. While structure members with the same index are not immediately adjacent, they are still close enough that they will usually be in the same memory page.

If a hybrid structure starts 32-byte aligned, the data will remain 32-byte aligned (since there are eight entries in each of the 4-byte float sub-arrays). This is convenient for Streaming SIMD Extensions instructions that require 16-byte alignment, as well as for prefetching a full cache line that contains just one particular member. For sequential processing of large data sets, it reliably provides good results.

Figure 2 illustrates the impact of SIMD and data order on the performance of a dot product algorithm.

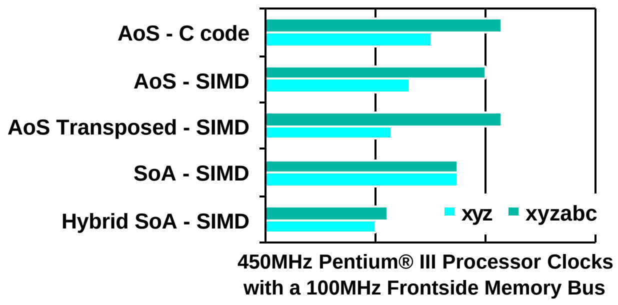


Figure 2: Data order and performance

A dot product was done between vectors of two large sets of 3-component (xyz) vectors. All were coded in C: The Streaming SIMD Extensions versions were implemented using "intrinsic functions" built into and optimized by an Intel® compiler. All use prefetch instructions to optimize the use of memory bandwidth. The "xyz" bars represent tests with data structures having only three members in the data structure, while the "xyzabc" bars represent tests where three extra structure members, not involved in the dot product, were included in the data set.

The Hybrid SoA approach gave the best overall performance. AoS algorithms did poorly (and became memory bound) when extra members were included in the same structure. The SoA and Hybrid SoA algorithms were nearly immune to extra structure members. The Streaming SIMD Extensions provided some small benefit to the AoS 'xyz' algorithm if the dot products were done one at a time (AoS – SIMD), and somewhat more if the data were transposed to SoA form before processing. The SoA algorithm was fully memory limited; it was unable to approach the best performance despite the natural SIMD ordering of data.

Optimizing Memory Use for Block Processing

Block processing algorithms typically read sequentially through a large array of data, modify the data, and write out to another large array. For example, converting an image from RGB format to YUV format entails sequentially reading the RGB components, computing the equivalent YUV components, and writing the latter out to a new array. An even simpler example is a block copy.

Many such algorithms will be memory bound, so anything that optimizes the flow of data is highly desirable. One attribute of such algorithms is that typically they process data once, and need not touch them again. In such a case, there is little point in saving results in cache, where they might displace other useful data.

The streaming store instructions (`movntps`, `movntq`) can be used to write results to the destination memory buffer without going through the caches. However, these instructions work best if they can take full advantage of write combining. Any access to memory or the L2 cache can cause premature flushing of the write-combining buffers, resulting in inefficient use of the memory bus. While writing out results with the streaming store instructions, data should only be read from the L1 cache, to avoid this performance penalty.

To ensure this is the case for a block-processing algorithm, a loop can be added that uses `prefetchnta` to read a sub-block of data (typically about 4KB) into the L1 cache. A normal processing loop would follow this, reading the L1 cached sub-block of data and writing results out with streaming store instructions. An outer loop, around both of the sub-block loops, would go through all the sub-blocks that make up the data set. One key issue arises when using this approach. The prefetch instructions only work when the virtual memory page addressed by the prefetch is mapped to a physical memory page by the Translation Lookaside Buffer (TLB) in the Pentium III processor. Typically the TLB is updated for a page the first time that page is accessed. Since the TLB has a finite number of entries (e.g., 64), page mappings that have not been used recently may no longer be cached in the TLB, which means the prefetch will not work.

To make sure the prefetches work, one merely needs to do one read from each 4KB memory page of the source data, shortly before starting the prefetch loop. Since initializing the TLB will take a while, if the read is done right before the prefetch loop, many of the prefetches might be quickly executed with no effect. This can be adjusted for in a variety of ways. For example, one can read once from an address 4KB ahead of the address where the prefetch loop begins, making sure not to read past the end of valid data.

The approach of breaking data into cache-fitting sub-blocks can also be useful if one wishes to do multiple passes over data that cannot all fit into cache. For example, one might wish to do a sequence of processing steps, each taking the previous step’s output as its input.

If one were to do each processing pass separately, intermediate results would have to be written out to memory and later reloaded from memory for the next step. Instead, one can often do all passes over each of many smaller, cache-fitting blocks, thereby minimizing memory data bus traffic.

TUNING 3D APPLICATIONS

In a typical 3D geometry engine, one would expect to find various functional components such as transformation, lighting and shading, clipping, culling, and perspective correction modules [2]. Deciding which component should be optimized can be difficult. Using the criteria discussed in the introduction, the transform and lighting functions were determined to be good candidates for optimization using the Streaming SIMD Extensions.

Transform and lighting functions are compute-intensive, SIMD-friendly inner loops that perform the same operation on large amounts of contiguous data. Use of the prefetch instruction allows data in either loop to begin loading several iterations prior to their use. The new approximation instructions are beneficial in eliminating long-latency square-root and division operations in the lighting loop, or in the transform loop when perspective correction is performed. Finally, clamping to a range of values within the lighting loop can be replaced by the new packed min/max instructions, eliminating two unpredictable branches per iteration.

The details of each optimization method are discussed below. In each case, data order and alignment are as discussed in the previous section.

3D Transform

The 3D transform is performed by multiplying a 4x4 transformation matrix by a 4-element vector. The vector is comprised of vertex elements X, Y, Z, and the constant 1, while the transformation matrix itself is calculated individually for each object in the scene. This operation produces intermediate values X', Y', Z', and W'. In some cases, the W' value is immediately used to normalize the intermediate vector (perspective divide), generating final values X", Y", and Z". Since the final result of the fourth element is always 1.0, the division of W' by itself can be ignored.

3D Transform Optimizations

Because the same transformation matrix applies to all vertices of a given mesh, and since there is a large amount of data to be processed, the transform was found to be a good candidate for SIMD programming. To experience the largest benefit from the Streaming SIMD Extensions, the full capacity of the SIMD registers was exploited. One register was loaded with four X values, X_0, X_1, X_2, and X_3, another with four Y values, and yet another with four Z values.

The first set of matrix elements was then loaded into a fourth register. To do this, two methods were possible. Each matrix element could either be (1) stored as a single floating-point value in memory, read into the lowest position of the 4-wide register using `movss`, then replicated four times using the `shufps` instuction; or (2) the element could be stored as an array of four identical floating-point values, aligned on a 32-byte boundary and read in four at a time using the `movaps` instruction. The latter proved to be optimal. Storing the entire matrix in this manner did increase the immediate size of the structure from 64 bytes to 256 bytes, but this was a small price to pay for the performance gained.

Matrix elements m_{01}, m_{02}, and m_{03} were loaded from memory in a similar fashion. With all of the data in registers in true SIMD format, the transform became a simple series of three multiply instructions followed by three addition intructions for each set of results (see Figure 3). Knowing that the same vertex data used to calculate the X' results would be needed to compute the Y' results, instructions were used in such a way as to overwrite the registers containing matrix data. Once this set of computations was completed, intermediate values X_0', X_1', X_2', and X_3' were written to a 32-byte aligned output buffer.

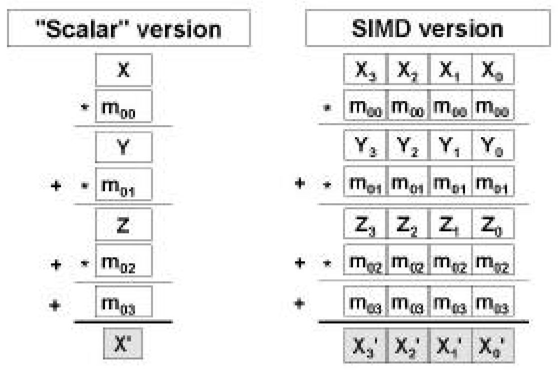


Figure 3: The SIMD transform produces four results while the conventional transform produces only one

The process was repeated for the first four Y values, this time loading matrix elements m_{10}, m_{11}, m_{12}, m_{13}, and again for Z' and W', with each of their respective matrix elements. The final result of the first transform iteration was 16 intermediate values. If at this point of the pipeline, a perspective divide is done, the `rcpps` instruction is of tremendous benefit. (See the section entitled *3D Lighting Optimizations* for more details.) Figure 4 compares the results of the Pentium III processor-specific code with the optimizations discussed above and a standard 'C' implementation of the same algorithm.

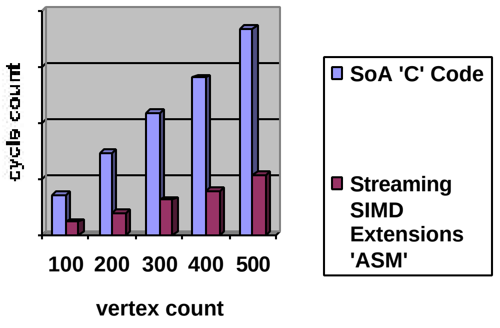


Figure 4: Transform cycle time for optimized Pentium® III processor assembly versus conventional 'C' code

3D Lighting

The point light is probably the most widely used light source in 3D graphics applications. To apply a point light to a given vertex, the vector from the vertex to the light source is first calculated. The length of this vector is computed and used to normalize the vertex-to-light vector. From the normalized light vector, the *diffuse* component of the current vertex is computed. Finally, the overall vertex color is calculated and checked to ensure that it falls within the range of [0.0, 1.0]. Values exceeding the range in either direction must be "clamped" to either the upper or lower bound. Once vertex color values are computed and clamped, they are stored to memory for use at render time.

3D Lighting Optimizations

The lighting distance calculation involves a standard square-root of a sum-of-squares. Again the full capacity of SIMD registers was used by loading four vertex values into each register and calculating the distance of four vertices from the same light source simultaneously. The normalization of the light vector was then performed by dividing the vector itself by the previously calculated distance. Though seemingly simple, these steps require two long-latency floating-point operations, the square-root and the divide, each of which requires about 36 clock cycles to complete. Beyond that, they are "unpipelined" instructions, meaning that no other instruction may be submitted to the execution port on which they are executing until the given instruction retires.

To overcome this, the Streaming SIMD Extensions include `rsqrtps` and `rcpps`, reciprocal approximation instructions that allow developers to accomplish the same workload as the long-latency instructions in a shorter time. By performing hardware table look-ups, these instructions have a reduced latency of two clock cycles and are fully pipelined, so that other operations may be issued during their execution. As a tradeoff, the instructions guarantee at least 11 bits of mantissa precision, as opposed to the full 23 bits offered by true single-precision instructions. Though 11 bits is typically enough for most 3D applications, some applications may require (or prefer) more.

The Newton-Raphson method is a mathematical algorithm designed to regain precision lost by this type of approximation. A single-pass Newton-Raphson iteration doubles the resultant accuracy to 22 bits; the 23rd bit can be recuperated after a second pass. An approximation followed by a single iteration is notably faster than its long-latency equivalent and can be determined by

$$\text{rcp}'(a) = 2*\text{rcp}(a) - a*\text{rcp}(a)^2$$

$$\text{rsqt}'(a) = (0.5)*\text{rsqt}(a)*(3-a*\text{rsqt}(a)^2)$$

Four normalized direction vectors were generated by multiplying the reciprocal square root of each distance squared by the previously calculated light vector. These values were then used to calculate the diffuse component of the vertex color.

The diffuse calculation involves a dot product of two vectors and could potentially produce a negative result. Graphics applications typically overcome this by performing a less-than-zero check and setting the value to zero if 'true.' This type of unpredictable, conditional branch performed once per iteration can be a performance bottleneck. To eliminate the branch, a `maxps` instruction was performed on the register of four diffuse results and a 4-wide register of 0.0 values. The max instruction zeroes out the negative values, while permitting non-negatives to pass through unchanged.

A similar optimization can be performed when calculating the final vertex color. This time, the tendency is for the value to exceed 1.0. The `minps` instruction clamps values, above the threshold, to 1.0 without the use of conditional branching. Large advantages of lighting optimizations versus a standard 'C' language lighting function are shown in Figure 5.

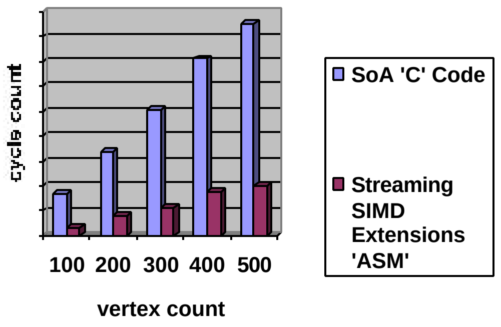


Figure 5: Lighting cycle time for optimized Pentium® III processor assembly versus conventional 'C' code

DIGITAL IMAGING

Digital imaging applications are typically comprised of algorithms wherein a small set of mathematical operations needs to be performed on large volumes of pixel data. Furthermore, each pixel consists of four components: Red, Green, Blue, and Alpha values. Hence, MMX™ technology significantly enhanced the performance of digital imaging applications. As most state-of-the-art imaging applications continue to embed richer video and graphics capabilities, the applications continue to demand much higher performance. The Pentium® III processor, with its associated Streaming SIMD Extensions, helps meet these new performance goals. The remainder of this section discusses how some of these new features help digital imaging algorithms enhance performance beyond those already achieved through MMX technology.

INTEGER SIMD EXTENSIONS

When implementing an SIMD imaging algorithm, one often encounters the need to rearrange data within an MMX™ register. The integer SIMD extensions include a shuffle instruction (`pshufw`) to enhance the performance of such frequently used operations. For example, an efficient SIMD implementation of alpha saturation would compare all of the R, G, and B components with the corresponding alpha value in parallel. To be able to do so, the alpha value itself needs to be replicated in a different MMX register as shown in Figure 6.

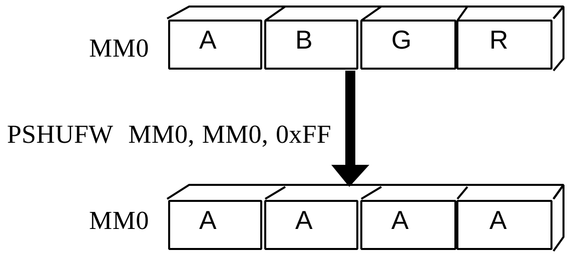


Figure 6: Broadcast alpha value

While this requires three instructions in MMX technology, the new instruction set would need just one.

Quite often, data-dependent branching has been an impediment in the process of mapping certain imaging algorithms to SIMD. For example, after computing an intermediate set of RGBA values, another set of computations might need to be executed if any of the R, G, B, and A values were below a certain threshold value. In a typical MMX implementation, the result of the condition check would be multiple mask patterns within an MMX register. However, extracting the required bits from these mask patterns into a register that can be used for addressing is usually very cumbersome. In certain cases, this might even negate the performance gains from an SIMD implementation of the algorithm. The new instruction `pmovmskb` addresses precisely this need. It extracts the required bits from the mask patterns in the MMX register and places them in a register that can be used for addressing.

Table look-up operations, such as the ones found in histogram-related algorithms, have always been critical to the performance of digital imaging. In such cases, each of the computed R, G, and B values is used as an index into its respective color look-up table. Such operations have been difficult to implement in MMX technology due to the fact that the computed RGBA values would be residing in an MMX register, which could not be used directly for addressing. Extracting each of them into the appropriate registers for addressing, fetching the contents from the table, and inserting them back into MMX registers was cumbersome and detrimental to performance. The integer SIMD extensions include a pair of instructions (`pinsrw/pextrw`) that helps enhance the performance of such algorithms.

In addition to the instructions mentioned above, the new integer SIMD extensions include several others that help enhance the performance of frequently used imaging algorithms. For example, the SIMD *unsigned multiply* instruction helps in the implementation of certain filter operations that were cumbersome using MMX technology. Likewise, the minimum/maximum instructions are useful during alpha saturation for bound checks, and the complete set of comparison operators facilitate all condition checking.

SIMD FLOATING-POINT

Current imaging implementations primarily involve fixed-point integer arithmetic. However, most state-of-the-art imaging applications are increasingly richer in their graphics capabilities and in their image quality. The algorithms therein should benefit significantly from the SIMD floating-point capability of the Streaming SIMD Extensions. Even if the underlying algorithms are implemented in floating-point, the enhanced floating-point performance helps yield near real-time response to typical user requests. Also, for intermediate results, the extra bits of available precision in a floating-point representation (relative to 16-bit fixed point) helps yield superior image quality. Moreover, implementing the algorithms in floating-point form reduces the need to deal with fixed-point arithmetic. This greatly boosts productivity by easing the task of code development, debugging, and maintenance. In imaging algorithms, the fundamental data object (RGBA pixel value) is of type integer. However, for the above-mentioned reasons, such as the need for extra precision and programming ease, several data transformations are implemented in floating-point. SIMD floating-point capability significantly enhances the performance of these implementations. The following bilinear interpolation example helps illustrate the usage of some of these SIMD floating-point instructions and also highlights some of the performance tradeoffs involved.

Bilinear Interpolation Example

The RGBA value of each pixel in the display image is calculated by a bilinear interpolation using RGBA values of four neighboring pixels in the source image (see Figure 7).

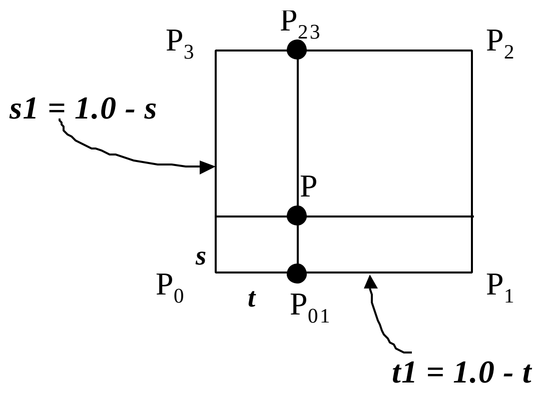


Figure 7: Bilinear interpolation

The R component of pixel P is calculated as follows:

$$R_{01} = t1 * R_0 + t * R_1$$

$$R_{23} = t * R_2 + t1 * R_3$$

$$R = s1 * R_{01} + s * R_{23}$$

From the above equations, it is evident that the bilinear interpolation steps involve a series of three linear interpolations. Each linear interpolation itself involves two multiplications and one addition for each value of R, G, B and A. Of course, when implemented in SIMD, all the four RGBA components can be computed in parallel. Initially, let us assume that we would like to perform these computations in floating-point since the SIMD floating-point capability might help us meet our performance goal. If so, as a first step, we will need to convert the RGBA pixel values from their typical byte representation to their float format. The steps involved in this are given in Figure 8. This conversion needs to be done for each of the four pixels in the source image. Note that both MMX™ technology and SIMD floating-point instructions are used in these steps, as are the MMX registers and the new Pentium® III processor registers. Overlapping the conversion steps for the four pixels better exploits available hardware as both the floating-point SIMD and the integer SIMD units will be operating in parallel.

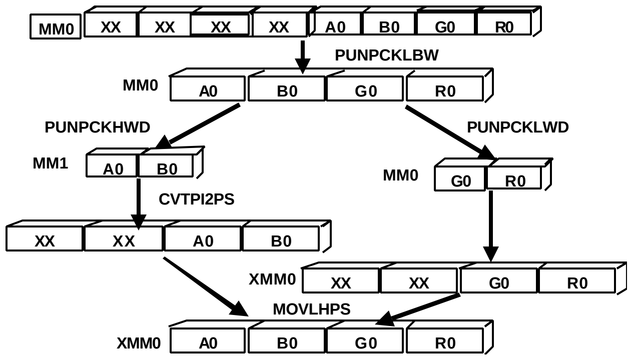


Figure 8: Packed byte to float conversion

Subsequent to this type conversion, the actual multiply-add step for each linear interpolation becomes relatively trivial (see Figure 9). Now, since the RGBA value of the result pixel is in float format, it needs to be converted back to integer type. The steps involved here are similar to those shown in Figure 8.

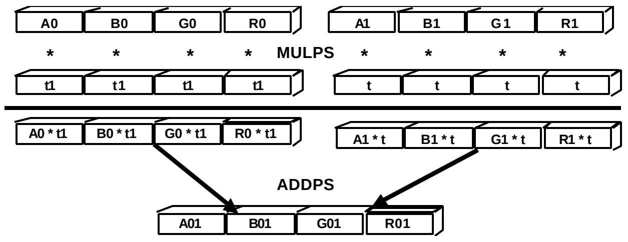


Figure 9: Linear interpolation

Analyzing the implementation indicates that the algorithm inherently required about nine basic instructions: two MULS and one ADD for each of the three linear interpolations. The decision to implement it using SIMD floating-point added about 29 additional instructions (six for each of the four source pixels from byte->float and five for the result display pixel from float->byte). However, the application would often perform several other floating-point operations such as lighting or other effects on the bilinearly interpolated pixel. In such cases, the byte<->float conversion time overhead can be amortized across all these additional floating-point operations. This helps yield enhanced performance using SIMD floating-point.

CACHE CONTROL INSTRUCTIONS

Given the typically large data sets in imaging, efficient cache utilization has a significant impact on performance. The Streaming SIMD Extensions have a few cache control instructions that help better utilize available hardware resources and minimize cache pollution. The different prefetch instructions help fetch data from memory to the different relevant levels in cache sufficiently in advance of their actual usage. For example, in a tile-based imaging architecture, while the execution units of the processor could be busy processing a certain tile's data, the memory subsystem could be busy *prefetching* the next tile's data. Likewise, when the final display pixel values have been computed, the streaming store instructions could be used to store them directly in memory without first fetching them into cache. This also helps minimize the potential for valuable data already in cache and needed for other computations from being evicted out of the cache.

To maximize the benefits from these cache control instructions, careful attention should be paid to issues such as identifying the data sets worth prefetching, the cache levels to prefetch to, and when to issue the prefetch.

VIDEO CODECS

Video codecs, such as MPEG and Digital Video (DV) codes, can obtain a performance increase by using streaming SIMD extensions. Table 1 gives examples of these increases.

	PSADBW	**PAVG**	**Prefetch, Streaming Stores**
Encode	Motion Estimation	Motion Estimation, Motion Compensation	Color Conversion, Motion Compensation
Decode		Motion Compensation	Color Conversion, Motion Compensation

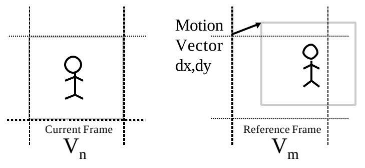
Table 1: Uses of Streaming SIMD Extensions for video codecs

MOTION ESTIMATION

Block matching is essential in motion estimation. Equation 1 is used to calculate the Sum of Absolute Differences (also referred to as the Sum of Absolute Distortions), which is the output of the block-matching function.

$$SAD = \overset{15}{\underset{i=0}{\text{å}}}\ \overset{15}{\underset{j=0}{\text{å}}} \left| Block_{ref}[i][j] - Block_{pred}[i][j] \right|$$

Figure 10 illustrates how motion estimation is accomplished.



$$\mathtt{SAD} = \overset{15}{\underset{i=0}{\text{å}}}\ \overset{15}{\underset{j=0}{\text{å}}} \mathtt{|V_n(x+i,y+j)-V_m(x+dx+i,y+dy+j)|}$$

Figure 10: Block matching

`dx` and `dy` are candidate motion vectors. Motion estimation is accomplished by performing multiple block matching operations, each with a different dx,dy. The motion vector with the minimum SAD value is the best motion vector.

Streaming SIMD Extensions provide a new instruction, `psadbw`, that speeds up block matching. The operation of this instruction is given in Figure 11.

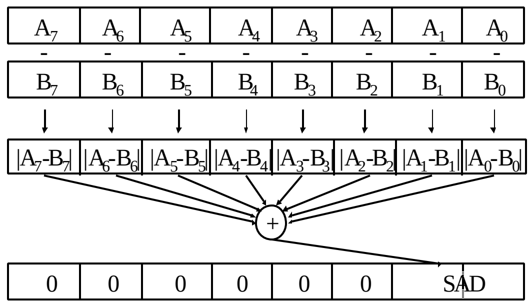

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
- - - - - - - -
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
$|A_7-B_7|$ $|A_6-B_6|$ $|A_5-B_5|$ $|A_4-B_4|$ $|A_3-B_3|$ $|A_2-B_2|$ $|A_1-B_1|$ $|A_0-B_0|$
+
0 0 0 0 0 0 SAD

Figure 11: PSADBW

Block matching is implemented by using the PSAD instruction as illustrated in Figure 12. The code to perform this operation is given in Table 2. This code has been observed to provide a performance increase of up to twice that obtained when using MMX™ technology.

Note that the nature of memory access of block matching will cause data cache line splits when the loads straddle 32-byte boundaries. This is due to the dx, dy changes of 1 (i.e., address variances are one byte at a time). The data loads are eight bytes at a time.

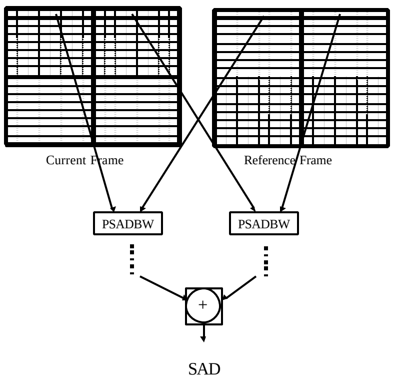


Figure 12: Block matching with PSADBW

```
psad_top:                  // 16 x 16 block
matching
     //  Do  PSAD  for  a  row,  accumulate
results
    movq mm1, [esi]
    movq mm2, [esi+8]
    psadbw mm1, [edi]
    psadbw mm2, [edi+8]

    // Increment pointers to next row
    add esi, eax
    add edi, eax

    // Accumulate diff in 2 accumulators
    paddw mm0, mm1
    paddw mm7, mm2
    dec  ecx         //  Do  all  16  rows  of
macroblock
    jg psad_top

    //  Add  partial  results  for  final  SAD
value
    paddw mm0, mm7
```

Table 2: Block matching

Hierarchical motion estimation is a popular technique used to reduce computational complexity and to provide potentially better motion vectors. Subsampling is illustrated in Figure 13.

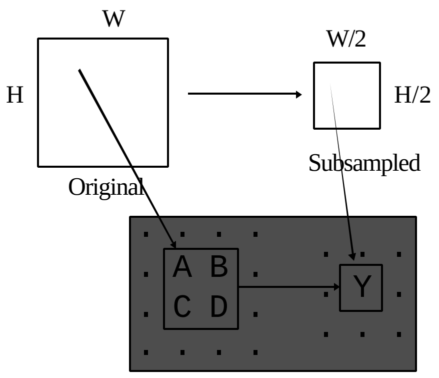


Figure 13: Subsampling for hierarchical motion estimation

Subsampling the original picture is sped up using the pavg instruction. Figure 14 shows the operation of pavgb.

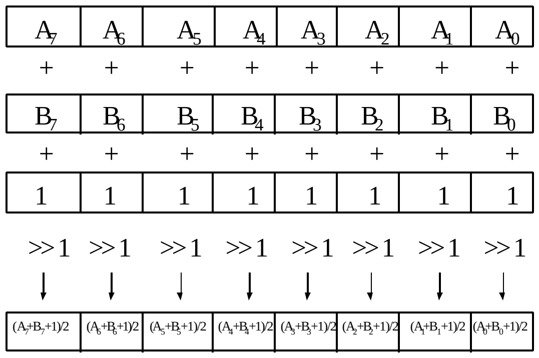

A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
+ + + + + + + +
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
+ + + + + + + +
1 1 1 1 1 1 1 1
>>1 >>1 >>1 >>1 >>1 >>1 >>1 >>1
$(A_7+B_7+1)/2$ $(A_6+B_6+1)/2$ $(A_5+B_5+1)/2$ $(A_4+B_4+1)/2$ $(A_3+B_3+1)/2$ $(A_2+B_2+1)/2$ $(A_1+B_1+1)/2$ $(A_0+B_0+1)/2$

Figure 14: PAVGB

The `pavgw` instruction is also provided in streaming SIMD extensions. It works like the pavgb instruction, but performs the averaging on four 16-bit values.

It is important to note that the additions are performed with an additional bit for accuracy (9 bits for `pavgb`, and 17 bits for `pavgw`). This avoids overflow errors. Once the average is performed (after the divide-by-2), the width of the result is the same as the input (8 or 16 bits).

While the `pavg` instructions operate on two values at a time, it is possible to use three `pavg` instructions to approximate 4-value averaging. The line below illustrates this in pseudo-code:

```
Y = pavg(pavg(A,B),pavg(C,D)-1)
```

This value is close to (A + B + C + D + 2)/4 which is the typical calculation used to perform subsampling. However, for the approximation, 87.5% of values match exactly, and 12.5% of the values are off by one least significant bit (LSbit). The maximum error is one LSbit. This error is often acceptable for performing motion estimation.

MOTION COMPENSATION

Motion compensation (MC) is used in both video decoders and encoders. Decoders perform inverse motion compensation (iMC), and encoders perform both MC and iMC. The accuracy of these calculations is important, especially for encoders, since their local decoder should track the operation of a high-quality decoder. In MC, bi-directional B-frames can require interpolation of two values. The MPEG standard specifies this as

$$Y = (A + B + 1)/2$$

The `pavg` instructions provide exactly this calculation.

Streaming SIMD extensions also provide prefetch and streaming store instructions. Since MC is often memory bound, prefetch operations can speed up MC. `prefetchnta` and `prefetcht0` have both been observed to provide a speedup. Which one offers the best improvement is dependent on how the decoder is implemented. For decoders that are writing the decoded picture to a graphics card memory, `movntq` (move non-temporal quad-word) can offer a benefit by not polluting the caches with data that will never again be needed by the decoder.

DISCRETE COSINE TRANSFORM

The Discrete Cosine Transform (DCT) and inverse Discrete Cosine Transform (iDCT) are used in video codecs. Decoders use the iDCT, and encoders use the DCT and usually the iDCT (if they have a local decoder). It is possible to gain a speedup from streaming SIMD extensions; however, the speedup is application-dependent. The SIMD floating-point instructions can be used to calculate a very accurate DCT/iDCT. However, it is possible to be IEEE 1180-1990 [3] compliant using SIMD integer instructions, such as those found in MMX™ technology. In general, for consumer electronics versions, SIMD integer implementations are sufficiently accurate and are the fastest. For professional or reference codecs, SIMD floating-point may be the preferred choice. To ease the burden on codec developers, both of these implementations are available in Intel's Image Processing Library.

VARIABLE LENGTH ENCODE

Encoders must create a bit stream based on the values after the Discrete Cosine Transform and quantization. This is called the Variable Length Encode (VLE). Often, especially in the case of B-frames, there are many zero values that must be detected and "skipped over." To aid in the processing of these values, the `pmovmskb` instruction can be used to evaluate eight values. Table 3 illustrates how `pmovmskb` can be used for this.

```
pxor      mm7,mm7    // zero mm7
movq      mm0,[esi]  // get eight Q values
pcmpeqb   mm0,mm7    // find zeros
pmovmskb  eax,mm0    // 8 flags into eax
```

Table 3: Variable length encode

If `eax` holds 0xff, then all eight values are zero.

COLOR CONVERSION

Color conversion is used by both encoders and decoders. Often encoders receive data in a format other than what they can directly encode (wrong chromenance space, interleaved vs. planar data, etc.). Decoders sometimes have to write the decoded picture to a graphics card's memory in a color space other than the color space that naturally is produced from the decode; this also requires a color conversion.

For encoders, color conversion is typically a memory-bound operation. It loads picture data from main memory (i.e., DMA'ed in from a video capture card), performs some (typically simple) calculation, and writes the data back out to memory. `prefetchnta` can speed up color conversion by bypassing the L2 cache on the load. The non-temporal prefetch is often the best prefetch for color conversion since the input will not be needed again by the codec. The store can then be performed using a normal store (e.g., `movq`) so the picture resides in L2 cache after the color conversion.

CONCLUSION

The order in which data is stored in memory, and how it is moved to and from the processor and its caches, can have a significant impact on the performance of an application. While the hybrid data order is technically the best overall match for SIMD, if an application must use the conventional array of structures order, it is generally best to transpose the data into the structure of arrays order in the SIMD registers for processing. The prefetch instructions can often reduce memory latency or optimize memory bandwidth. When processing large blocks of data, splitting the data into subsets that fit the Pentium® III processor caches can avoid unnecessary memory overhead.

Managed use of memory and the 4-wide SIMD registers provide big benefits in the 3D transform. The results of the transform code tested showed an improvement of 3.0x to 3.7x for Pentium III processor-optimized assembly code over standard 'C' code. 3D Lighting also showed significant gains (~4x) through the use of approximation and branch-elimination instructions.

The integer extensions ease implementation of typical imaging algorithms in SIMD while also extending their performance beyond those achieved through MMX™ technology. Likewise, the floating point SIMD, when used appropriately, enhances the accuracy and performance of algorithms with floating-point implementations. Moreover, it eases code development and validation by reducing the need to deal with fixed point arithmetic. Several of these techniques have been successfully applied in the high-performance Image Processing Library which is part of the Intel® Performance Library Suite [4].

Streaming SIMD Extensions can be used to greatly speed up functions commonly found in video codecs. These functions include motion estimation, motion compensation, variable length encode, and color conversion. The new `psadbw`, `pavgb`, and `pavgw` instructions, as well as prefetch and streaming stores, are paticularly useful for video codecs. Speedups of 2x have been observed for motion estimation, and speedups of 1.3x have been observed for entire encoder applications.

ACKNOWLEDGMENTS

The tuning concepts contained in this paper include refinements based on the optimization work of Intel engineers and organizations from groups such as the Microprocessor Labs, the Folsom Design Center, and Developer Relations and Engineering.

REFERENCES

[1] J. Wolf "Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions using the VTune™ Performance Enhancement Environment," Intel Technology Journal, Q2, 1999.

[2] A. Watt, *3D Computer Graphics 2nd Edition*, Addison-Wesley Publishers Ltd., Essex, England.

[3] IEEE Circuits and Systems Society, IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform, *IEEE Std. 1180-1990*.

[4] http://developer.intel.com/vtune/

AUTHORS' BIOGRAPHIES

James Abel focuses on software applications for future Intel® processors. In his ten years at Intel, he has held several software and hardware positions, including the development of Intel's software Dolby* Digital decoder, embedded microcontroller design, and Design Automation. James obtained a B.S. degree in engineering from Bradley University in Peoria, Illinois, in 1983 and an M.S. degree in computer science from Arizona State University in 1991. His e-mail is james.c.abel@intel.com .

Kumar Balasubramanian works with software developers to help their applications take advantage of Intel's new processor capabilities. He managed the integration of the Streaming SIMD Extensions into several business applications. Kumar has been with Intel for seven years and has held leadership roles in Intel's CAD engineering organization and with Intel Architecture Labs to develop some of the first applications using MMX™ technology. He has an M.S. degree in computer engineering from Dartmouth College. His e-mail is kumar.balasubramanian@intel.com.

Mike Bargeron obtained a B.S. degree in electrical engineering from Brigham Young University. He started with Intel's Software Performance Lab in 1997. Since coming to Intel, Mike has been involved in performance tuning 2D and 3D graphics applications for the PC. Specifically, he has worked with MPEG motion video as well as several 3D game titles. His e-mail is michael.l.bargeron@intel.com.

Tom Craver works with 3D graphics IHVs to help them optimize their driver software on Intel's latest processors. Previously he developed and validated driver and user interface software for cable modems and for Intel's DVI multimedia technology. Prior to joining Intel, Tom was a member of the technical staff at the David Sarnoff Research Center in Princeton, NJ, and before that, he was with AT&T's Bell Laboratories. Tom holds B.S. degrees in physics and computer science from the University of Illinois. He also has a M.S. degree from Purdue University. His e-mail is tom.r.craver@intel.com.

Mike Phlipot works with desktop software developers to integrate Intel's newest processor capabilities into their applications. Most recently he has been helping 3D game developers take advantage of the Streaming SIMD Extensions. In his ten years with Intel, he has held various engineering and management positions in technologies that include digital video compression and cable modems. Mike has a B.S. degree in mechanical engineering from General Motors Institute and a M.S. degree in computer engineering from the University of Michigan. His e-mail is mike.p.phlipot@intel.com.

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

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

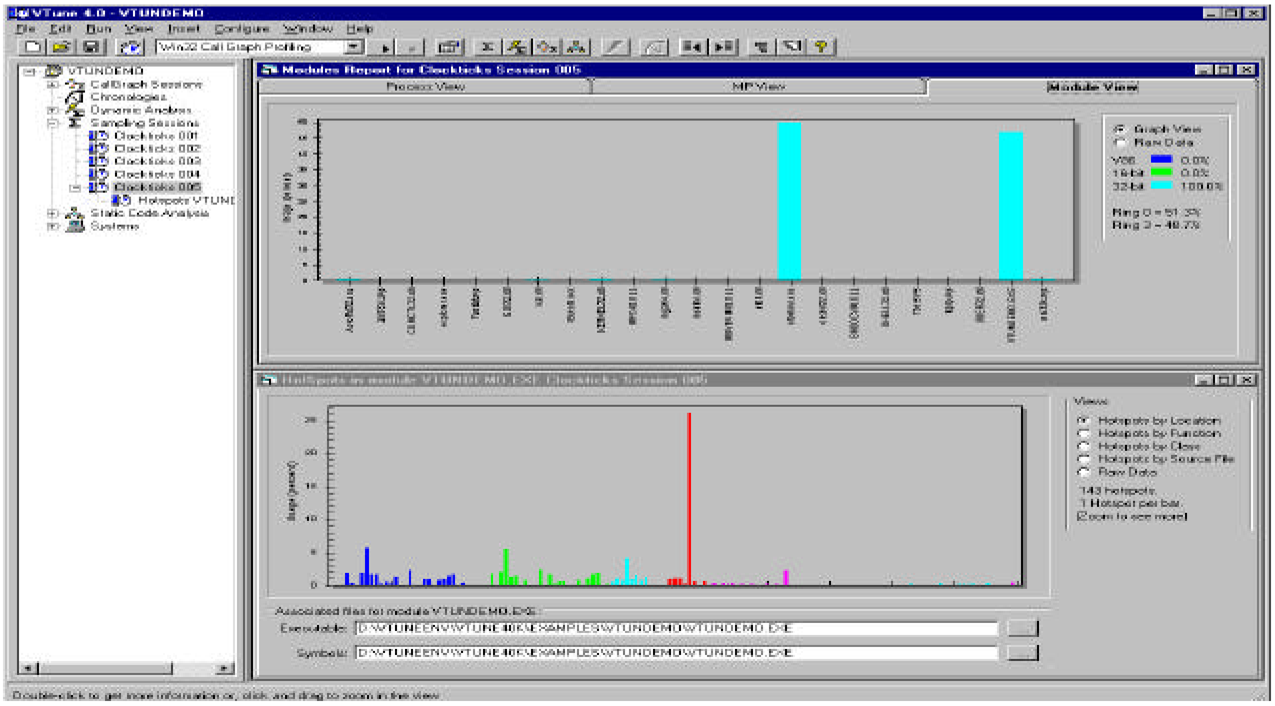
Event-based sampling is the most commonly used method for analyzing application performance with the VTune Analyzer. It allows the user to select any of a number of different events implemented in the processor. These events allow the user to hone in on specific aspects of an application's use of the 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%

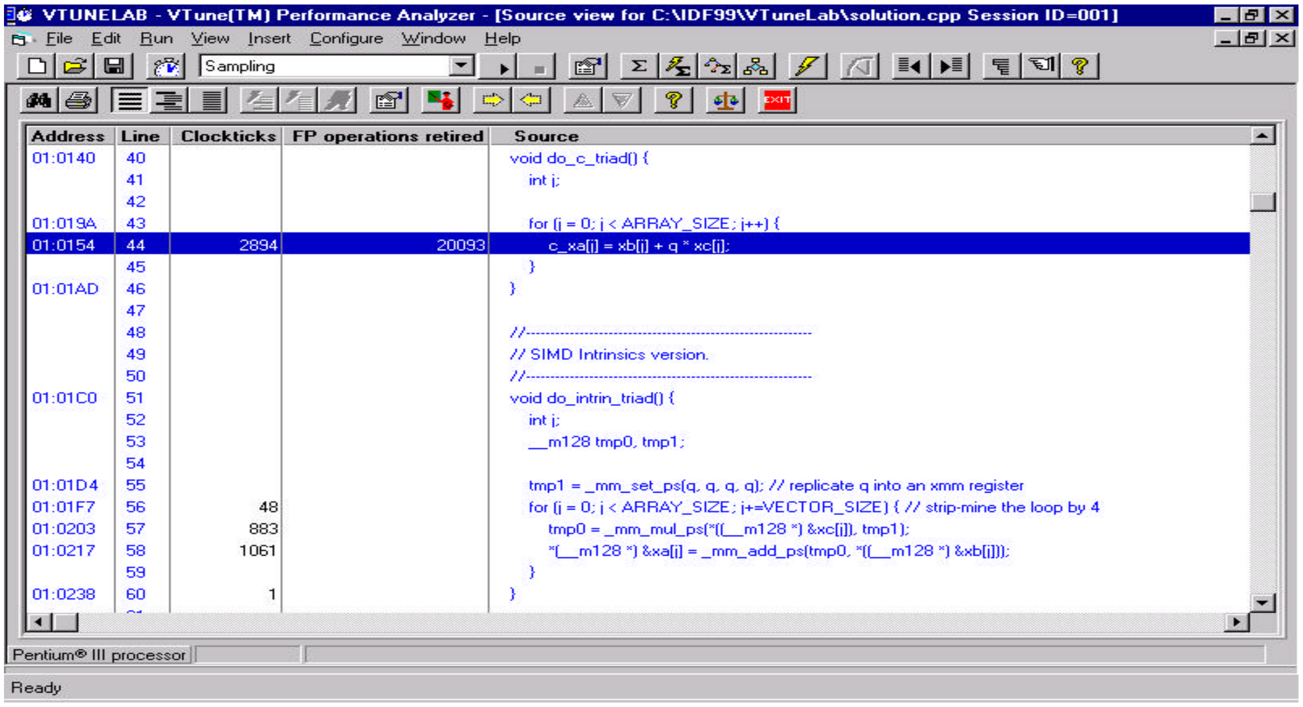
Ring 0 = 51.3%
Ring 3 = 48.7%

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
Pentium® III processor
Ready

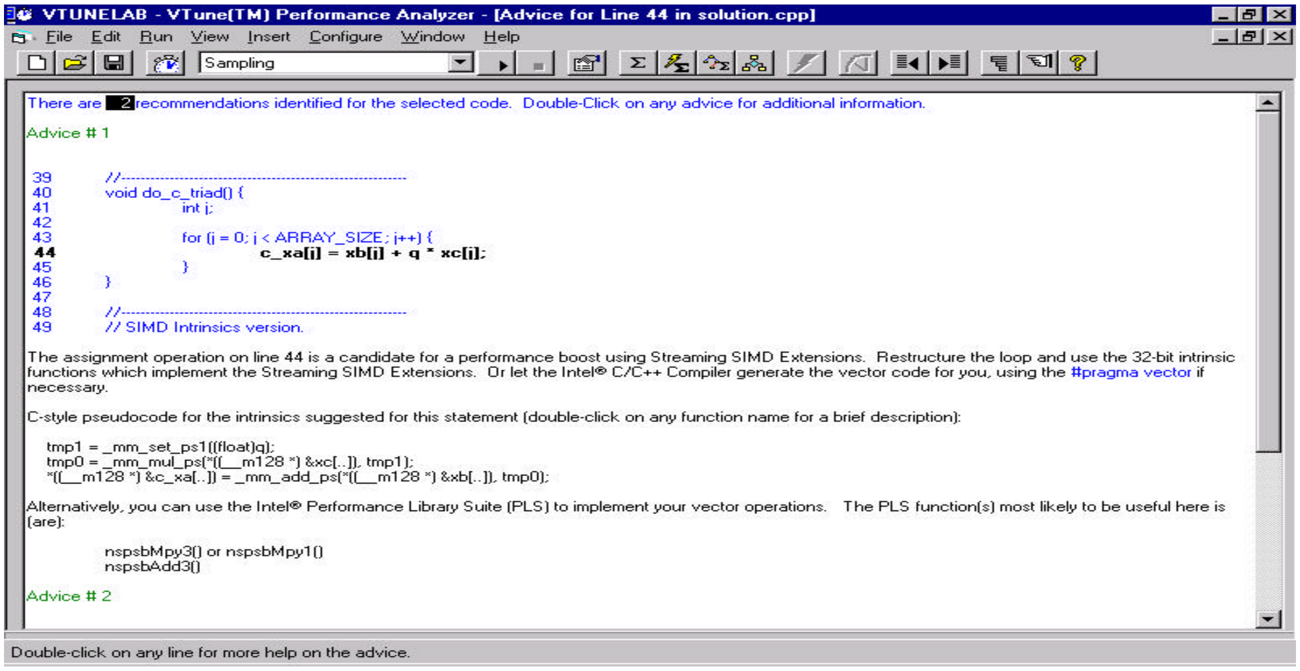
Address	Line	Clockticks	FP operations retired	Source
01:0140	40			void do_c_triad() {
	41			int j;
	42			
01:019A	43			for (j = 0; j < ARRAY_SIZE; j++) {
01:0154	44	2894	20093	c_xa[j] = xb[j] + q * xc[j];
	45			}
01:01AD	46			}
	47			
	48			//--
	49			// SIMD Intrinsics version.
	50			//--
01:01C0	51			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()
```

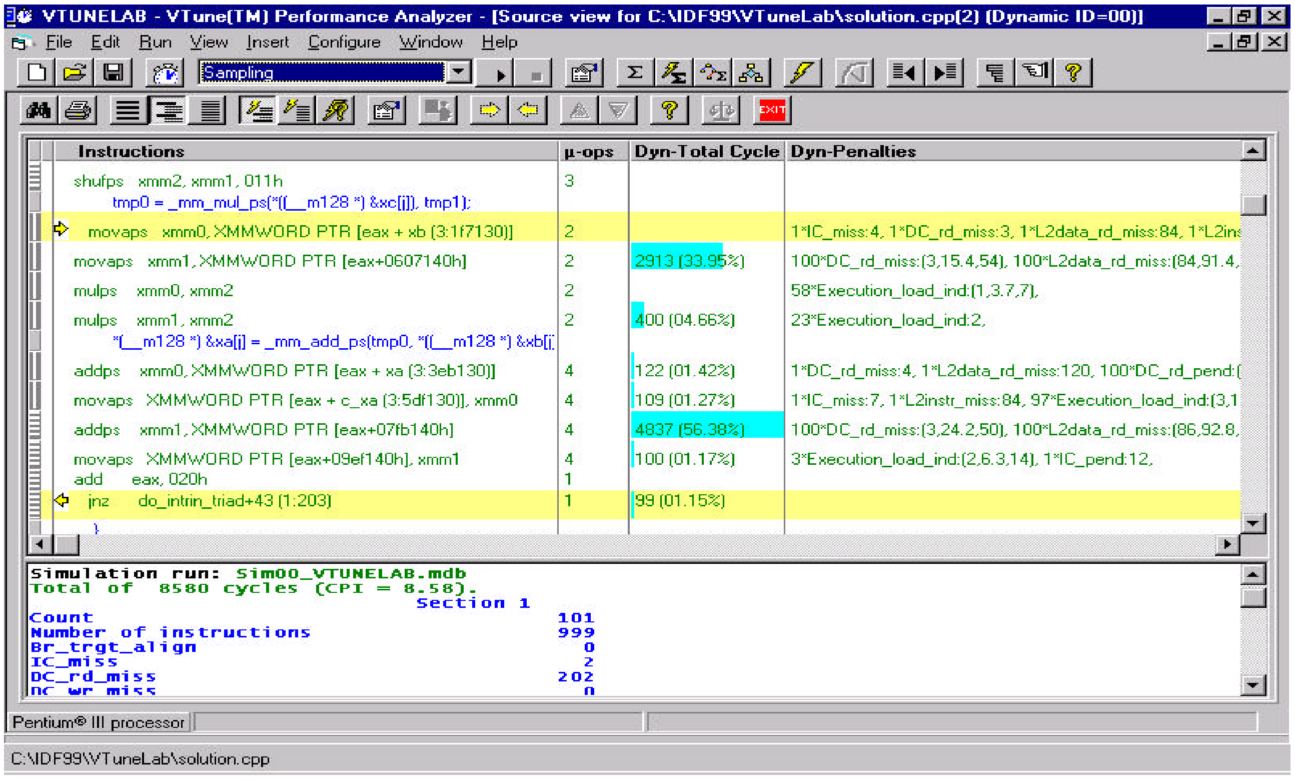
Advice # 2

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^{TM} 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.