



# Utilizing SIMD Instructions on the INtime RTOS with the Intel IPP Library

Paul Fischer, TenAsys Corporation and  
Ying Song, Intel Corporation

March, 2009

## Abstract

Applying DSPs (digital signal processors) to real-time embedded systems can result in significant performance improvements, especially where complex mathematical algorithms are involved. Algorithms that perform repetitive operations on large data sets are particularly well suited to the application of DSPs. The data to be transformed can be collected and processed in bulk or it can be continuously streamed (sampled) data that must be processed in a finite period of time to generate a real-time control signal. Motion control, image processing, speech recognition, and computer vision applications are some of the many applications that can benefit from the application of DSP hardware.

Unfortunately, there are significant drawbacks to the application of DSP hardware: hardware cost, software complexity, and the need for a time-deterministic operating system. All three can be tamed by the application of a SIMD-optimized library on a multi-core processor with an RTOS that provides control over determinism and priority of tasks.

This whitepaper introduces the use of SIMD instructions, via the Intel® IPP library, on the INtime RTOS for Windows. Combining the raw performance of the SIMD instruction set with the hard real-time determinism of the INtime RTOS, on a multi-core processor alongside the general-purpose programming environment of Windows, gives real-time embedded system developers the means to lower hardware costs by the elimination of DSP hardware or enhance applications by incorporating DSP functionality without significant software complexity.

## SIMD Instructions

A DSP is poorly suited to general-purpose tasks, it is designed to quickly execute basic mathematical operations (add, subtract, multiply and divide) on multiple operands. The DSP repertoire includes a set of very fast multiply and accumulate (MAC) instructions to address matrix math evaluations that appear frequently in convolution, dot product, and other multi-operand math operations. Because it is difficult to write general-purpose applications on a DSP they are used as co-processors (e.g., the GPU in a video card) alongside a general-purpose CPU, where the DSP is provided with a data set and operating instructions by the general-purpose processor. This arrangement results in a level of hardware and software complexity that prevents their application in many systems that could benefit from their use.

The MAC instructions that comprise much of the code in a DSP application have an analog in many modern general-purpose processors referred to as Single Instruction Multiple Data instructions (SIMD). Like a DSP, these instructions perform mathematical operations very efficiently on large arrays of data. Unlike a DSP, when using a general-purpose processor that supports SIMD it is easier to integrate the overall application algorithms with the complex mathematical algorithms, because both run on the same processor and can, therefore, be part of a unified logical execution stream.

For example, an algorithm that changes image brightness by adding (or subtracting) a constant value to each pixel of that image must read the RGB values from memory, add (or subtract) the offset, and write the new pixel values back to memory. When using a DSP co-processor that image data must be “packaged” for the DSP (placed in an area accessible to the DSP), the DSP must be signaled to execute the transformation algorithm, and then the data must be “returned” to the general-purpose processor. Using a general-purpose processor with SIMD instructions simplifies this process of packaging, signaling, and returning the data set.

Go to [http://en.wikipedia.org/wiki/Digital\\_signal\\_processing](http://en.wikipedia.org/wiki/Digital_signal_processing) for an overview of some of the techniques used in digital signal processing applications.

## Intel MMX and SSE Instructions

The MMX and SSE (Streaming SIMD Extensions) instructions found on Intel Architecture processors are the most widely available SIMD instructions available today. MMX and SSE instructions have been a part of every Intel Architecture (IA) processor produced since the Pentium III.

These MMX and SSE instructions operate on integer and floating point operands in a manner that is analogous to a DSP. In the latest IA processors the number of available SIMD instructions, in all their variations, count over 200! These SIMD instructions implement a variety of packed arithmetic, move, compare, conversion, and logical operations, all designed to address the needs of a variety of digital signal processing algorithms.

See [http://en.wikipedia.org/wiki/Streaming\\_SIMD\\_Extensions](http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions) for an overview of the Intel Architecture SIMD instructions and browse the Intel Software Knowledgebase at <http://software.intel.com/en-us/articles/all> for detailed articles on the subject.

The SSE instructions continue to be enhanced and evolve with successive generations of Intel processor architectures. For example, the latest addition to the SSE instruction set,

SSE4, is designed to benefit media applications and accelerate string and text processing for database applications or algorithms that perform pattern matching and search operations.

See <http://www.intel.com/technology/architecture-silicon/sse4-instructions/> for more information on the latest SSE instructions.

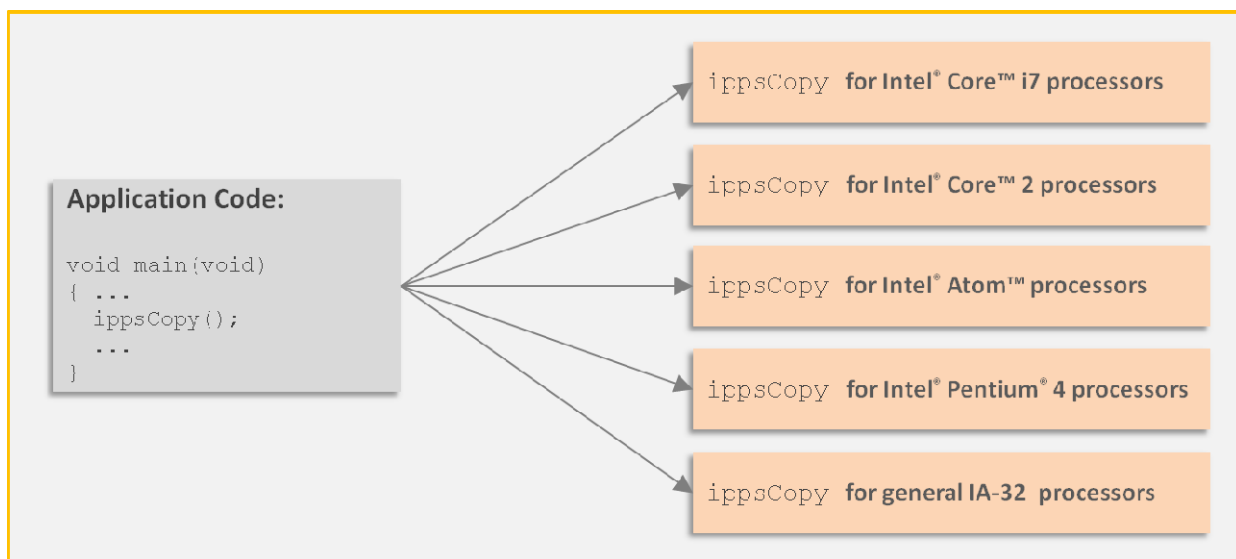
Given the large number of SIMD instructions, and the fact that each new processor generation introduces new instructions not supported in previous generations, the idea of adopting SIMD instructions to achieve DSP-equivalent functionality might seem overwhelming. Fortunately, there is another way to take advantage of SIMD instructions without having to understand their details, or even worry about which specific processor generation(s) your application will use.

## The Intel IPP Library

The Intel® Integrated Performance Primitives (the IPP library) is a collection of functions optimized for Intel Architecture SIMD instructions. The library takes full advantage of the advanced MMX and SSE instructions found on x86 processors without requiring that you become an expert with the SIMD instruction set.

The IPP library also includes a “dispatch” feature (see the `ippStaticInit()` function) that determines which processor architecture your application is running on and configures the library, at run-time, to use the optimum set of SIMD instructions for that generation of processor. In other words, when your application runs on an older “Pentium with MMX” processor only MMX instructions are used by the library, but when it runs on a Core 2 Duo the MMX and SSE instructions through SSE4.1 are utilized.

This dispatching feature enables optimized multi-platform development with a single application image. You do not need to build multiple processor-specific executables. A single API for all processor generations removes the burden of platform optimization.



Automatic CPU Detection and Dispatching

The most current list of processors supported by the IPP library can be found here: <http://www.intel.com/software/products/ipp>

Obviously, there is a small penalty associated with the overhead of the dispatching feature, in the form of a jump table into the processor-specific implementation of each function (see the *Static Linking (with Dispatching)* section in Chapter 5 of the *Intel IPP User's Guide* for more information). If you need to eke out every last bit of performance from the library you also have the option of linking with a processor architecture-specific library to eliminate the dispatch overhead.

### Function Groups (Application Domains)

The IPP library is divided into a number of functional groups, or *application domains*. These domains encompass a broad range of digital signal processing functions for handling tasks like matrix arithmetic, digital filters, audio, image, and video encoding and decoding, and string processing. The list of “domains” at the time of this writing is summarized in the following table:

Video Decode/Encode	Audio Decode/Encode	JPEG/JPEG2000
Data Compression	Cryptography	Speech Coding
Speech Recognition	Image Processing	Image Color Conversion
Computer Vision	Signal Processing	Vector/Matrix Mathematics
String Processing	Data Integrity	Ray Tracing/Rendering

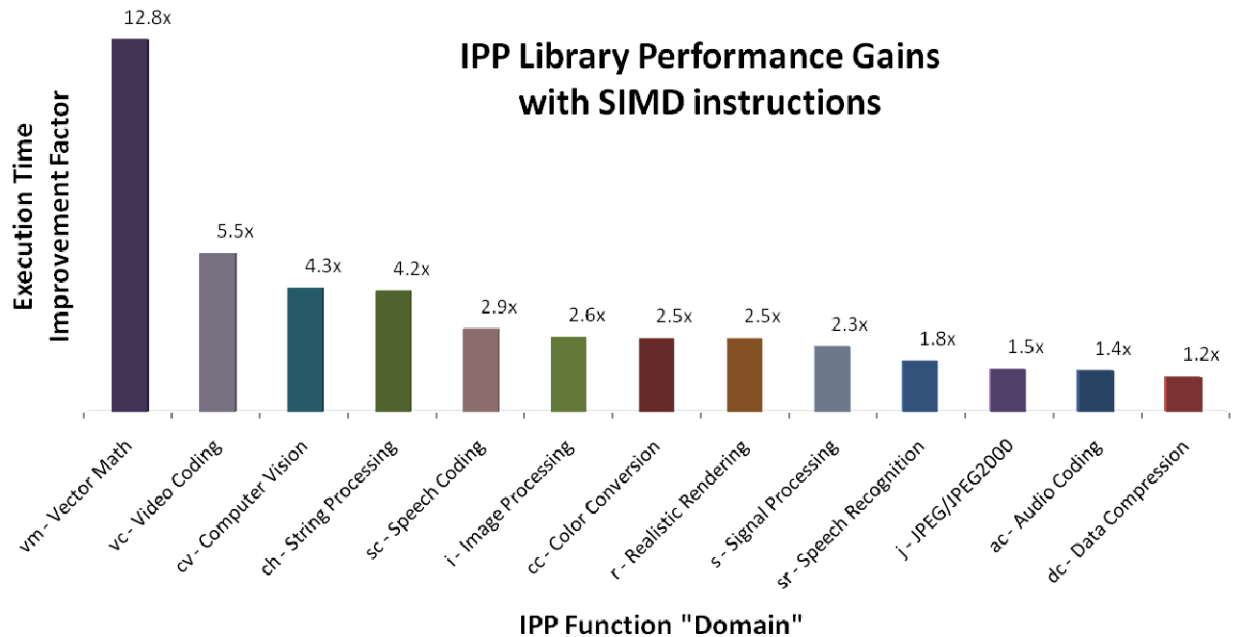
The most current list of domains in the IPP library can be found here:  
<http://www.intel.com/software/products/ipp>

For each application domain, the IPP library provides function *primitives* that implement key algorithm and performance-sensitive operations. These *primitives* perform single, specialized operations with minimal overhead. Using the primitives you have the flexibility to maintain and control the organization of your code and data structures, the numeric precision, and even how to handle errors. The library also provides variants of functions optimized to specific data operand size and precision requirements.

A more detailed description of the IPP application domains can be found in the whitepaper “Boosting Application Performance using Intel Performance Libraries”:  
<http://www.intel.com/support/performancetools/libraries/ipp/sb/cs-015088.htm>

## SIMD Performance Gains

The following chart illustrates the raw performance improvements possible by replacing expertly constructed C functions based on standard 32-bit x86 instructions with equivalent IPP functions, for many of the application domains covered in the IPP library.



results shown are for the Intel IPP Performance Tests for Windows on a 2.50 GHz Core 2 Duo T9300

The graph represents simple aggregate performance gains due to SIMD instructions measured with version 6.00.044 of the Intel IPP library.

This data was generated using the *Intel IPP Performance Tests* (aka *perfsys*) provided with the IPP library on a 32-bit Lenovo T61p Windows Vista laptop containing an Intel Core 2 Duo T9300 CPU (6 MB cache, 2.50 GHz clock, 800 MHz FSB). There is one *perfsys* test for each application domain. Each *perfsys* test calls every IPP primitive within its application domain, measures the time required to execute each primitive, and writes those results to a text file.

In order to measure only the effect of SIMD instructions, the system on which the tests were performed was configured to operate as a single-core 32-bit CPU (one of the two cores was disabled), eliminating optimizations resulting from scheduling threads over multiple processor cores. Variations due to CPU clock speeds, cache size, RAM size or type, and FSB performance were eliminated because the same machine was used for all tests.

## Processor Architectures

There are five general 32-bit processor architectures supported by the tested version of the IPP library. They are shown in the table below in order of increasing SIMD functionality. The base “px” architecture does not use SIMD instructions, only those instructions available on a standard 32-bit Intel Architecture processor. The other four processor architectures in the table add increasing levels of SIMD instruction set support.

<b>px</b>	C-optimized for all IA-32 processors
<b>w7</b>	Optimized for Pentium 4 processors
<b>t7</b>	Optimized for Pentium 4 processors with SSE3 instructions
<b>v8</b>	Optimized for 32-bit applications on Intel® Core™ 2 and Intel® Xeon® 5100
<b>p8</b>	Optimized for 32-bit applications on 45nm Intel® Core™2 Duo processors

Each `perfsys` IPP domain test was run for each of the 32-bit processor architectures shown in the table. The T9300 CPU used in the test machine is a Penryn family processor that supports SIMD instructions through SSE4.1 (the maximum level supported by the tested version of the library).

The *IPP Library Performance Gain* chart, at the beginning of this section, compares the relative increase in performance between the base “px” architecture and the SSE4.1 “p8” architecture. In other words, the chart shows how much faster the application domain primitives executed when they were allowed to use SIMD instructions compared to functions using standard x86 32-bit instructions.

## Real-World Performance Gains

The chart in the previous section is a somewhat artificial and conservative measurement of performance gains; while it clearly shows that using SIMD instructions results in faster execution, it does not measure real-world gains. The performance gains shown on the chart should not be interpreted as an upper limit of what is possible when using SIMD instructions. *In fact your gains may be much better than those shown on the chart!*

The `perfsys` tests measure an aggregate of execution times for each of the primitives in a domain, for multiple parametric variations of each function. This results in hundreds or even thousands of individual measurements. But does this measure how much of a performance boost you will get from using SIMD instructions, or the IPP library? Not really.

Your actual performance gains depend heavily on the mix of primitives you use, and how often they are called by your application. For example, if you compare individual `perfsys` line item results for primitives in the JPEG/JPEG2000 application domain test, which has an aggregate gain of 1.5x over the base level (a 50% gain), you will find that more than two-thirds of the individual primitives achieve performance gains greater than that 1.5x base level. In fact, more than half of the primitives see a 3x improvement, with the highest gaining primitive measuring an impressive 45x improvement in execution speed!

## JPEG Compression and Decompression

To measure real-world performance improvements we need a real-world application. The IPP samples include a test application that applies the Independent JPEG Group (IJG) library to compress and decompress image files. This test application was used to evaluate real-world performance improvements resulting from the use of SIMD instructions.

Details regarding the IJG library, and the full source to the library, can be found on the Independent JPEG Group web site at: <http://www.iijg.org>.

The IJG library is a free and portable JPEG compression/decompression library written in the C programming language. It is a well known JPEG library that continues to be widely used to this day.

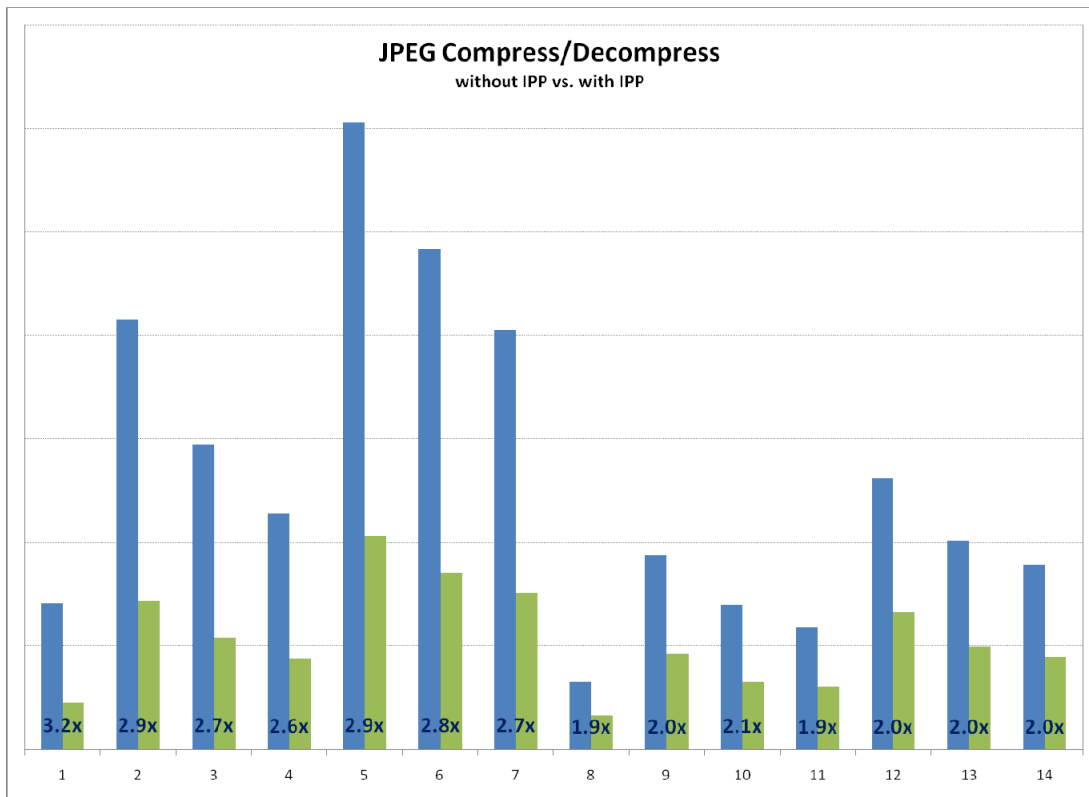
Several parts of the IJG library were substituted with Intel IPP JPEG primitives without loss of functionality. Substitutions were made so the test application could be compiled and linked to use either the IJG library as originally written (version 6b of 27 March 1998) or with IPP primitives.

Exact details about how the IPP library primitives were adapted to the IJG library, and which primitives were used, can be found in the documentation that accompanies the IPP sample projects library.

In addition to the modifications made by Intel to the IJG library, the test program that utilizes the IJG library was adapted so it could be compiled to execute as either a Windows console application or an INtime console application. The file input and output buffers were also increased in size, using the ANSI C `setvbuf()` function, to eliminate file I/O overhead (disk delays) in the measurements.

## JPEG Real-World Results

The green and blue bars in the following chart represent the amount of time required to perform a set of JPEG encode and decode functions on a collection of images containing random pixels (taller bars represent longer times). Blue bars represent the execution time using the original IJG library; green bars are the execution time for the IJG library modified to use IPP primitives. The numbers at the bottom of each pair of bars (e.g., 3.2x for test 1) quantify the real-world performance increase resulting from the inclusion of the IPP library and SIMD instructions in the IJG library.



One test cycle consists of seven encode operations (numbered 1-7 on the x-axis) and seven decode operations (numbered 8-14). One bar is an average of six successive tests. Images contain random pixel data and are 1024 x 768 pixels in size. Images for each of the seven operations vary by the number of color channels or the encoding block size used. The same set of image data was used for each run.

Clearly the simpler IPP `perfsys` tests, which aggregate all the primitives in an application domain, do not tell the whole story. Depending on what an application does, how frequently it does it, and what subset of the IPP primitives from a given domain are used, the savings can be far greater than those given by the `perfsys` tests. If we went solely by the `perfsys` results we would expect to see a 1.5x increase in performance. However, as the chart above shows, the performance increase for this specific application varies between 1.9x and 3.2x.

## Real-Time Determinism Gains

The Intel IPP library is designed and sold for use with the Microsoft® Windows®, Apple® Mac OS® X, and Linux operating systems. With no variants of the library targeted for use with an RTOS, how can developers take advantage of this tool to easily apply SIMD instructions to real-time applications?

### Using the IPP Library with the INtime RTOS

The INtime RTOS is able to use the IPP library in part because it utilizes the Microsoft® Visual Studio® integrated development environment (IDE) as its build and debug platform. The entire INtime software development kit (SDK) integrates with the Visual Studio IDE, allowing developers to use the popular Visual Studio compiler, linker, and debugger to create real-time applications for the INtime RTOS. This means that INtime applications use the same calling conventions and library and object code file formats as those used by Windows 32-bit applications.

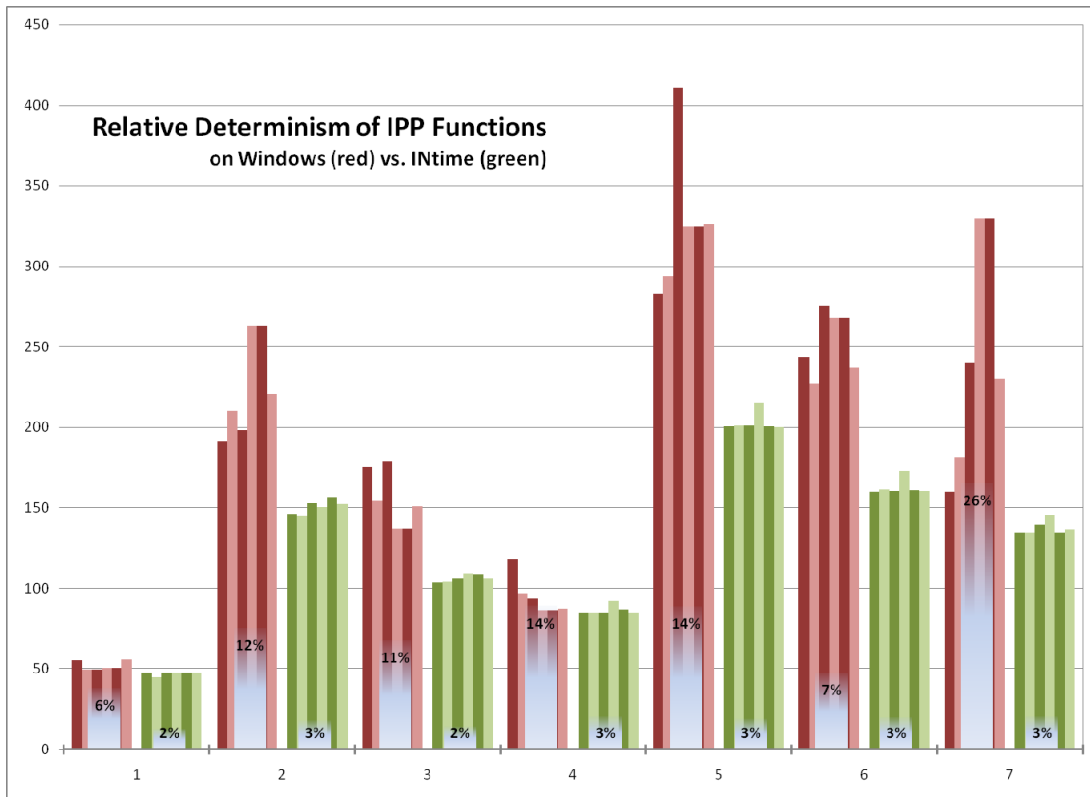
Because the Intel IPP library is OS-agnostic, it does not require OS-specific features to be used, real-time developers can link the 32-bit Windows static IPP library file with their INtime real-time application and directly call the SIMD optimized IPP library primitives from within their real-time application. This is a far easier and more time-efficient means to apply the Intel Architecture SIMD instructions to a real-time platform than writing your own library of functions in assembly language.

Combining the IPP library with INtime gives you the raw performance gains of the SIMD instruction set and the real-time determinism of an RTOS. In some applications displacing a DSP requires both performance and determinism.

### Busy versus Idle

The performance charts in the previous sections of this paper consist of data collected on an idle system. That is, only the test application was running and there were no other significant activities consuming CPU cycles on the test machine when the measurements were made. An idle machine, while representative of ideal conditions, is not necessarily realistic for measuring the real-world performance that will be encountered by an application.

To measure real-world variations in execution time of the JPEG test application our test machine was “loaded down” by multiple simultaneous system-intensive activities. These consisted of: a string search through all files on the local hard drive, viewing a small video in a continuous loop with a third-party video player, and playing MP3 files in Windows Media Player while it generated a “visualization” display representative of the music being played. The test results are shown in the chart below.



The measurements in this chart are the same seven encoding variants (1-7) from the JPEG test in the previous charts, using the same JPEG test application and image files; the seven decoding variants (8-14) have been omitted to keep the chart readable. Each red and green cluster contains six bars, one for each of six successive test runs. The gray bars that overlay each cluster are the standard deviation of the bars in that cluster expressed as a percent of the mean execution time of that cluster.

Because Windows and INtime share a single machine, activity on the Windows OS represents potential interference for the JPEG encode application. However, because INtime processes always have precedence over Windows processes, the JPEG encode application always operates in a fixed time when executing as an INtime process. This is clearly illustrated by the INtime measurements (green bars), which are consistently faster and more consistent in their execution times than the Windows measurements (red bars), when the system is heavily loaded.

In this case, running the JPEG test application as an INtime process (a real-time application) on a busy system results in variations measured as 3% or less of the mean time to execute. Whereas the same application running as a Windows process exhibits variations in run times between 6% and 26%. On an idle machine the red and green bars (the Windows and INtime runtimes) are nearly identical; the difference between the two operating systems is guaranteed determinism, not performance.

## Static Linking

The INtime RTOS does not support Windows DLL files (dynamic link libraries), so real-time applications must be linked with the static edition of the IPP library. The static edition of the library does not limit the range of functions that can be applied to your application, it simply means your application's executable file directly includes that subset of the library you need, and is loaded into memory along with your application at run time, rather than being loaded dynamically (like a DLL).

To insure optimal performance with the static IPP libraries, one additional initialization step is required. Before any IPP primitives are called the `ippStaticInit()` function must be called to initialize the library so it will use the optimum set of SIMD instructions based on the specific processor architecture that is identified at run time. Without this call the performance of your application will suffer because only the general IA-32 ("px" base line) version of the IPP functions will be used. This is the mechanism by which your application dynamically adapts to the specific processor architecture at run time without requiring multiple executables or burdening your code with the overhead needed to handle those variations.

## Where to go for Additional Information

TenAsys is not an authorized reseller for the Intel IPP library. Please [visit the Intel IPP page](#) or contact your authorized Intel reseller for pricing and availability of the Intel IPP library. You need the standard *Intel® Integrated Performance Primitives for Windows* product in order to integrate the IPP library with your INtime real-time applications.

Microsoft Visual Studio is all that is required to create IPP applications for the INtime RTOS. All the standard INtime wizards and debug tools are available for development of your application.

INtime applications are limited to the IPP static linkage models. Run-time dispatch (automatic detection of the underling CPU) is supported by the static linkage model, insuring that optimum performance of the IPP library is always achieved, regardless of the underlying processor architecture.

**For more details regarding Intel SIMD instructions:**

[Streaming SIMD Extensions](#)

[Intel® Streaming SIMD Extensions 4 \(SSE4\) Instruction Set Innovation](#)

[Using the new Intel Streaming SIMD Extensions \(SSE4\) for audio, video and image apps](#)

**For more details regarding the Intel IPP library:**

[Intel® Integrated Performance Primitives](#)

[Intel® Integrated Performance Primitives Benchmarks](#)

[Intel® IPP linkage models - quick reference guides](#)

## Intel processors that support the IPP library

Intel® Pentium 4, Xeon and Pentium M processors include support for MMX through SSE2 technology. Processors based on Intel® Core and Intel® Atom support the MMX, SSE, SSE2, SSE3, and SSSE3 instruction sets. Those based on the Intel® Core 2 Duo also support the SSE4.1 instruction set. And those based on the Intel® Core i7 support the SSE4.2 instruction set.