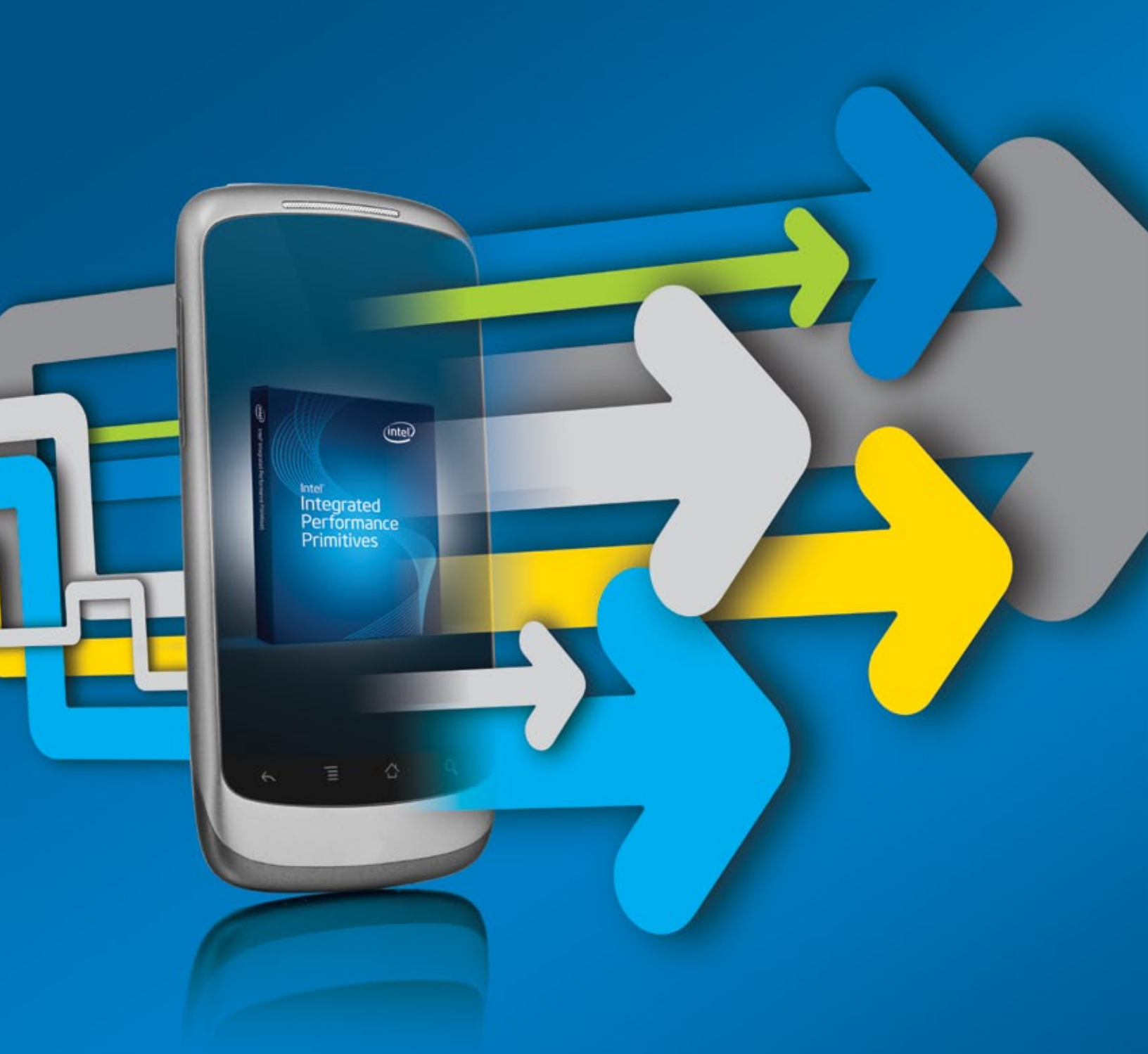

Pockets of Parallel Computation, Fitting in Your Pocket

The Intel® Integrated Performance Primitives Landscape for the Intel® Atom™ Processor

by Robert Mueller, Noah Clemons, and Paul Fischer
Software Technical Consulting Engineers, Intel

Discover how performance libraries, such as the Intel® Integrated Performance Primitives (Intel® IPP), impact applications for small form-factor devices, streamlining and unifying the computational execution flow for data-intensive tasks.

Employing performance library functions in applications running on small form-factor devices—otherwise known as *intelligent systems*—based on the Intel® Atom™ processor can be a great way to streamline and unify the computational execution flow for data-intensive tasks. This minimizes the risk of data stream timing issues and so called *heisenbugs*. Heisenbugs are hard to reproduce, and often seemingly random, runtime issues caused by the finely orchestrated timing of different System on Chip (SoC) components and digital signal processor (DSP) devices getting out of tune.

Performance libraries such as the Intel® Integrated Performance Primitives (Intel® IPP) contain highly optimized algorithms and code for common functions including signal processing, image processing, video and audio encoding and decoding, cryptography, data compression, speech coding, and computer vision. Advanced instruction sets help the developer take advantage of new processor features that are specifically tailored for certain applications. One calls the Intel IPP, as if it is a black box pocket of computation for a low-power or embedded device: 'in' flows the data and 'out' receives the result. In this fashion, using Intel IPP can take the place of many processing units created for specific computational tasks. Intel IPP excels in a wide variety of domains (**Figure 1**) where the Intel Atom processor for intelligent systems is utilized:

Without the benefit of highly optimized performance libraries, developers would need to carefully hand-optimize computationally intensive functions to obtain adequate performance. This optimization process is complicated, time consuming, and must be updated with each new processor generation. Intelligent systems often have a long lifetime in the field and there is a high maintenance effort to hand-optimize functions.

As seen in **Figure 1**, signal processing and advanced vector math are the two function domains that are most in demand across the different types of intelligent systems. Frequently, a digital signal processor (DSP) is employed to assist the general purpose processor with these types of computational tasks. A DSP may come with its own well-defined application interface and library function set. However, it is usually poorly suited for general purpose tasks. DSPs are designed to quickly execute basic mathematical operations (add, subtract, multiply, and divide). 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. The MAC instructions that comprise much of the code in a DSP application are the equivalent of Intel® Supplemental Single Instruction Multiple Data Streaming Extension 3 (Intel® SSSE3) instructions. Like the MAC instructions on a DSP, these Intel SSSE3 instructions perform mathematical operations very efficiently on vectors and arrays of data. Unlike a DSP, the Single Instruction Multiple Data (SIMD) instructions on an Intel Atom Processor are easier to integrate into applications using complex vector and array mathematical algorithms, since all computations execute on the same processor and are 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 coprocessor, that image data must be packaged for the DSP (placed in a memory area that is accessible by the DSP), signaled to execute the transformation algorithm, and finally returned to the general purpose

	Aerospace and Defense	Network Equipment	Consumer	Medical	Industrial	Automotive
Data Integrity	■	■		■		
Realistic Rendering			■	■		
String Processing	■	■				
Matrix/Vector Math	■	■	■	■	■	■
Speech Coding	■	■	■			■
Audio Coding	■		■			■
Video Compression	■		■	■		■
Image Compression	■		■	■		
Computer Vision	■				■	
Image Color Conversion			■	■		
Image Processing			■	■	■	■
Signal Processing	■	■	■	■	■	■
Cryotography	■	■		■		
Data Compression	■	■		■		

Figure 1: Intel® Integrated Performance Primitives function domains for the embedded space

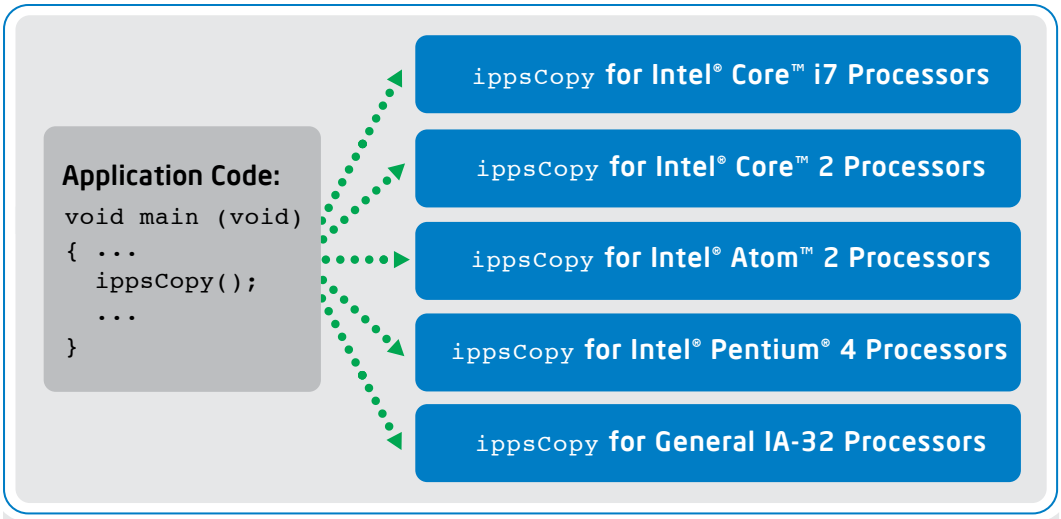

Figure 2: Library dispatch for processor targets

processor. Using a general purpose processor with SIMD instructions simplifies this process of packaging, signaling, and returning the data set. The Intel IPP library primitives are optimized to match each SIMD instruction set architecture so that multiple versions of each primitive exist in the library.

Intel IPP can be reused over a wide range of Intel® architecture-based processors and, due to automatic dispatching, the developer's code base will always pick the execution flow optimized for the architecture in question without having to change the underlying function call (**Figure 2**). This is especially helpful if an embedded system employs both an Intel® Core™ processor for data analysis and aggregation and a series of Intel® Atom™ processor based SoCs for data preprocessing and collection. In this scenario, the same code base may be used in part on both the Intel Atom processor SoCs in the field and the Intel Core processor in the central data aggregation point.

With specialized SoC components for data streaming and I/O handling, combined with a limited user interface, one may think that there are not a lot of opportunities to take advantage of optimizations

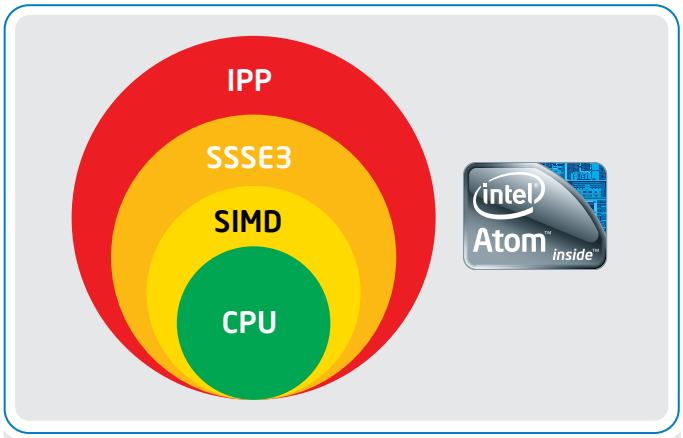

Figure 3: Intel® Integrated Performance Primitives is tuned to take advantage of the Intel® Atom™ Processor and the Intel® Supplemental Single Instruction Multiple Data Streaming Extension 3 instruction set

and/or parallelism with the Intel Atom Processor line, but that is not the case. There is room for:

- Heterogeneous asynchronous multi-processing (AMP) based on different architectures, and
- Synchronous multi-processing (SMP), taking advantage of the Intel® Hyper-Threading Technology and dual-core design used with the latest generation of Intel Atom processors designed for low-power intelligent systems.

Both concepts often coexist in the same SoC. Code with failsafe real-time requirements is protected within its own wrapper (managed by a modified round-robin real-time scheduler), while the rest of the operating system (OS) and application layers are managed using standard SMP multiprocessing concepts. Intel Atom processors contain two Intel Hyper-Threading Technology-based cores, and may contain an additional two physical cores resulting in a quad-core system. In addition, Intel Atom processors support the Intel SSSE3 instruction set. A wide variety of Intel IPP functions found at http://software.intel.com/en-us/articles/new-atom-support/ are tuned to take advantage of Intel Atom processor architecture specifics as well as Intel SSSE3 (**Figure 3**).

Throughput intensive applications can benefit from the ease of use of Intel SSSE3 vector instructions, and parallel execution of multiple data streams through extra-wide vector registers for SIMD processing. As just mentioned, modern Intel Atom processor designs provide up to four virtual processor cores. This fact makes threading an interesting proposition. While there is no universal threading solution that is best for all applications, Intel IPP has been designed to be thread-safe:

- Primitives within the library can be called simultaneously from multiple threads within your application
- The threading model you choose may have varying granularity
- Intel IPP functions can take advantage of the available processor cores directly via OpenMP*.
- Intel IPP functions can be combined with OS-level threading using native threads, Intel® Cilk™ Plus, or any other member of Intel's family of parallel models.

The quickest way to multithread an application that uses the Intel IPP extensively is to take advantage of the OpenMP threading built into the library. No significant code rework is required. However, only about 15 to 20 percent of Intel IPP functions are threaded. In most scenarios. It is therefore preferable to look to higher-level threading to achieve optimum results. Since the library primitives are thread safe, the threads can be implemented directly in the application, and the performance primitives can be called directly from within the application threads. This approach provides additional threading control and meets the exact threading needs of the application (**Figure 4**).

When applying threading at the application level, it is generally recommended to disable the library's built-in threading. Doing so eliminates competition for hardware thread resources between the two threading models, and thus avoids oversubscription of software threads for the available hardware threads.

Besides performance and maintainability, footprint and power consumption are also important considerations when developing a software stack for low-power intelligent systems. On a per instruction basis, SIMD instructions can consume slightly more energy than non-SIMD instructions. However, because they execute more efficiently—allowing an application to complete in less time—the net result can be less total energy consumed to complete the job. Thus, the use of SIMD instructions allows improved performance of critical workloads, while draining the battery less.

Intelligent systems frequently must work within confined memory and storage limits. Thus, the need to control the application footprint on a storage device (total binary size) or in memory during execution can be critical. If an application cannot be executed within the available resources, the reliability of the entire system can be in jeopardy.

Intel IPP provides flexibility in linkage models to strike the right balance between portability and footprint management (**Table 1**).

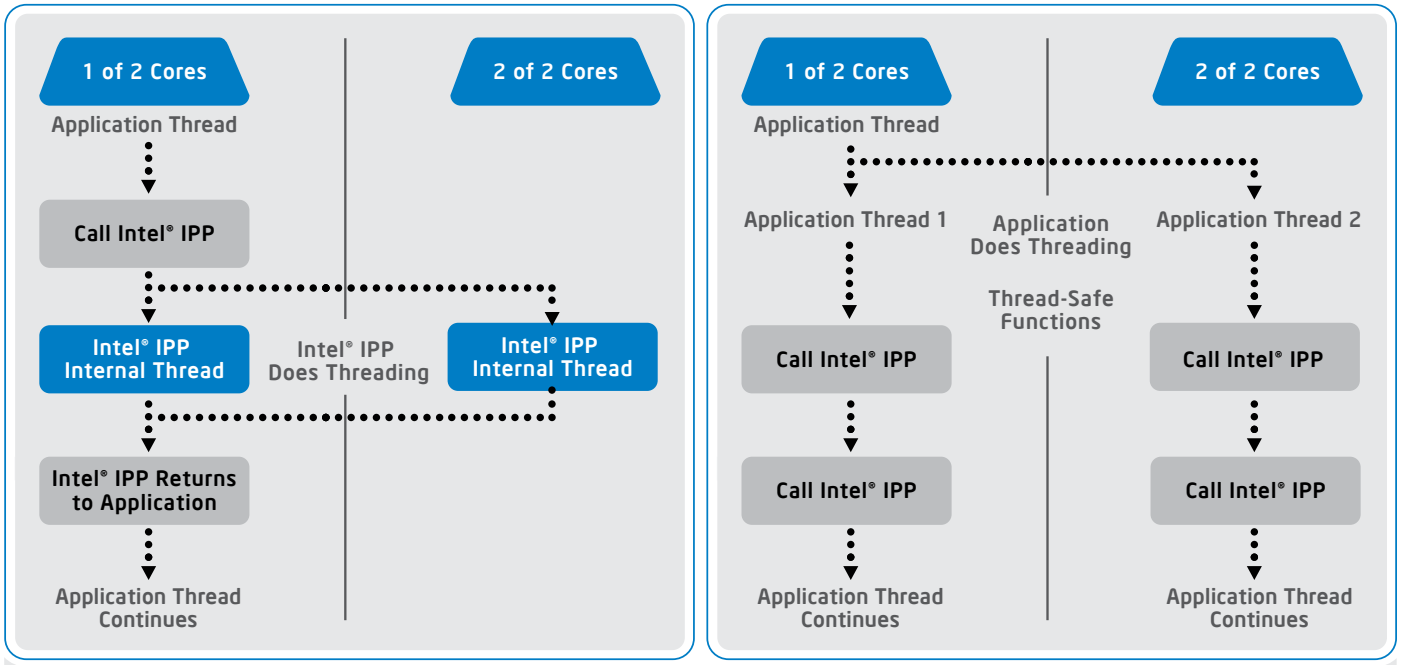

Figure 4: Function-level threading and application-level threading using Intel® Integrated Performance Primitives

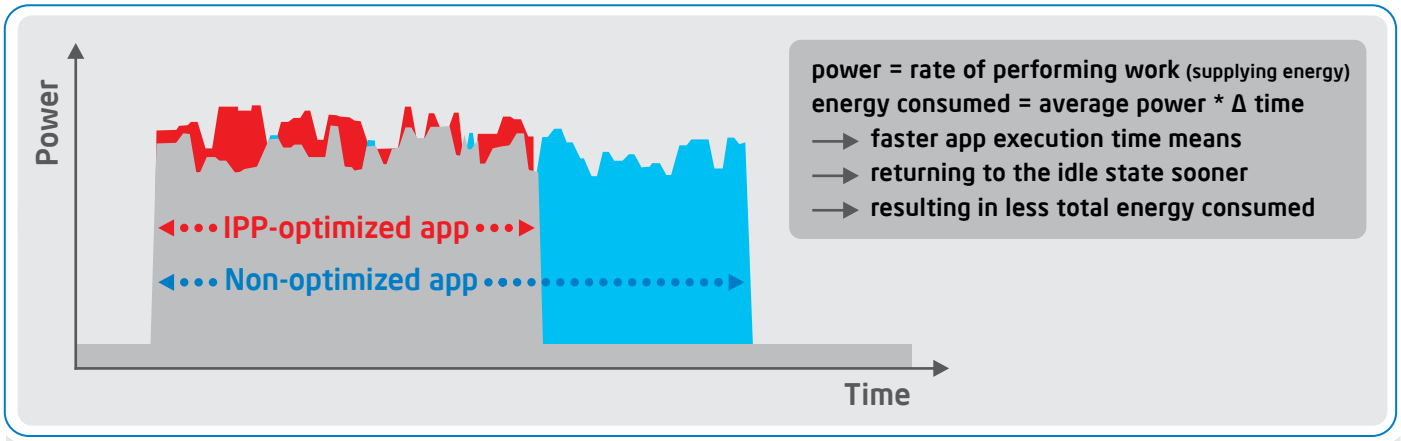

Figure 5: Impact of optimized library usage on power-consumption

The standard dynamic and dispatched static models are the simplest options to use in building applications with Intel IPP. The standard dynamic library includes the full set of processor optimizations and provides the benefit of runtime code sharing between multiple Intel IPP-based applications. Detection of the runtime processor and dispatching to the appropriate optimization layer is automatic.

	Standard Dynamic	Custom Dynamic	Dispatched Static	Non-Dispatched Static
Optimizations	All SIMD sets	All SIMD sets	All SIMD sets	Single SIMD set
Distribution	Executable(s) and standard Intel IPP DLLs	Executable(s) and custom DLLs	Executable(s) only	Executable(s) only
Library Updates	Redistribute as-is	Rebuild and redistribute	Recompile application and redistribute	Rebuild custom library, recompile application, and redistribute
Executable Only Size	Small	Small	Large	Medium
Total Binary Size	Large	Medium	Medium	Small
Kernel Mode	No	No	Yes	Yes

Table 1: Intel® Integrated Performance Primitives linkage model comparison

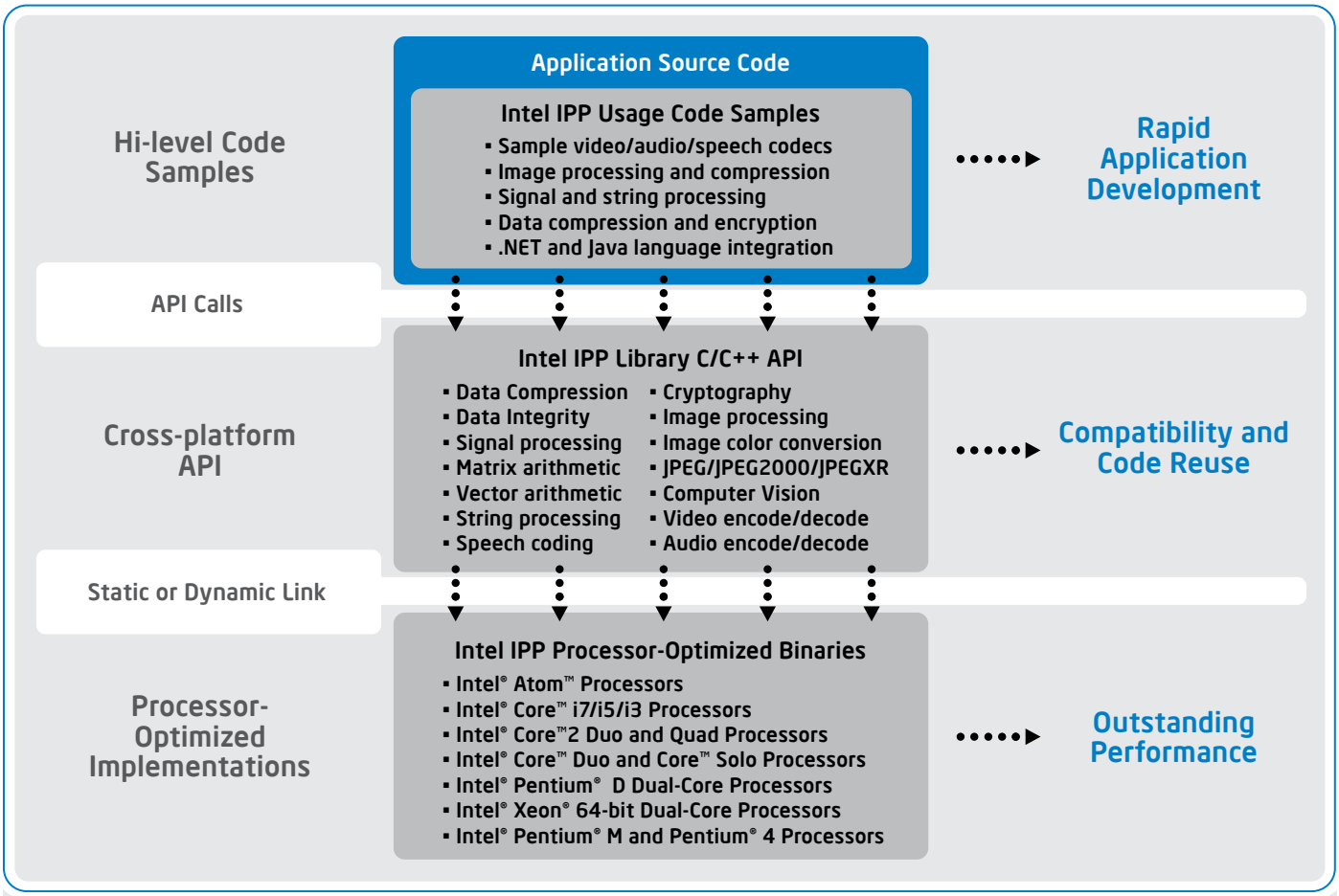

Figure 6: Intel® Integrated Performance Primitives overview. Many domains are covered along with sample source code for rapid development.

If the number of Intel IPP functions used in your application is small, and the standard shared library objects are too large, using a custom dynamic library may be an alternative.

To optimize for minimal total binary footprint, linking against a non-dispatched static version of the library may be the approach to take. This approach yields an executable containing only the optimization layer required for your target processor. This model achieves the smallest footprint at the expense of restricting your optimization to one specific processor type and one SIMD instruction set. This linkage model is useful when a self-contained application running on only one processor type is the intended goal. It is also the recommended linkage model for use in kernel mode (ring 0) or device driver applications.

The development ecosystem for embedded intelligent systems is not focused exclusively on the dominant operating systems of the desktop and server world. Frequently a SoC may use a custom embedded Linux OS based on something like Yocto Project* (http://www.yoctoproject.org) or Wind River* Linux (http://www.windriver.com) for the application and user interface layer, while other parts of the chipset may be running a fail-safe, real-time operating system (FTOS or RTOS).

Currently, Intel IPP is delivered for and validated against five different operating systems: Microsoft Windows*, Linux, Mac OS* X, QNX Neutrino*, and Wind River VxWorks*. QNX* and VxWorks* are limited to single-threaded static variants of Intel IPP. Application of Intel IPP to a "non-supported" operating system requires that the OS is compatible with the Application Binary Interface (ABI) defined by one of the aforementioned five operating systems, and that memory allocation routines can be accessed through a standard C library or mapped via *glue code* using a special i_malloc interface.

The atomic nature (no locks or semaphores) of the Intel IPP function implementation means that it is safe to use in the deterministic environment of a RTOS. An example of the value of applying the Intel IPP library to an RTOS would be the TenAsys INtime* RTOS for Windows. (http://www.tenasys.com). The INtime RTOS is an OS designed to run alongside Windows, handling real-time requirements on a Windows-based platform. The ABI used by INtime RTOS is compatible with the Windows ABI and employs Windows compatible function calling conventions. Using the Intel IPP in conjunction with such an RTOS expands its appeal by providing the performance of SIMD-based data throughput intensive processing, with determinism usually only characteristic of DSPs.

Intel IPP addresses the needs of the native application developer found in the personal computing world, as well as the intelligent system developer who must satisfy real-time system requirements with the interaction between the application layer and the software stack underneath. By taking Intel IPP into the world of middleware, drivers, and OS interaction, it can also be used for embedded devices with real-time requirements and dominant execution models. The limited dependency on OS libraries and its support for flexible linkage models makes it simple to add to embedded cross-build environments, whether they are RTOS-specific or follow one of the popular GNU*-based cross-build setups like Poky-Linux* or MADDE*.

Developing for intelligent systems and small form factor devices frequently means that native development is not a feasible option. Intel IPP can be easily integrated with a cross-build environment and used with cross-build toolchains that accommodate the flow requirements of many of these real-time systems. Use of Intel IPP allows embedded intelligent systems to take advantage of Intel SSSE3 vector instructions and extra-wide vector registers on the Intel Atom processor. Developers can also meet determinism requirements, without increasing the risks associated with cross-architecture data handshakes of complex SoC architectures.

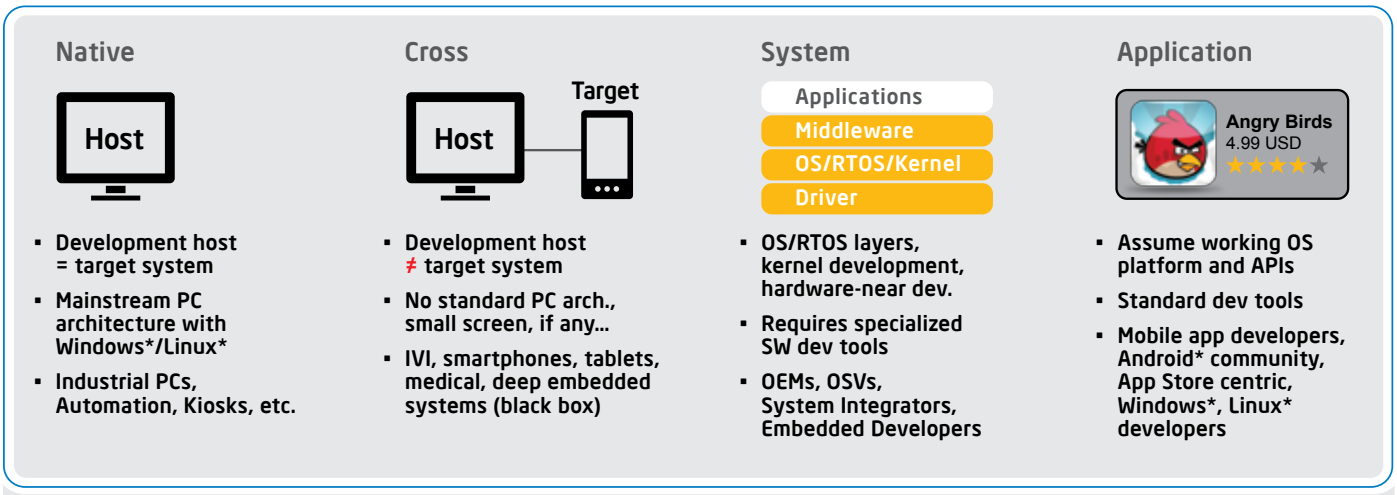

Figure 7: Intel® Integrated Performance Primitives can be used over a full range of development setups and software stack targets

Developing for embedded small form factor devices also means that applications with deterministic execution flow requirements have to interface more directly with the system software layer and the OS (or RTOS) scheduler. Software development utilities and libraries for this space need to be able to work with the various layers of the software stack, whether it is the end-user application or the driver that assists with a particular data stream or I/O interface. Intel IPP has minimal OS dependencies and a well-defined ABI to work with the various modes addressed in (Figure 7). One can apply highly optimized functions for embedded signal and multimedia processing across the platform software stack, while taking advantage of the underlying application processor architecture and its strengths—all without redesigning and returning the critical functions with successive hardware platform upgrades.

Getting Started with Intel® Integrated Performance Primitives on Intel® Atom™ Processors

- Purchase Intel IPP, or download a trial copy (http://software.intel.com/en-us/articles/intel-ipp/)
- Check out the free code samples that come with Intel IPP (http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-code-samples/), and see if any match the needs of one of your algorithms (implemented or planned). The Intel IPP code samples are available for the Windows, Linux, and Mac OS* operating systems. Instructions on how to build each sample can be found in sample-specific ReadMe files located in each sample's main directory. Some samples include more detailed documentation in a doc directory, usually in the form of a PDF file.
- If no example code sample matches your needs, take a look at the reference manual (http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/ippxe/ipp_manual_lnx/index.htm), which is organized by function domains, including signal processing, and then by classes of functions within that domain. See **Figure 1** for the 12k Intel IPP functions in fourteen different domains.
- Next, choose a linking option from the many described in **Table 1** that works for you, and you are ready to begin. The article located at http://software.intel.com/en-us/articles/introduction-to-linking-with-intel-ipp-70-library/ provides a more in depth discussion of these options. □

BLOG highlights

Some Performance Advantages of Using a Task-Based Parallelism Model

SHANNON CEPEDA, Intel

As part of my focus on software performance, I also support and consult on implementing scalable parallelism in applications. There are many reasons to implement parallelism, as well as many methods for doing it—but this blog is not about either of those things. This blog is about the performance advantages of one particular way of implementing parallelism— and luckily, that way is supported by several models available.

I am talking about task-based parallelism, which means that you design your algorithms in terms of "tasks"(work to be done) instead of the specifics of threads and CPU cores. There are several parallelism models currently available that use tasks: Intel® Threading Building Blocks (TBB), Intel® Cilk Plus, Microsoft* Parallel Patterns Library, and OpenMP 3.0, to name a few. With a task-based library (like the ones mentioned) you can use pre-coded functions, pragmas, keywords, or templates to add parallelism to your application ...

SEE THE REST OF SHANNON'S BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804