

Accelerating High-Performance Embedded Applications Using a Massively Parallel Processing Array

By Laurent Bonetto
Technical Marketing Engineer
Ambric, Inc.

This article discusses how Massively Parallel Processing Arrays (MPPAs) can be used to accelerate high-performance embedded system applications. In this article, we discuss:

- The requirements of high-performance applications
- How MPPAs compare with other architectures.
- The Ambric Am2045 architecture and software tools in the context of an application, detailing the amount of effort involved in programming an MPPA architecture to implement a JPEG image compression application.

Conventional solutions and requirements for high-performance embedded applications

High-performance embedded video and imaging applications are characterized by complex algorithms that need to process data at a high rate of throughput. For example, the video market segment contains several high-end applications, such as video codecs, medical imaging algorithms, and intelligent imaging, which involve tens to hundreds of operations-per-pixel and target large image resolutions. Similarly, many wireless applications apply complex transforms to incoming streams of high-throughput data. Many of these high-end applications rely on state-of-the-art standards and continuously changing proprietary algorithms.

As a result, targeting high-performance DSP applications requires that developers create implementations that:

- Are fast enough to meet demanding processing requirements
- Are developed quickly enough to reach the market on time
- can easily be upgraded to provide a different functionality

There are three types of architectures commonly used today to meet these requirements: Application-specific Integrated Circuits (ASICs), Field-programmable Gate Arrays (FPGAs), and high-end Digital Signal Processors (DSPs).

ASICs provide excellent performance but lack any kind of programmability. This type of architecture is therefore commonly found in very high-volume applications where any savings on cost or power consumption can justify the lack of programmability. Lower volume applications do not justify the high cost, risk, and complexity of the initial development associated with ASICs.

FPGAs are a step down from ASICs in terms of performance and a step up in term of ease of use. Since they rely on reprogrammable logic, FPGAs are inherently less silicon-efficient than ASICs. What is lost in efficiency is gained in flexibility, because FPGAs, unlike ASICs, are programmable. The common programming language for FPGAs is Register Transfer Language (RTL), a language applying very basic operations such as Boolean equations and control logic

on registers such as Verilog or VHDL. FPGAs are versatile solutions that are well suited for interfacing devices (thanks to their extensive and programmable IOs), prototyping early silicon, or running low-volume, high-performance applications.

High-end DSPs are at the opposite end of the spectrum from ASICs. DSPs are processors programmed in a mix of high-level language and assembly language, a property that is appealing to software developers. These processors have been tuned to handle DSP applications efficiently.

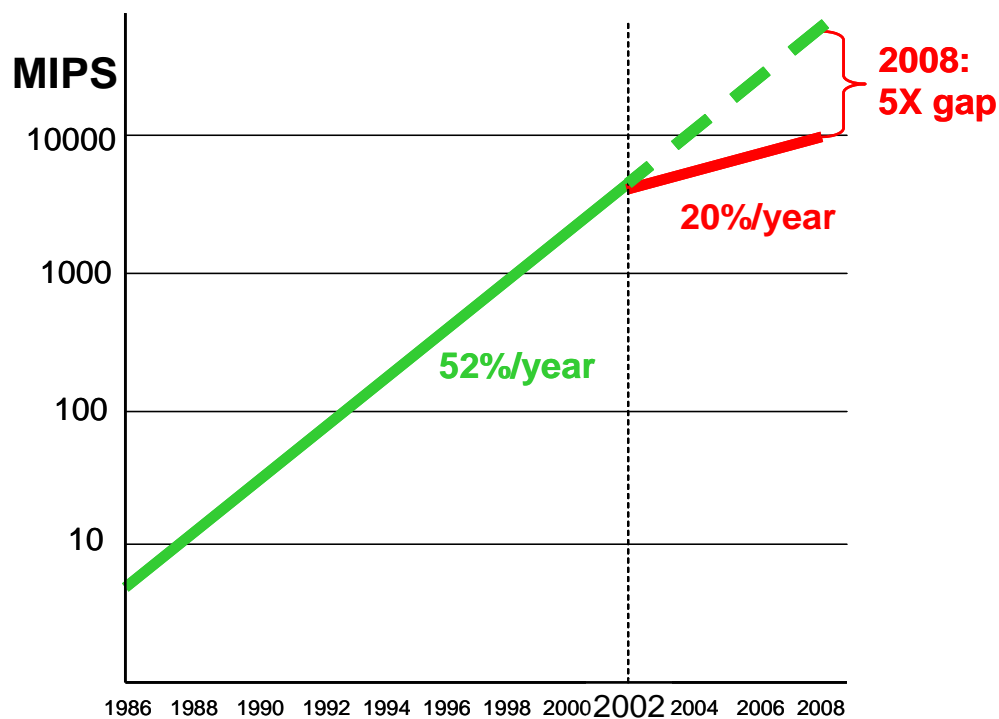
For example, most DSPs include hardware multipliers for efficient multiply and multiply-accumulate operations, and support two memory accesses per cycle, to run more efficiently algorithms such as filters that operate on two arrays at once. High-end DSPs have been further optimized to meet more demanding processing requirements. These processors include:

- Multiple specialized instructions that implement specialized operations commonly found in DSP applications (e.g., Sum-of-Absolute-Differences or SAD)
- An increased number of pipeline stages to reach higher clock speeds
- An increased number of processing units and more complex sets of parallel instructions to increase the number of operations that can run in parallel during each processor cycle.

Limitations of ASIC, DSP, FPGA and multicore

Today's conventional approaches to embedded systems design have major limitations when it comes to high performance applications.

Figure 1: Processor increase in speed over the past three decades.



From Hennessy & Patterson, Computer Architecture, 4th ed., 2006

FPGA

One of the main challenges of using FPGAs is programming them. Writing code for an FPGA requires two distinct sets of skills of which few people possess both. On the one hand, writing efficient code on an FPGA requires a good understanding of hardware constraints, such as timing closure, and skills in writing code at the logic level. On the other hand, working with complex DSP applications on FPGAs requires good software skills to create and modify the reference application. It also requires knowledge of DSP theory to know how an algorithm can be modified to run more efficiently or to interpret correctly the properties of the signals being manipulated so as to resolve bugs more quickly.

Another problem with FPGAs is the complexity involved in creating a correct design. Studies show that the amount of time a developer spends per gate decreases at a slower pace than is needed to keep up with the exponential increase in transistor density. Large designs require test benches and extensive simulation, as is the case with an ASIC. FPGA designs of average complexity often take hours to place and route; this impedes significantly the development and validation of code. This increased programming complexity results in either longer development times or designs that do not efficiently use the logic available.

ASICs

ASICs pose the same programming challenges as FPGAs, while also incurring very high NRE costs. In addition, ASICs cannot be modified. This makes ASICs appropriate for very high-volume and cost-sensitive applications but unsuitable for most other markets.

DSPs

High-end DSPs pose an interesting contradiction. The strength of DSPs is ease of programming: software developers code entire applications in a mix of high-level and assembly languages, which is a major advantage compared with coding at the register-transfer level (RTL). However, in recent years, high-end DSPs have lost this advantage.

Now, to keep pace with growing application requirements, high-end DSPs are packed with specialized hardware that complicates the tasks of software and compiler developers, who need to master the implications of each hardware feature to create optimized code.

For example, does the processor have an increased pipeline? If so, there will be more instructions with longer latencies, forcing the software developer to perform instruction rescheduling, loop unrolling, and other coding contortions to avoid stall cycles. Another example: does the processor have a specialized instruction or a specialized hardware accelerator to handle a specific function? If so, this will require the developer to architect the code and data flow to take advantage of that hardware. It will also mean that more code needs to be written by hand as the compiler will be likely unable to take advantage of the specialized hardware from a high-level C description of the algorithm. These examples illustrate the fact that the more sophisticated the hardware, the more assembly code the software developer needs to write by hand and the more difficult that code is to write.

Another important factor in the use of high-end DSPs is that practically all architecture improvements that could provide a noticeable improvement in the performance of a single processor have been developed. As illustrated in ["Figure 1:" on page 2](#), above, the overall speed improvement of DSPs has dropped from 52 percent to 20 percent each year over the past five or six years. This 20 percent rate is essentially the rate at which clock speed increases as a result of

moving to lower process nodes. This means that high-end DSPs cannot satisfy the demands of an increasing number of high-end applications.

Alternate solutions for high-performance embedded applications

The conventional architectures have strengths and weaknesses, so it seems reasonable to create systems that combine different types of architectures. For example, some systems combine multiple FPGAs and DSPs on a single board. This provides the advantages of re-using subsystems that have been tested successfully, leveraging the programmability of DSPs, and using FPGAs to speed up specific parts of the high-end application with which the DSP cannot keep up. While this approach makes theoretical sense, the resulting complexity of this type of system makes it very difficult to debug and maintain. Some of the common challenges include the need to master different types of architectures, the need to access simultaneously the state of multiple devices on the same board, and the need to use different tools. As a result, many of these complex systems fail to work properly and do not make it into actual products.

Multicore approaches

Another approach to developing high-end embedded applications is the use of multicore processors. Multicore processors generally include two types of cores: RISC, to handle control-oriented tasks and/or DSP, to handle the most demanding algorithms of an application.

In its smallest form, a multicore processor is dual-core and contains one core of each type, with both relying on the same shared memory for inter-core communication. With this approach, the RISC core generally runs an operating system and any non-demanding application code. The RISC core uses the DSP to offload the most compute-intensive tasks. This approach presents the advantage of using each core for its best use. However, the multicore approach still presents many challenges. The performance of a multicore device is limited by the performance of the DSP, which has the limitations that were already described in [the previous section](#). In addition, in these kinds of dual-core solutions, the RISC and DSP cores are generally quite different: each core comes with its own programming language, development environment, set of libraries, and architectural features, which all require expert knowledge on the part of the software developer. In addition, there is rarely an integrated development environment for programming the RISC and DSP cores together.

Another multicore approach is combining on the same device a larger number of identical DSP-oriented cores connected to shared memories. The shared memories store code and data and are used for inter-processor communication. Since each processor has similar access to a common memory space, this architecture is called a symmetric multiprocessor (SMP). Increasing the number of DSP-oriented cores that run in parallel allows higher performance. However, this comes at a high cost on the programming model. The gain in performance is only present if the software developer is able to share the workload of the application in such a way that each core runs a portion of the application at all times and keeps the inter-core communication overhead to a minimum. To achieve this, the software developer must write code that forks off to separate threads of execution, explicitly manages data sharing and synchronization among the threads, and at the same time, keeps data movements across cores to a minimum.

Writing such code is difficult and prone to errors. And, if a thread halts somewhere, another thread could change memory so its state would not be the same when it restarts. Ensuring that no thread mistakenly affects the behavior of another thread is challenging and entirely up to the software developer. This is because the SMP programming model and languages have no intrinsic protections. Instead, software developers must explicitly lock and unlock variables (incurring additional overhead) and explicitly create and join threads to assure correct

synchronization. Worse yet, SMPs do not scale well, so keeping multi-threaded code safe, finding enough work to run on all cores at once, and ensuring that memory doesn't become a bottleneck all become more difficult with the increased number of cores communicating through the same shared memory.

MPPAs: The multicore approach taken to the extreme

If having a few cores makes code significantly more difficult to write and less reliable, there seems to be little value in having hundreds of cores running in parallel on one chip. That is, unless inter-core communication relies on a completely different model. This is what Massively Parallel Processor Arrays (MPPAs) provide.

MPPAs include a large number of processors that communicate together. The number of processors can vary considerably but is currently in the hundreds. Unlike the SMP approach, each processor in an MPPA is strictly encapsulated, accessing only its own code and memory. Point-to-point communication between processors is directly realized in a configurable interconnect. This architecture leads to a very different programming model, one where each processor devotes separate channels for each type of input and output data it needs to share with other processors. The code running on each processor is fully encapsulated in that it doesn't make any assumption about which processor it communicates with. Each processor runs a specific task with the guarantee that no other processor will affect its state. As long as each data stream coming in and out of a processor is in the expected format, each processor is guaranteed to always produce the same functionality.

Far from compounding the problems inherent in SMP architectures, an MPPA having hundreds of processors simplifies the work of the software developer by providing much more flexibility. With SMP architectures, having only a few cores available makes it challenging for the software developer to find ways to split the application among each core and maintain a similar workload on all cores. In contrast, MPPAs with hundreds of available processors provides the software developer with finer granularity. A full application maps naturally into a number of functions, each of which maps into separate processors. Having some underutilized processors running trivial functions is acceptable given the large number of processors available. Similarly, there are usually enough processors available that the most demanding functions can be further decomposed to run on multiple processors. This process will be illustrated in more detail in the sections of this article that discuss the JPEG application example. See [“Step 2: Optimizing the design” on page 13.](#)

Each processor on an MPPA is usually much simpler than a high-end DSP because it does not need to be highly optimized to single-handedly meet the requirements of an application. With hundreds of processors on a chip, it is crucial that the hardware logic used in each processor remains simple. This keeps the silicon area low and optimizes the overall performance-per-transistor and performance-per-watt. Ultimately, keeping the processors simple serves the needs of both hardware and software designers.

Another desirable property of an MPPA is its scalable architecture. In MPPA architectures, blocks containing a small number of processors and local memories are joined together in a configurable 2D mesh interconnect. This approach, similar in concept to what FPGAs offer at the gate level, provides a straightforward path to producing more powerful parts with faster processors. Increasing performance is achieved simply by extending this 2D network and replicating a larger number of blocks. Unlike high-end DSPs, MPPAs do not rely on a continuous addition of new features to the processor architecture to guarantee steady performance increases.

Another advantage of MPPAs is that, unlike FPGAs, their elementary programming component for the software developer is a processor, not a gate. This gives the software developer major benefits not found with FPGAs and ASICs. Specifically, 1) the ability to write code in a language such as C or assembly that provides a behavioral description of the algorithm, as opposed to RTL, which is a hardware-oriented language and 2), the ability to run designs in a matter of seconds instead of spending hours on placement and routing.

In addition, high-performance applications generally offer numerous opportunities for data or functional parallelism, so an MPPA outperforms the fastest high-end DSPs by a big margin. When used in applications that match the common types that MPPA processors handle efficiently (typically, 8-bit, 16-bit, and 32-bit), an MPPA rivals the performance of one or more high-end FPGAs.

That each processor on an MPPA is simple removes many of the difficulties of assembly programming for high-end DSPs. And, because these processors are strictly encapsulated, they do not have the lower reliability of SMPs. However, a chip running hundreds of processors in parallel has its challenges, all of which should be addressed by the MPPA architecture and its associated set of tools:

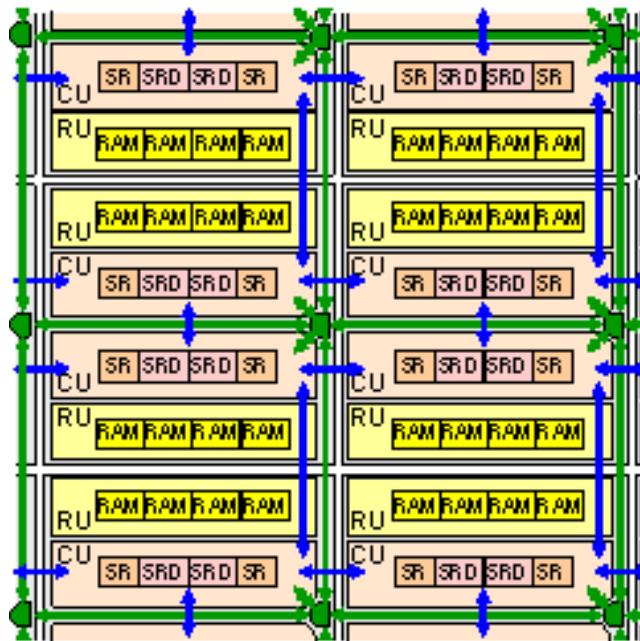
- Getting started running something slow but functional on the chip
- Quickly developing code that uses the hundreds of processors available
- Ensuring that code scales well to use more or less of the chip depending on the processing requirements of the applications
- Keeping processors synchronized with each other
- Keeping all processors busy
- Debugging a design with hundreds of processors running in parallel

Addressing these challenges requires a well designed programming model and a combination of architecture and tools that can realize the programming model. Not all MPPAs offer these features.

Ambric took the original approach of first developing a robust and usable programming model and then implementing an architecture and set of tools. This approach produced the Ambric Am2045, an MPPA device that fulfills the requirements of high-performance embedded system applications while being highly scalable and highly productive to program.

Introduction to the Ambric Am2045 MPPA

The Am2045 features hundreds of processors and local memories organized into blocks (referred to as brics) and communicating through a 2D interconnect network. The Am2045 includes a 9-row-by-5-column array of brics surrounded by external SDRAM interfaces, a PCI Express interface, and general-purpose I/O (GPIO) ports. Each bric has four Streaming RISC processors (SRs), four Streaming RISC processors with DSP extensions (SRDs), and eight 2KB RAM banks. Within each bric, these processors and memory banks are organized in two Compute Units (CUs) and two RAM Units (RUs).

Figure 2: Part of the bric array showing neighbor channels and a distant channel network

SRs and SRDs have their own instruction memories. SRDs can also run larger code by executing instructions stored in the RAM banks. Memory banks are not shared: each bank, assigned to processors at compile time, can be read from or written to by only one processor.

Any Ambric processor can send data to another processor on the chip or access any RAM bank on the chip using the 2D interconnect of channels. Ambric channels are word-wide, self-synchronizing pathways for communicating between processors. When a processor attempts to write to a channel already full, or read from an empty one, the processor stalls automatically to let each processor execute code independently without any synchronization issues. Each Am2045 channel has a bandwidth of 10 gigabits per second.

The Am2045 interconnect is a configurable three-level hierarchy of channels (Figure 2). At the base of the hierarchy are the local channels inside each CU (not shown in the figure). At the next level, blue channels directly connect CUs with neighboring CUs. At the top level, the green network for long connections is a 2D circuit-switched interconnect of channels and configurable switches. Switches are statically configured.

Programming the Am2045

The Am2045 platform architecture implements the Ambric Structural Object Programming Model. Objects consist of one or more software programs running concurrently on an asynchronous array of Ambric processors and memories. A leaf object runs on a single processor or memory while a composite object is a hierarchical composition of objects.

Objects run independently at their own rates. They are strictly encapsulated, execute with no side effects on one another, and have no implicitly shared memory. Objects exchange data and control through a structure of self-synchronizing Ambric channels. Each channel is word-wide, unidirectional, point-to-point, strictly ordered, and acts like a FIFO-buffer. Objects are mixed and matched hierarchically to create new objects snapped together through channel interfaces.

The application developer expresses an object-level parallel structure in a coordination language called aStruct, which defines how objects are connected to each other. This language is essentially a simple textual representation of an implementation block diagram.

Having hundreds of processors available to implement an application provides a lot of flexibility to the user. The software developer can divide the target application into individual function objects that map naturally into separate processors. Then the task of implementing each of these objects on a processor becomes simple and self-contained. This Structural Object Programming Model makes it efficient for a team of software developers to work in parallel and quickly implement any given application because each programmer is assigned several simple, well-defined functions to run on individual processors.

The code running on an individual object can be written in Java or assembly language. Because of the architecture's simplicity, the RISC assembly language is relatively straightforward to use for producing optimal code. The Java language, which is compiled into assembly before running on each processor, was chosen because it is a simple, powerful, and safe language. Java overlaps for the most part with C with the notable exception that software developers need not use pointers in Java. The resulting code is simpler to read, as arrays are always accessed as an offset from a base. The code is also safer since no memory contents can inadvertently be modified due to the incorrect use of a pointer.

DSP developers familiar with C may wonder about the differences between the embedded C code familiar to them and the Java code used to program Ambric processors. There are very few differences between the two, with the exception of pointer use (as mentioned above).

[Table 1 on page 9](#) summarizes the differences between C and Java for code that is usually found in embedded DSP applications.

The table illustrates how similar Java and C are when developing DSP embedded code. In fact, when embedded Java code needs to be developed from C, the simplest approach consists of pasting the C code directly into the Java programming environment and correcting the errors underlined by the Java editor. Apart from the occasional use of pointers, the C code would usually compile readily and run under the Java environment.

Table 1: Differences between C and Java for embedded code

C	Java
Variable declarations: Same (except no unsigned in Java)	
<pre>int x0, x1, temp=1, i=0; int acc, array[16];</pre>	<pre>int x0, x1, temp=1, i=0; int acc, array[16];</pre>
Arithmetic operations: Same	
<pre>x0 = x1 = 0xff & array[0]; temp -= (x0 + x1)<<2; array[i++] = x0 * x1;</pre>	<pre>x0 = x1 = 0xff & array[0]; temp -= (x0 + x1)<<2; array[i++] = x0 * x1;</pre>
Control code: Same	
<pre>while(true) { for (int j=0;j<N;j+=2) acc+=array[j]; if (acc>=MAX) break; }</pre>	<pre>while(true) { for (int j=0;j<N;j+=2) acc+=array[j]; if (acc>=MAX) break; }</pre>
Function calls: Same	
<pre>temp = max(x0,x1);</pre>	<pre>temp = max(x0,x1);</pre>
Pointer manipulations: Rewrite (For example, use arrays instead.)	
<pre>*ptrDelayLine++=0;</pre> <p>Note: If <code>ptrDelayLine</code> is not properly initialized, that instruction will modify some unknown memory contents and may cause various hard-to-track bugs.</p>	<pre>delayLine[index++]=0;</pre> <p>Note: If <code>index</code> is not properly initialized, this instruction will cause an error message and the program will stop safely.</p>

In addition to the common support for Java, Ambric libraries provide support for special types that declare incoming and outgoing channels. Thanks to these types, retrieving words from an incoming channel or writing words to an outgoing channel is as simple as accessing a regular variable.

Ambric aDesigner is the Eclipse-based Integrated Development Environment (IDE) developers use to:

- compile and assemble object code
- simulate execution
- automatically place and route objects and structure onto processors, memories, and channels
- generate, load, and run designs on the chip
- access processor state and memory contents during debugging
- and gather and display profiling information

Full applications take between a few seconds and one minute to compile.

Target application example: JPEG codec

This section illustrates how to apply this programming model to successfully implement embedded applications by using a JPEG codec and discussing its implementation on the Am2045.

Relevance of JPEG

JPEG is a standard for image compression. Or rather, JPEG is the standard for image compression today. While the video compression space is cluttered with a sea of codecs like MJPEG, MPEG-2, MPEG-4, H.264, AVC Pro, Xvid, WMV—just to name a few—JPEG seems to dominate the image encoding space.

The JPEG compression format is also found in the video space in the MJPEG video codec, which is essentially a sequence of JPEG images put together.

Finally, almost all video compression standards share many functional blocks with JPEG. Most video codecs operate on small blocks of pixels (8x8 or similar sizes) on which they perform similar operations such as transformation into the frequency domain (DCT or similar algorithms), quantization, run-length encoding, and so on.

JPEG is a representative and realistic example of a high-performance embedded application, and JPEG is simpler to describe than many other high-end applications, making it practical for the context of this paper. The process for implementing JPEG on the Am2045 is identical to what is required for developing other high-end applications, such as the existing Am2045 GT Video Reference Platform that can be used for H.264, MPEG2, JPEG2000 and many other powerful codecs.

Algorithm overview

In the remainder of this article, we use the term JPEG to refer to the most common JPEG mode: baseline JPEG.

JPEG is a lossy image compression codec. The encoder transforms an image into a compressed bit stream using the following operations:

- Each image is converted to a chroma-luma color space where the chroma pixels can be downsampled to provide a first opportunity for compression.
- The image is divided into 8x8 blocks of luminance (Y) and chrominance pixels, which are transformed into the frequency domain using a Discrete Cosine Transform (DCT). This transformation decorrelates the image, concentrating most of the energy into a few (low-frequency) coefficients.
- Each DCT-transformed block is then multiplied by a quantization matrix, which provides a "knob" to adjust the level of compression: Increasing the quantization deteriorates the image quality but increases the compression ratio.
- Each quantized coefficient is then read in a zigzag order, a pattern that orders the coefficients from low to high frequencies. This introduces large groups of consecutive zeros toward the end of each block.
- The coefficients in zigzag order are then run-length encoded, a process that encodes each non-zero coefficient as a pair where the first number represents the number of preceding zeros and the second number represents the value of the non-zero coefficient. This step allows us to compress long runs of zeros.

- Finally, the run-length codes are entropy coded using Huffman tables. These tables allocate fewer bits to the most common codes, thus reducing the average size needed to represent each code.
- The resulting string of Huffman code is packed into a JPEG file following a header containing the information needed to decode the file, such as the quantization and Huffman tables used for encoding each color component.

An additional compression step consists of encoding the first coefficient of each block (the DC coefficient) as a subtraction from the DC coefficient of the preceding block, thus taking advantage of the similarities between neighboring blocks.

The JPEG decoder simply applies the transformations described above in reverse order: the decoder first extracts and decodes the Huffman codes from the packed bit stream, then rearranges the data from a zig-zag order to a block order, performs an inverse quantization, and finally an inverse DCT (IDCT).

Design development

Most software developers undertaking a complex embedded design development break that process into three major steps:

- Creating a functionally correct design as quickly as possible
- Optimizing the design to meet the system-level objectives
- Executing the design on hardware

We now discuss each of these steps in detail and explain how each step compares to an MPPA as compared to other conventional architectures such as FPGAs and high-end DSPs. We also will illustrate this development process by discussing the implementation of the JPEG codec on the Ambric Am2045 MPPA architecture using Ambric aDesigner software development tools. This approach demonstrates how an MPPA supported by proper software development tools offers a straightforward path to creating optimized designs.

Naturally, software developers are not restricted to follow the three-step approach discussed in this article, and aDesigner is designed to offer that flexibility. For example, a software developer may need to develop prototype blocks and then test them directly in hardware prior to optimizing them. Given that the Ambric aDesigner tool suite takes only a few seconds to about a minute to place and route designs, developers can follow this approach successfully.

Step 1: Producing a functionally correct design

Objective

A common task for software developers is to produce a reference design that will run, if not on the target chip, at least in the development environment of that chip. That design isn't expected to run fast, but it needs to produce correct output so that it can be used as a starting point for progressively creating a fully optimized implementation.

Usually, the design being produced is abstracted from most features of the target architecture, which allows the design to be less complex and more general.

aDesigner provides a programming environment that simulates Java code without compiling or running the code on the actual chip. This environment is perfectly suited for developing generic

code since no architecture limitations are enforced—an arbitrary number of processors and channels can be used and the amount of on-chip memory available is unlimited.

Details of the approach taken

High-level algorithm descriptions usually come in the form of detailed specifications and their associated reference code usually written in MATLAB or C. In the case of JPEG, our starting point was the ITU T.81 specifications and one of the most commonly used, freely available, and reasonably optimized C reference source codes for JPEG: the IJG JPEG library (version 6B).

The IJG JPEG library is a good representative of the kind of reference code that embedded software developers start from: it contains a large amount of code not needed for the target embedded application such as, in our case, code supporting multiple JPEG modes other than JPEG baseline. In this kind of situation, the software developer is left with two approaches. He or she can become familiar with the entire reference code and trimming down what isn't needed, or write new reference code from scratch and borrow code sections from the reference code when practical. We chose the latter approach. Both approaches usually require a comparable amount of effort.

Test vectors

A step that can be time consuming when developing a design from scratch is creating test vectors to validate progressively the functionality of the blocks as they are created. We solved that challenge by developing the JPEG encoder and decoder in parallel so that each encoding and decoding block could be used to test the other without any need to create reference output data.

Allocating processors

The resulting reference JPEG implementation is a very natural decomposition of the algorithm into a small number of processors each of which independently runs a section of the JPEG algorithm roughly corresponding to one of the blocks presented earlier in the section [“Algorithm overview” on page 10](#). The process of allocating processors to tasks flows naturally from the desire to keep the implementation simple to write.

For example, one processor successively runs the zig-zag encoding block followed by the quantization block by simply reading the input data for the quantization in a zig-zag order. Another processor runs the run-length encoding block and Huffman encoding blocks together because devising a way for these two blocks to communicate efficiently while running independently on two processors would be cumbersome.

Following the same principle of easing the work of the software developer, other blocks can instead be divided across multiple processors. This was the case with the byte-stuffing operation required by the JPEG standard: byte-aligned 0xFF symbols must be followed by a zero byte. Writing code that successively packs bit strings of arbitrary sizes into bytes, performs byte-stuffing, and then packs the byte stream into 32-bit words is difficult and error-prone. It also results in code that is excessively long, is difficult to follow and to optimize in assembly, and has low throughput.

On a conventional architecture, such a sequential approach is the only approach available because dividing the work into smaller chunks—for example by using multiple threads or multiple function calls—introduces too much overhead. On the other hand, with an MPPA, dividing the work makes it simpler and more importantly doesn't add cycle overhead. We used one processor to pack the bit strings into 32-bit words and used another processor to read these words one byte

at a time and generate a stream of bytes that has been byte-stuffed. The resulting code running on each processor is both simpler and more efficient.

In summary, we can use the hundreds of processors available on the MPPA to our advantage whenever it makes the coding simpler. And since all processors stall automatically when reading from an empty channel or writing to a full channel, no code is required for inter-processor synchronization.

Creating the actual design

We will now discuss the amount of work involved in creating the actual design. Creating a design from scratch using Java requires a level of effort comparable to creating the same design in C and much lower than what is required for using RTL. Most blocks were simple enough to be written from scratch with only two exceptions:

- DCT: A good DCT implementation takes months to develop, and it makes sense to leverage existing optimized code. We reused the DCT code found in the IJG C reference code. How much effort was required to convert that C code into Java code? Zip! To be more precise, we had to replace five invocations of the macro MULTIPLY that implemented a fixed-point multiply by an actual multiplication instruction. In other words, all of the C code used in the IJG DCT compiles readily as Java code.
- Initialization routines for creating the Huffman lookup tables: The Huffman tables encoded in the JPEG header are in a compact form. Any efficient JPEG implementation requires these to be expanded. The process for expanding these tables doesn't need to run fast but isn't trivial to write. Again we reused the code provided with the reference standard. Effort to convert that C code into Java code? None. All of the C code used in the IJG Huffman initialization routines compiles readily as Java code. The only changes consisted of renaming variables to follow a different coding convention.

These two examples illustrate the point made earlier that embedded code written in Java and C is most often identical. See [Table 1 on page 9](#).

Step 2: Optimizing the design

Objective

Optimizing an application to meet its requirements is the most common and time-consuming task that embedded software developers face.

The first step is common to all architectures: translate the application requirements into a cycle budget. That is, determine how many cycles we have available to consume an input word or produce an output word.

After determining the cycle budget, the software developer must devise a plan to meet that cycle budget.

- With conventional DSPs, the approach consists of sharing that cycle budget across all functions in the target application. The DSP developer profiles the entire application, identifies the most time-consuming functions, and proceeds to optimize them until the entire application runs fast enough.
- With FPGAs and ASICs, the approach consists of developing and validating faster hardware, with more parallelism, function-specific operations, etc.
- MPPAs combine the benefits of both approaches:

- Like an FPGA, but unlike a conventional DSP, an MPPA gives the software developer the option of trading area for speed. This allows software developers to accelerate blocks of arbitrary complexity with no significant rework of the code. Time-consuming blocks can be parallelized onto multiple processors using functional or data parallelism. Functional parallelism consists of running different parts of an algorithm on successive processors. Data parallelism consists of running the same algorithm on independent blocks of data using different processors.
- Like a conventional DSP, but unlike an FPGA, the MPPA enables the software developer to optimize the code running on a single processor in software without the cumbersome process of developing and verifying RTL code. Code that requires a moderate increase in speed usually remains written in a high-level language, relying occasionally on intrinsics to use processor instructions when they provide a significant code speedup. Code that requires more speedup is usually written in assembly. This optimization process is simpler on an MPPA than on a high-end DSP for multiple reasons:
 - The functional code to be optimized on an MPPA is usually simpler to start with, as was explained in the section [“Allocating processors” on page 12](#).
 - Each individual processor on an MPPA is simple. As a result, the software developer doesn't have to rely heavily on optimization tricks such as code pipelining, loop unrolling, or algorithm restructuring to keep multiple execution units busy at every cycle.
 - With MPPAs, when a chain of processors executes a section of a design, only the processors that are the bottlenecks and do not meet their cycle budget need to be fully optimized. With high-end DSPs, all functions that contribute noticeably to the total cycle count must be heavily optimized.

Software developers who have spent weeks hand-tuning assembly code to save a few extra cycles in a single function will appreciate the appeal of the MPPA optimization process.

The quality of the development tools also has a key impact during the optimization phase because this is usually where most of the time is spent. The aDesigner tool set offers all conventional debugging features that are standard in the industry today. In addition, it facilitates the debugging and validation of any code being developed by letting the user insert taps on the input and output ports of any processors. This technique allows developers to collect reference input and output data to debug any single processor or quickly identify the source of a bug across a design of arbitrary complexity.

In summary, software developers can optimize MPPAs in behavioral programming languages, which are much more practical than the RTL language that is practically unavoidable for FPGA and ASIC users. At the same time, simpler target processors combined with less stringent requirements for each processor's code makes fully optimized assembly code less difficult to write and less often required than with conventional high-end DSPs.

Details of the approach taken

Determining the cycle budget

The first optimization step is to determine a cycle budget. Our primary goal for this project was to encode at average quality sixty 640x480 images a second, which, assuming a 300 MHz processor clock speed, translates into a cycle budget of 5.4 cycles per incoming color component. Since JPEG is a compression codec, the data rate decreases as we progress in the processing chain and the cycle budgets increase accordingly as illustrated in [Table 2 on page 15](#).

Table 2: Cycle budget for the encoder target

Cycle budget	
5.4 cycles	Per input byte (R, G, or B)
~9 cycles	Per Huffman code produced
~90 cycles	Per 32-bit output word

Note that the first number (cycle budget per input byte) is independent of the quantization and the image being encoded. The following two numbers vary depending on the image being encoded and the quality settings being selected. As a result, these two numbers are padded to guarantee that the encoder meets the target throughput when encoding any image at a similar quantization level.

Meeting the cycle budget

The reference design and the fully-optimized design both run in the same aDesigner environment, which allows optimization to be done progressively one step at a time.

The aDesigner's profiling tools enable us to measure how many cycles each processor takes to produce an output sample and compare these numbers to the cycle budgets above. Not surprisingly, complex blocks like the DCT that are at the beginning of the processing chain and perform a lot of computation on each pixel require the most optimization and as such must be written in assembly. Blocks like the bit string packing that come at the end of the processing chain with a large cycle budget per output sample require no modification and can remain in Java.

On MPPAs, as with conventional DSPs, some functional blocks lend themselves better than others to the parallelism available at the assembly instruction level. For example, blocks like color conversion, DCT, or quantization can take full advantage of instructions operating on half words while some portions of blocks like run-length and Huffman encoding must remain serial (i.e., processed only one sample at a time). As a result, after first-order optimization of blocks that is required in assembly or in Java was completed, we compared the cycles spent by each function to the cycle budgets described in Table 2 above.

We found that only the run-length and Huffman encode block exceeded its cycle budget significantly. It exceeded its cycle budget by about 30%.

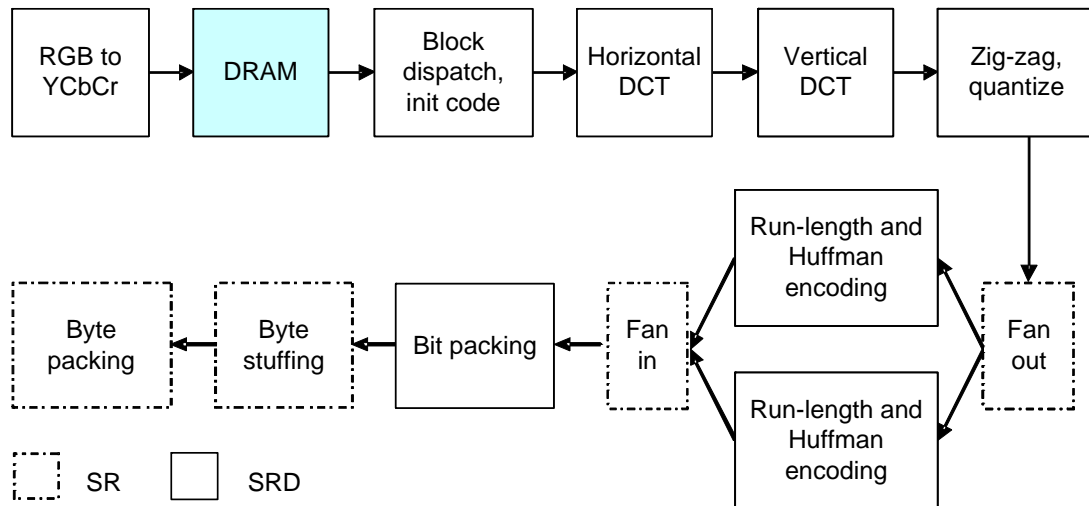
With a conventional DSP, a software developer not able to meet his or her target requirements is confronted with two choices: further optimize the most time-consuming functions that have already been optimized to a first order (i.e. save a few more cycles on functions that have a big impact on performance), or optimize more of the less time-critical functions (i.e. save a larger number of cycles on functions that have a small impact on performance). Each is too time consuming to be desirable—each small additional increase in performance comes at the expense of a large amount of work.

With an MPPA like the Am2045, the software developer can use a simpler third approach—apply data parallelism across more processors. We chose this approach for improving the run-length and Huffman encode step. Two processors are assigned to run the same run-length and Huffman program in parallel. This general approach speeds up any code of arbitrary complexity with only trivial changes to the code. In our case, this technique allowed the run-length and Huffman encode block to go from exceeding its cycle budget by 30% to outperforming by about 25%.

Resulting implementation

The resulting design fits in a few SR and SRD processors as shown in Figure 3. The design relies on a combination of Java and assembly code. Longer non-time critical code such as the initialization routines expanding the Huffman tables or those creating the JPEG file header is written in Java and shorter time-critical code is written in assembly.

Figure 3: Mapping of the entire JPEG encoder onto SR and SRD processors



Due to the simplicity of the MPPA processors, the assembly code is straightforward: the most complex block (run-length and Huffman encode) consumes about 100 assembly instructions. The second most complex blocks (horizontal and vertical DCT) run practically the same code: around 60 assembly instructions. Other assembly blocks average about 20 assembly instructions each. Normal programming practice was used to write the assembly code and no attempt was made to get unduly creative. Nonetheless, as can be seen, these numbers are much lower than what would be seen on a typical high-end DSP.

In summary, we wrote approximately 300 lines of assembly code and used only about two bricks out of 45 available to run the entire JPEG encoder and meet our application requirements. Most resources (except the SDRAM used to buffer macro blocks) are contained within these bricks with complete encapsulation. This leaves ample resources for other applications to execute in parallel on the chip without any risk of affecting the functionality of the JPEG encoder or deteriorating its performance. The same is not true with conventional DSPs, which rely on time sharing the CPU in order to run multiple applications all at the same time, an approach that adds resource switching overhead and, more importantly, affects the contents of cache, thus resulting in poorer performance when applications are combined.

The SDRAM bandwidth utilization is in general well balanced with the on-chip resource utilization (processors and local memory). This is the case with the JPEG implementation, which uses about 4% SDRAM bandwidth against the 5% utilization of on-chip resources discussed above.

In terms of processing throughput, each processor meets its cycle budget, and therefore the implementation exceeds the requirements. The bottleneck happens to be the processor packing the Huffman codes into a bit string: it consumes an average of 7.5 cycles per Huffman code being consumed. Therefore the overall JPEG encoder achieves a throughput of approximately 72 frames per second against the target of 60 frames per second.

Step 3: Executing the design on the chip

Objective

Most of the design development and optimization occurs in a simulation environment, which is typically the preferred development environment for software developers. The next step for the developer is to take a design that produces correct output on a simulator and run it on the chip to ensure consistent functionality and meet design speed constraints. This step requires the following considerations:

Fitting code into on-chip memories

Almost all code developed by the process described in the above sections is already broken down into pieces small enough to fit into the local memory of each processor. Code that does not fit into the local memory of one processor can either be split onto several processors or further optimized for space by using more function calls, when this is an option.

Fitting local data arrays into on-chip memories

The Am2045 is a non-cached architecture, which means that the software developer is in full control of which data arrays go into on-chip memory and which remain in external memory. This architecture results in deterministic performance, a key advantage over cached architectures. aDesigner automatically allocates up to four memory banks (for a total of 8 KB) to each data array or group of data arrays created by the software developer. As a result, most arrays fit readily into on-chip memory. Again, the most common exception is for processors running a large amount of non time-critical code where a large number of small arrays are typically present. When combined, these arrays may end up taking more space than is available in an RU's memory. Common solutions include reusing the same arrays for temporary calculations, performing in-place computations in which the output buffer replaces the input buffer, or using more than one processor to run the initialization code (because each processor can access a different RU's memory banks).

Placing and routing the design

Due to the large granularity of the design, the place and route task on an MPPA is considerably simpler than it is on an FPGA or ASIC. aDesigner handles the place and route by deciding which processors and local memories to use to run the application, and how to connect processors and memories according to the software developer specifications. This step takes less than a minute to complete, even for a large design that uses most of the resources on the chip.

Running the design at the expected speed

Running an application on the chip allows the developer to obtain a full and accurate characterization of processor activity that takes into account system level issues such as SDRAM latencies. aDesigner captures a number of types of profiling data measured on hardware at run-time, such as processor % utilization, channel activity, instruction profiling, etc. By displaying profiling data on a time axis using an intuitive graphical color-coded display, aDesigner allows the software developer to identify situations that may cause an application to underperform such as a processor on a time-critical path that remains inactive as a result of an insufficient buffering of data.

Details of the approach taken

The Am2045 software developer uses the same aDesigner integrated development environment to target the chip or use the simulator. aDesigner allowed the entire code and data arrays to fit readily into local memory, including the initialization routines. For example, aDesigner automatically allocated one additional bank to store the larger code that handled the block dispatch and initialization of quantization and Huffman tables as it didn't entirely fit into the processor's local instruction memory. The design automatically placed and routed in less than 5 seconds.

Running the design on hardware identified one object that had been left in Java and was too slow " the byte stuffing object. It included a time-critical loop with multiple tests that were better handled in assembly than by the compiler. After that object was optimized, aDesigner's profiling tool, which non-intrusively samples processor activity at run time, confirmed that the processor expected to be in the critical path (the object performing the bit packing) was kept busy 100% of the time. This confirmed the analysis that had been made in the simulator environment. As a final validation step, we confirmed that we were able to encode more than 60 640x480 frames in less than a second using different quantization tables leading to the desired image quality.

Programming with MPPAs: conclusion

The JPEG implementation on the Ambric Am2045 MPPA architecture shows a programmable implementation that can easily be scaled to achieve levels of performance higher than any high-end DSP and comparable to those achieved by many FPGAs.

The following summarizes the key common points or differences between creating a design on an MPPA versus using other programmable architectures targeted at high-end applications:

- **Partitioning:** Embedded designs running on conventional DSPs are usually divided into multiple functions. With MPPAs, these functions map logically onto separate processors without the overhead of function calls.
- **Behavioral language:** The availability of a behavioral software language to develop a reference implementation that closely models the final algorithm implementation is a key advantage compared to FPGAs and ASICs, which rely on the RTL language to produce a realistic model of any implementation.
- **Code safety:** Unlike other parallel architectures, MPPAs guarantee that each processor has its own local memory that cannot be affected by other processors running on the chip. All processor communication is handled safely and efficiently through self-synchronizing channels. Also, bugs inherent to C such as pointer errors disrupting unrelated code are simply not possible when the underlying programming language is Java, as is the case with the Am2045.
- **Debugging:** The process of debugging applications on an MPPA is similar to what software developers experience with conventional DSPs in that each function can be implemented and tested one at a time. When a bug occurs in a design, port taps allow a software developer to analyze quickly the streams going in and out of any processor in a design to identify quickly the source of the bug.
- **Optimization:** Optimization is easier on MPPAs than on any other conventional architecture due to the simplicity of each processor. This advantage is critical as software developers spend a lot of time writing code in assembly to meet performance requirements. In addition, the ability to use data or functional parallelism provides software developers with additional optimization options.

About the author

Laurent Bonetto is technical marketing engineer at Ambric. His working focus at Ambric is on the aDesigner comprehensive development tool suite and the Am2000 family of chips and boards. Mr. Bonetto is also involved in defining architecture instruction sets and developing application firmware at Ambric. He has expertise in a wide range of processor architectures. Previously, Mr. Bonetto worked at Cradle and, prior to that, at Berkeley Design Technology Inc. (BDTI) where he was benchmarking and analysis program manager. Mr. Bonetto received his ME degree from the Georgia Institute of Technology. He also holds undergraduate degrees in mathematics and engineering from the Electrical and Computer Engineering School, Supélec, France.