

Boosting Software Processing Performance With Coprocessor Synthesis

Abstract

The burgeoning embedded software content of system-on-chip (SoC) designs mandates significantly enhanced processing capability to meet stringent performance objectives. Consequently, many SoCs deploy additional processors—or coprocessors—that operate under the control of the central processing unit (CPU) to boost processing performance.

For example, a standard general-purpose processor core can be used as a loosely coupled coprocessor. However, it generally has insufficient parallel resources to process compute-intensive software with the requisite performance. Even when it meets performance objectives, it often consumes excessive power.

Alternatively, the SoC can deploy a custom instruction set processor (CISP), which utilizes hardware resources better optimized to execute the selected tasks, limiting the power consumption increase. However, a CISP design is time-consuming and requires processor architecture expertise. It also requires retargeting of the existing embedded software, which is especially difficult in the case of unfamiliar software that the design team wants to re-use without modification.

The Cascade Coprocessor Synthesis solution suffers from none of these limitations. Cascade automates the design of a loosely coupled performance- and resource-optimized coprocessor, without the need for processor architecture expertise. Design time is measured in days.

The Cascade solution can be used to co-optimize coprocessor architecture and software before synthesis, or simply to synthesize an optimized coprocessor that accelerates software ‘as is’, with no software retargeting. Cascade accepts user-defined performance and gate usage constraints, generates candidate architectures, and performs ‘what if’ analyses to determine the optimum architecture, prior to synthesis. Cascade works with the designer’s software development tools, and integrates easily into the designer’s established SoC, FPGA and structured ASIC design flows.

The synthesized coprocessor is a programmable engine that communicates with the CPU via the system bus. It can be designed as a slave to the CPU, or as an autonomous streaming coprocessor that uses Direct Memory Access (DMA).

Embedded Software Drives SoC Design

The embedded software content of the average SoC design accounts for more than 50% of its functionality. The software is the prime source of the device’s value-add and differentiation, while software programmability is essential to the success of a fast-turn, multi-generational derivative SoC design strategy. Consequently, the silicon implementation is now a network of processing, storage, and communications elements devised and optimized largely to deliver the embedded software’s functionality.

A survey of SoC development engineers conducted by the market research company, International Business Strategies, quantifies the situation (see Figure 1). The analysis applies only to software developed by the semiconductor manufacturer. It does not comprehend software that the system manufacturer develops to enable and differentiate the system end product. According to the survey, the SoC embedded software development effort:

- Surpassed the hardware design effort at the 130nm process technology node.

- At 90nm, exceeds the 180nm SoC hardware and software development effort combined.
- Has a compound growth rate of 71%, compared to 28% for the hardware development effort.

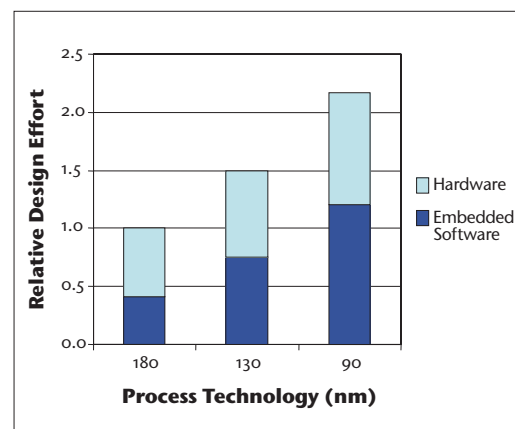


Figure 1: Relative SoC Hardware and Software Design Effort at Successive Process Technology Nodes

To process such volumes of software, advanced SoC designs deploy several microprocessors and digital signal processors (DSP). However, these resources are not always adequate. According to a survey conducted by Embedded Market Forecasters, nearly 70% of embedded system designs fail performance objectives by at least 10%, with nearly one-third failing by 50%.

Replacing the CPU with a higher performance version of the same type may solve the system performance problem—if it possesses sufficient hardware resource parallelism to exploit the parallelism inherent in the software. In the case of Reduced Instruction Set (RISC) processors—the CPU of choice in SoC design—this is often not the case when processing compute-intensive software. Even when the replacement CPU achieves higher system performance, it may consume excessive power—an unavoidable consequence of the processor’s deployment of non-optimized hardware resources that satisfy a broad range of processing requirements.

The solution is to add parallel processing resources that are optimized to exploit the parallelism inherent in the particular software to be accelerated—a coprocessor.

Coprocessor Evaluation Criteria

A coprocessor must offer higher levels of software re-usability and performance predictability than those that have been ‘acceptable’ to date, while meeting time to market objectives. The deployment of any given coprocessor methodology must be evaluated according to the following criteria:

1. Clearly, the coprocessor must deliver the requisite performance within the permissible power consumption constraints.
2. The coprocessor must be deployed without modification of software partitioning, memory architecture and communications protocols.
3. The coprocessor’s deployment costs must be tolerable. Costs include intellectual property (IP) license and royalty fees, tool license and support fees, any SoC platform redesign costs, and any software redevelopment costs.

In the case of custom-designed coprocessors, the design environment must:

1. Accept performance and hardware resource usage as input constraints. Without such constraints, designers must be experienced in the ‘what works, what doesn’t’ of processor design.
2. Offer pre-implementation ‘what if’ analyses of performance and hardware resource usage, eliminating the time-consuming characterization of multiple implementations further downstream. Without such analysis, designers must be experienced in processor design.
3. Enable co-optimization of both hardware architecture and software to achieve optimum performance and hardware resource usage.

4. Create a coprocessor that can execute legacy software ‘as is’, without the need for software retargeting. This enables the extensive re-use of third-party software, such as industry-standard algorithms and software developed by an engineering team that is no longer accessible.
5. Automatically generate the verification testbench.
6. Allow the design team to work with its established design flows and development languages.
7. Avoid the necessity of acquiring specialized expertise, such as processor design expertise.

Coprocessor Evaluation

General Purpose Processor Cores

Deployment of an additional general purpose RISC core with the same Instruction Set (IS) as the CPU is a time-honored means of adding a coprocessor—if it has sufficient parallel resources. Unfortunately, this approach is becoming ineffective as the SoC’s compute-intensive software content burgeons.

A standard DSP core offers much greater parallelism, and is consequently a popular choice in data intensive applications, such as imaging and audio/video processing. However, such a core may well be over-resourced for the task in hand, resulting in unacceptably high area overhead and power consumption.

Moreover, use of an additional core requires—at a minimum—software repartitioning, memory architecture redevelopment, and communication protocol re-design, as well as software redevelopment when the core possesses an IS that is different from that for which the software was developed. It may also incur additional IP license and royalty costs.

CISP Implementation

The design team can develop and optimize a CISP that achieves the requisite performance with fewer hardware resources than a standard core. Consequently, it may consume less power. There are two basic methodologies for developing CISP architecture and implementation, namely:

1. Configurable IP, which offers a building-block approach to processor design.
2. EDA design tools, which generate RTL from a manually written architectural description or from the Instruction Set itself.

These methodologies have a number of common attributes. Both:

- Require software re-partitioning.
- Require memory re-architecture and communication protocol redevelopment.
- Necessitate retargeting of software, i.e. no software ‘as is’ re-use.
- Do not accept performance and gate count input constraints.
- Do not offer pre-implementation ‘what if’ analysis.
- Require the developer to use the vendor’s software toolset.
- Require processor architecture design expertise.

In both cases, the designer is generally required to learn the vendor's proprietary design language, although some configurable IP solutions are C-driven. The configurable IP approach generally provides a verification testbench, but EDA techniques generally do not.

Coprocessor Synthesis

CriticalBlue's Cascade coprocessor synthesis solution solves these problems by avoiding them. It leverages the parallelism inherent in the embedded software to automatically synthesize a loosely coupled coprocessor optimized to execute that software with the desired performance, and with the minimum necessary gate count and associated power consumption. The coprocessor enables the CPU to power down during coprocessor operation, thus reducing power consumption further.

The coprocessor can be synthesized either as a slave to the CPU or as a more autonomous streaming processor that consumes less power for the same performance. Each coprocessor type can execute software other than that for which it was synthesized, and:

- Operates as a seamless extension of the CPU (see Figure 2).
- Communicates with the CPU via the system bus.
- Executes pre-compiled software directly, eliminating software repartitioning and retargeting.
- Eliminates memory re-architecture and communications protocol redevelopment.

The Cascade solution:

- Generates the coprocessor architecture and RTL implementation automatically.
- Accepts performance and gate count input constraints.
- Identifies the optimum architecture with fast pre-implementation 'what if' analysis.
- Enables co-optimization of both hardware architecture and software.
- Executes legacy software 'as is', eliminating software retargeting.

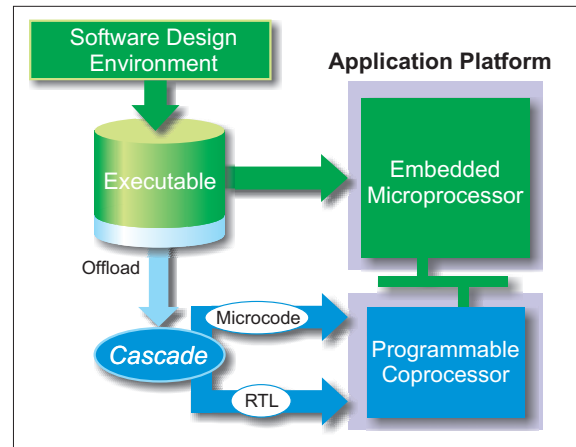


Figure 2: Cascade Coprocessor as a Seamless Extension of the CPU

- Automatically synthesizes the hardware and software interface with the system bus.
- Automatically generates the verification testbench.
- Works with the designer's established software development tools.
- Requires no expertise in either processor design or new design languages.
- Integrates seamlessly with the established RTL and FPGA design flows.

In addition to these automated attributes, Cascade has optional manual optimization capabilities that increase performance further by enabling the designer to deploy one or more custom functional hardware units that implement lower level software functions. Each unit is integrated into the coprocessor architecture and, whenever the original lower level software function is called, the equivalent custom hardware unit is invoked instead. All control aspects of the algorithm remain in the software domain, greatly simplifying hardware development. As with the automated attributes, these manual options can be explored in a 'what if' fashion, enabling the designer to determine the optimum configuration.

Table 1: Summary Of Coprocessor Attributes

EVALUATION CRITERIA	COPROCESSOR SYNTHESIS	CUSTOM IS Configurable IP	PROCESSOR EDA Tools	STANDARD PROCESSOR CORE
Software re-use 'as is', without retargeting?	Yes	No	No	Yes, if same IS
Software repartition necessary?	No	Yes	Yes	Yes
Memory architecture modifications required?	No	Yes	Yes	Yes
Communications protocol modifications required?	No	Yes	Yes	Yes
Performance and gate count input constraints?	Yes	No	No	N/A
Early performance and resource usage feedback?	Yes	Yes	No	N/A
Automatic testbench generation?	Yes	Yes	No	Provided
Processor design expertise required?	No	Yes	Yes	No
Design language expertise required?	No	Varies	Yes	No

Evaluation Summary

The primary attributes of the various solution options outlined above are summarized in Table 1.

The Cascade solution deploys the parallel processing resources of a customized processor, but at a deployment cost (as defined in the Coprocessor Evaluation Criteria section) dramatically lower than that of a coprocessor implemented by an additional processor—standard or CISP (see Figure 3).

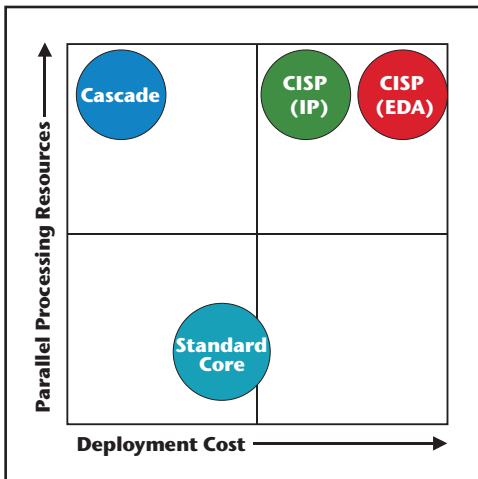


Figure 3: Relative Parallel Processing Resources and Deployment Costs Of Coprocessor Implementations

Cascade Coprocessor Synthesis Design Flow

The basic Cascade coprocessor synthesis design flow is shown in Figure 4.

Bottleneck Identification

Cascade is used initially to analyze the profiling results of the application software running on the CPU or other microprocessor and, under the control of the user, to identify the specific tasks to be migrated to the coprocessor.

Architecture Synthesis and Performance Estimation

Comprehending user-defined constraints, such as gate count, clock cycle count and bus utilization, Cascade then analyzes the instruction code—including both control and data dependencies—and automatically maps the selected tasks onto a dedicated coprocessor that is architected to deploy the maximum parallelism consistent with the input constraints. Cascade deploys data cache units to create a coprocessor memory architecture with high bandwidth memory communications. The technology provides gate count and performance estimates, including estimates of communication overhead with the CPU.

Coprocessor Performance Analysis and ‘What If’ Analysis

Cascade then generates an instruction- and bit-accurate C

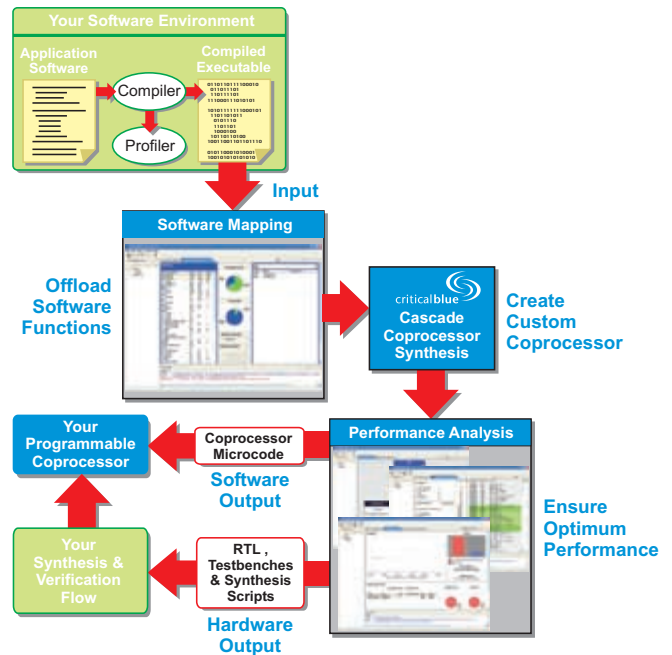


Figure 4: Cascade Coprocessor Synthesis Design Flow

model of the coprocessor architecture. This model is used in conjunction with the CPU’s Instruction Set Simulator (ISS) and the stimuli derived from the original software to undertake performance analysis, such as performance profiling, memory access activity and activation trace data. Cascade automatically analyzes the performance of all available cache architectures and selects the optimum configuration. Similarly, it identifies instruction and cache ‘misses’ early in the design flow, avoiding the performance surprises so common at system integration. The model is also used to validate the coprocessor within a system level simulation environment. The technology can quickly generate multiple candidate coprocessor architectures, enabling rapid ‘what if’ trade-off analysis of performance, functionality, area, and development time, using actual performance data.

Hardware Synthesis

Cascade then generates the coprocessor hardware, delivering synthesizable RTL code in either VHDL or Verilog. Cascade also generates the circuitry necessary to enable the coprocessor to communicate with the CPU’s bus interface, thus circumventing the hardware integration problems associated with the use of additional processors.

Testbench Generation

The technology automatically generates a testbench that is used to verify the coprocessor implementation using the same stimuli and expected responses as those of the CPU, ensuring functional equivalence.

Microcode Generation

Cascade simultaneously generates coprocessor microcode, automatically modifying the original executable code so that function calls are automatically directed to a communications library, which manages coprocessor handoff and communicates parameters and results between the CPU and the coprocessor. Microcode can be generated independently of the coprocessor hardware, allowing modified executable code to be targeted at an existing coprocessor design, albeit with some potential performance loss.

This basic flow can be utilized in two ways, as shown in Figure 5. The first flow shows the general case, in which coprocessor architecture and software are co-optimized for optimum performance and hardware resource usage. The second flow shows a constrained version of the general case, in which a coprocessor is designed to accelerate legacy software ‘as is’, without software optimization or redevelopment.

As previously noted, the Cascade coprocessor can be synthesized either as a slave to the CPU or as a more autonomous streaming processor that consumes less power for the same performance.

A streaming coprocessor utilizes fewer hardware resources and, consequently, consumes less silicon area. The primary source of resource reduction is Direct Memory Access (DMA), which simplifies main memory interaction and, therewith, the bus interface logic. It also eliminates the need for

coprocessor cache control logic. Using DMA, streaming memory units enable direct data transfer to and from the system memory, with no CPU intervention. The streaming memory units provide efficient communication by automatically deploying buffer and pre-fetch capabilities. The synthesis of a streaming coprocessor requires adherence to some common, non-limiting coding guidelines to enable the use of non-coherent memory caching units and to simplify interaction with the CPU.

Customer Project Examples

The results of five customer projects—selected to reflect the wide range of applications that have been successfully implemented with Cascade—are shown in Table 2 on the next page.

In each case, Cascade met or exceeded the customer’s acceleration requirement. Two of these examples will be discussed. Both demonstrate the efficacy of using custom functional hardware unit, although many projects may not actually require such units.

Real Time Image Processing

The project entailed the acceleration of two computationally intensive tasks: image defect correction and aperture correction using color interpolation. Executing these tasks with the customer-specified performance would require an ARM9 processor to execute 225 instructions per pixel at 30 frames per second (fps),

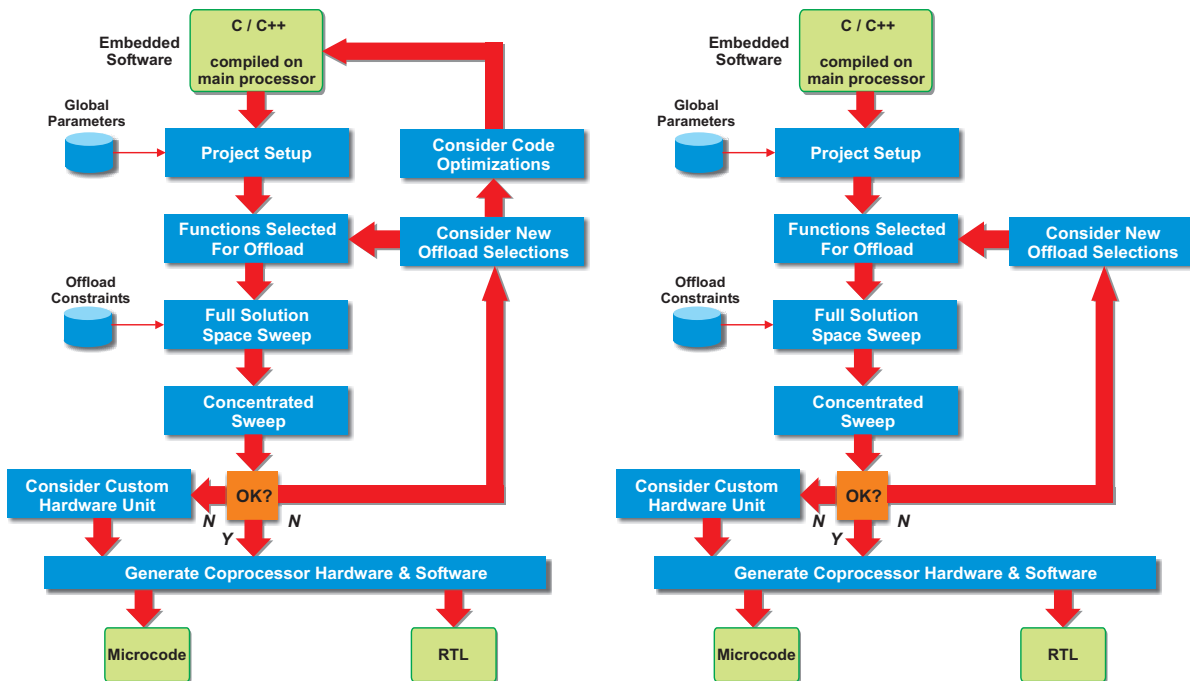


Figure 5: Cascade’s Alternative Optimization Flows

Table 2: Customer Project Results

ATTRIBUTE	REAL TIME IMAGE PROCESSING	WIRELESS ERROR CORRECTION ALGORITHM	ENCRYPTION ALGORITHM	MPEG-4 ADAPTIVE FILTER	AUDIO PROCESSOR IMPLEMENTATION
Lines of C Code	~200	~600	~120	~100	~12,000
Acceleration vs. ARM9 @ same T_{CLK}	5x/15x	4.9x	5x	4.5x	>90% CPU load reduction
Design Time	2/10 days	5	2	4	40

requiring a clock speed of 2.1GHz. In practice, both tasks are often implemented in dedicated hardware, requiring many engineer-months of design and verification effort.

Using the original software code, a Cascade processor achieved an acceleration of 5x, which delivered an execution rate of 15 fps. With slightly modified code and with a custom functional hardware unit, the Cascade processor achieved an acceleration of 15x, which increased the execution rate to 45fps.

Wireless Error Correction Algorithm

The example is that of the BCH3.c triple error correction encoder/decoder algorithm, described in ~600 lines of portable application C code. This algorithm presents many obstacles to parallelism extraction, including nested loops, complex conditionals, arbitrary pointer dereferencing, and variable strides through arrays. Cascade automatically identified four inner loop kernels that consumed 85% to 95% of the algorithm’s execution time on an ARM9. For instance, one of the decode loops requires between 1,024 and 16,384 executions, consuming up to 35 clock cycles.

Cascade synthesized a coprocessor to execute the critical path loops, while a custom functional unit was designed to execute the non-critical path arithmetic operations. The coprocessor accelerated the decode loop by a factor of ~9, while the overall algorithm execution performance was increased by a factor of 4.9. A one-line code modification would have increased the acceleration by a further factor of 1.7, but code optimization was specifically excluded by the customer. The total design effort was ~5 engineer-days, including ~2 engineer-days for the custom functional unit.

A Note On Hardware Acceleration

Design planning comprehends analysis of both required technical characteristics, such as performance and power, and commercial considerations, such as design time/effort and re-usability. This analysis dictates the criteria for HW/SW partitioning, i.e. the partitioning of the algorithms between (programmable) processors and faster, lower power non-programmable dedicated hardware accelerators. For instance a complex algorithm such as MPEG 4, which uses ~20,000 lines of code, may be realized

with the very high performance part of the algorithm implemented in one or more dedicated hardware accelerators, with the rest in embedded software. Thus the two implementation types are complementary, and should not be perceived as substitutional or mutually exclusive. In fact, the custom functional hardware units that may be deployed in a Cascade coprocessor are themselves dedicated hardware engines, albeit generally with simple architectures.

Conclusion

Coprocessor Synthesis automatically synthesizes a coprocessor that executes the software with the requisite performance and with the minimum necessary resource usage. It deploys the parallel processing resources of a customized processor, but at a deployment cost dramatically lower than that of a coprocessor implemented by a standard or CISP processor. Cascade enables simultaneous optimization of hardware architecture and software, or simply creates the optimal architecture to accelerate legacy code, which is particularly important in the deployment of unfamiliar software. No new tool chains are introduced, and no new languages must be learned. Cascade is a non-intrusive process that fits within the standard tools for existing software, system level and hardware implementation flows.



US: +1 408 467 5091
 UK: +44 131 524 0080
 www.CriticalBlue.com

*©Copyright CriticalBlue, June 2005
 CriticalBlue, the Critical Blue logo and Cascade
 are trademarks of CriticalBlue Limited. All other
 trademarks are the property of their respective
 owners.*