

Acceleration Of A Standard Algorithm Within Power Constraints Using A Cascade-Synthesized Coprocessor

Introduction

This application note describes how Cascade coprocessor synthesis was used to design a programmable coprocessor optimized to accelerate a standard compute-intensive algorithm – the BCH3.c triple error correction encoder/decoder algorithm – by a factor of ~5 over an ARM9, within the user-specified gate count and power consumption constraints, and without software re-development. Cascade automatically optimized cache memory design and bus communications overhead to maximize system performance. The project was completed in five days, and required no processor design expertise.

Design Requirements and Challenges

The BCH3.c algorithm corrects transmission bit errors caused by a 'lossy' environment. It is deployed in very high data rate communication systems such as Synchronous Optical NETwork (SONET) and Asynchronous Transfer Mode (ATM) networks, in both wired and wireless communications.

The basic customer requirement was to accelerate the BCH3.c algorithm – which consists of ~600 lines of generic, processor-independent application C code – by a factor of 4x to 5x, within specified power constraints. The algorithm's compute-intensive nature mandates the deployment of greater parallel processing resources than those available in a general purpose processor core, but contains obstacles to parallelism extraction, such as nested loops, complex conditionals, arbitrary pointer dereferencing, and variable strides through arrays. The customer specifically excluded software code modification in order to (a) maintain its re-usability and (b) eliminate any need to acquire expertise in the design of this particular algorithm.

Cascade Solution Overview

Cascade coprocessor synthesis generates and optimizes a coprocessor that accelerates software

offloaded from the main CPU, in a matter of days. The coprocessor deploys the parallel processing resources normally associated with more design-intensive custom processors, but requires no processor design expertise. The coprocessor can be reprogrammed with modified or new software, and it supports multimode operation, whereby multiple algorithms may be executed in series.

Cascade requires no compiler, and supports the continued use of the established main CPU and its associated investment in design tools and infrastructure. It requires no new language expertise, and its RTL output can be verified and debugged with the design team's established tools.

Cascade's synthesis methodology supports two use cases. Utilizing user-defined performance requirements and resource constraints, the designer can either:

- Co-optimize the coprocessor architecture and software to maximize overall system performance, or
- Optimize a coprocessor to accelerate legacy software 'as is'.

In both cases, Cascade maximizes system performance by:

- Enabling optimal software partitioning.
- Automatically optimizing cache design with data pre-fetch capability to minimize memory/system latency, and
- Automatically minimizing bus communication overhead.

Cascade Design Flow

Cascade identifies execution bottlenecks and exploitable parallelism candidates by using the compiled binary executable software code to execute

representative data sets, and analyzing the machine code. Using these results, Cascade synthesizes an initial coprocessor architecture consisting of customized control logic and datapaths. It then explores the design space to make optimal performance and gate count trade offs by iteratively allocating additional processing and connectivity resources to the bottlenecks, and measuring the outcomes.

Cascade automatically generates microcode, synthesizable RTL with synthesis scripts, an instruction- and bit-accurate C functional model, and a testbench that verifies the implementation with the same stimuli and expected responses as those of the CPU, ensuring functional equivalence. The RTL implementation then proceeds through the designer's own SoC, FPGA or structured ASIC design and verification flows. The coprocessor design is shown in figure 1.

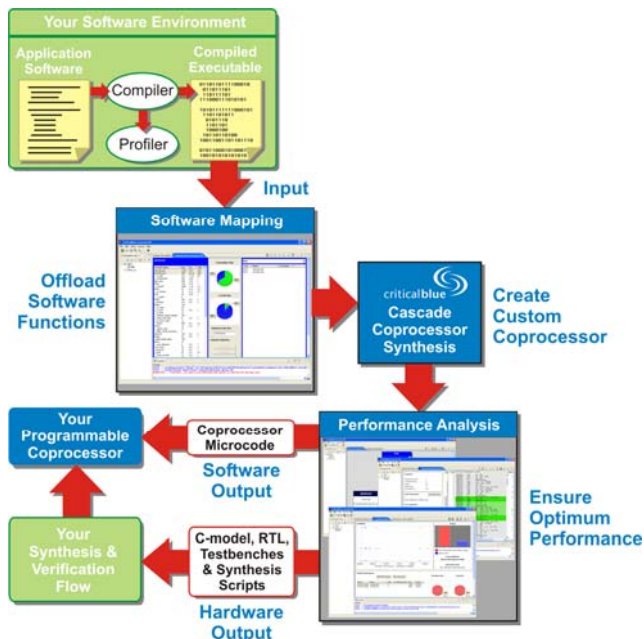


Figure 1: Coprocessor Synthesis Design Flow

Bottleneck Identification

Cascade is used initially to analyze the profiling results of the application software running on the main CPU and, under user control, to identify the specific routines to be executed on a 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 routines onto a coprocessor that deploys the maximum parallelism consistent with the input constraints. Cascade provides gate count and performance estimates, including estimates of communication overhead with the main CPU.

Performance Analysis and 'What If' Analysis

Cascade then generates an instruction- and bit-accurate C model of the coprocessor architecture. This model is used with the main 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. Automatic analysis and simulation identifies instruction and cache 'misses' early in the design flow. The model is also used to validate the coprocessor within a standard C or SystemC simulation environment. Cascade quickly generates multiple candidate coprocessors, 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 as synthesizable RTL code in either VHDL or Verilog, which the designer then verifies using the same stimuli and expected responses as those used by the main CPU. Cascade generates the circuitry necessary to enable the also coprocessor to communicate with the main CPU's bus interface, thus eliminating the hardware integration problems associated with the use of additional processors.

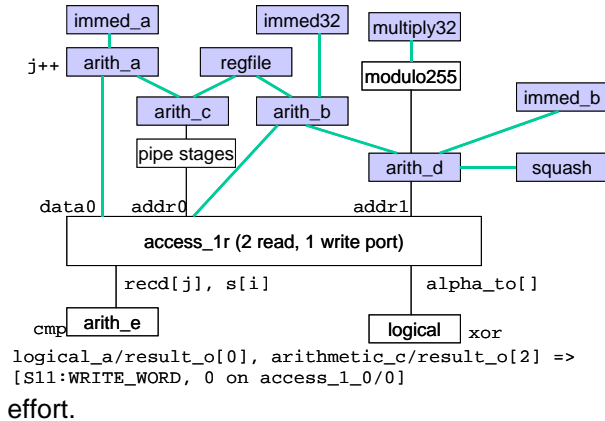
Microcode Generation

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 main CPU and the coprocessor. Microcode can be generated independently of the coprocessor hardware, allowing new microcode to be targeted at an existing coprocessor design.

Interactive Capabilities

Cascade supports manual intervention. For instance, the designer can undertake manual code optimization within Cascade and can deploy custom functional hardware units to achieve even greater processing

performance. These manual options can be rapidly explored in a 'what if' fashion, enabling the designer to determine the optimum configuration with minimal



The BCH3.c Design Project

The algorithm defines two primary functions: encode_bch and decode_bch. Profiling confirmed the designer's initial judgment that four inner-loop routines – DEC1, DEC2, ENC1 and ENC2 – would consume 85 % to 95% of the algorithm's execution time, even though they represent only ~3% of the total code. The algorithm description for DEC1 (below) demonstrates why:

```
for (j=0; j < length; j++)
  if (recd[j] != 0)
    s[i] ^= alpha_to[(i*j)%n];
```

The length of the message varies from 64 bits to 1,024 bits, requiring 64 to 1,024 loop executions. The loop is nested within another loop that comprehends the desired error correction factor, n. In this case, n = 16, so the total number of loop executions necessary to complete DEC1 varies from 16*64 = 1,024 to 16*1,024 = 16,384. These serial operations consume 35 clock cycles on an ARM9 core.

Cascade was used on the critical path s[i] loops, and a custom functional unit was used to execute the non-critical path (i*j)%n arithmetic operations. The coprocessor functional block diagram is shown in figure 2.

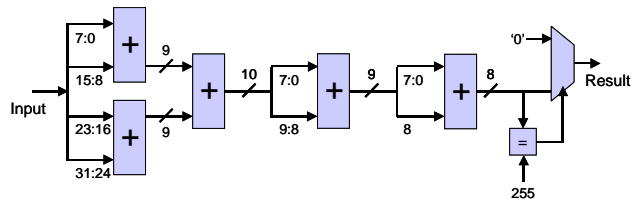
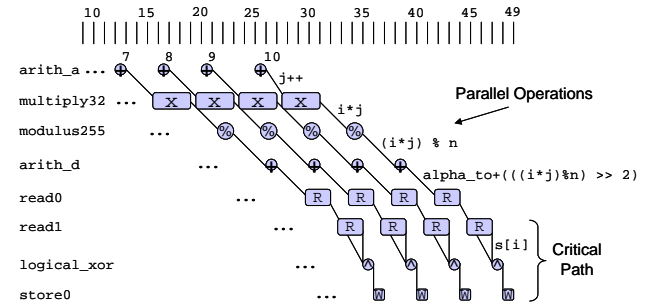


Figure 2: BCH3.c Acceleration Functional Block Diagram Using Coprocessor Synthesis

The ten colored functional blocks execute simultaneously, demonstrating the degree of



parallelism achieved by the approach. The introduction of so much parallelism not only accelerates the algorithm execution, but also reduces by an order of magnitude the power consumed by registers and buses.

The coprocessor synthesis technology automatically generated all blocks except the 'modulo255' block (see figure 3). This simple block was manually designed and verified in less than two days.

Figure 3: Custom Functional Block

The execution of the DEC1 algorithm over time is shown in figure 4, which details four of the ten loop unrolls performed by Cascade.

Figure 4: DEC1 Execution vs. Clock Cycles

It can be seen that the critical path operations – which retain their loop dependencies – execute in 4 cycles, a nearly 9x acceleration over the 35 cycles used by the ARM9. This execution time can be accelerated further by a factor of ~1.7, using a one-line code optimization, but software modifications were specifically excluded from this design project.

Function	Execution Time (cycles)		Acceleration
	ARM9™	Coprocessor	

Encode	550,233	125,926	4.37
Decode	1,430,294	280,432	5.10
Total	1,980,527	406,358	4.87

Table: Coprocessor Acceleration of BCH3.c vs. ARM9

Overall Results

The table compares the performance of 256-bit message execution on an ARM9 and on the Cascade coprocessor, operating at the same clock frequency. The whole design consumed 5 engineer-days of effort.

Conclusion

Cascade synthesized the optimized coprocessor necessary to accelerate the compute-intensive algorithm within gate count and power consumption constraints. The project required no processor design expertise and no software re-development.



US: +1 408 573 3609

UK: +44 131 524 0080

©Copyright CriticalBlue, July 2007. CriticalBlue, the CriticalBlue logo, and Cascade are trademarks of CriticalBlue Limited. All other trademarks are the property of their respective owners.