Co-processor Synthesis: A New Methodology for Embedded Software Acceleration

Dr Ben Hounsell, Co-founder; Richard Taylor, CTO
Critical Blue Ltd, SMC, West Mains Road Edinburgh EH9 3JF, UK
ben.hounsell@criticalblue.com; richard.taylor@criticalblue.com

Abstract

This paper introduces co-processor synthesis – a methodology that provides design benefits by implementing hardware co-processors directly from embedded software. The paper examines the design benefits in this new approach vs behavioral synthesis and configurable processor methodologies.

1. Introduction

Embedded microprocessors form the heart of today's electronic products, but frequently lack the processing performance to support many of the functions design engineers need to implement in software. As a result design teams resort to implementing key functions as hardwired accelerators, at the cost of discarding the inherent flexibility afforded by a software implementation.

By incorporating the co-processor synthesis design methodology (Figure *1*) into the EDA tool flow, dedicated co-processors can be synthesized to accelerate software tasks that might otherwise require manual hardware design.

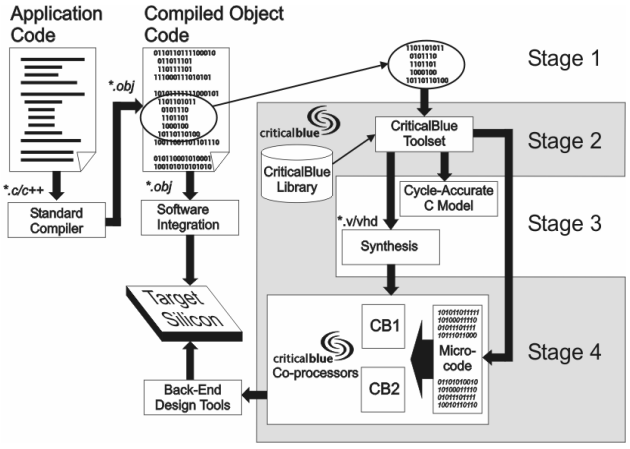

Figure 1. Cascade's Design Flow.

To ensure seamless integration with current EDA tools and file formats, co-processor synthesis is implemented as a point tool, Cascade, that sits above the current EDA flow. Cascade is not bound to a particular design language or set of third party tools as it operates on compiled executable code targeted at the main processor. By extracting parallelism from compiled object code Cascade leverages standard embedded software languages such as C and C++, enabling users to develop dedicated co-processors using their existing software development environments.

2. Automatically Customized Co-processor

The Cascade methodology starts with the premise that the execution time of an application is dominated by the execution of a few key functions or loop kernels that represent only a small percentage of the total application code. The main processor is designed for general purpose code execution and its hardware is not specialised for any particular type of code. Typically, key functions and loops will contain significant instruction level parallelism that can be exploited by a processor with appropriate execution resources. Even superscalar and VLIW based general purpose processors can typically only exploit a limited quantity of the parallelism available to the architecture.

The philosophy of co-processor synthesis is to analyse compiled executable code, along with a detailed instruction level trace captured by the tool, and map the software onto a co-processor with the execution and connectivity resources that reflect the code requirements. Whilst providing significant instruction level parallelism, the co-processor is relatively lightweight in its control logic overhead. The Cascade tool is able to balance temporal and spatial computation in the architecture depending on the inherent code parallelism available and user constraints. Extracting parallelism from code, either at an object or source level, but the co-processor contains low overhead speculation mechanisms to reduce the impact of serialisation caused by memory address aliasing considerations.

The methodology does not attempt to replace the main processor, thus minimising system design disruption and allowing the co-processor to avoid much of the infrastructure overhead of general purpose processors. Instead, the co-processor is dedicated to a particular task, but with much of the flexibility implied by a software implementation.

3. Co-processor synthesis design methodology

We implement the methodology in four steps. In the first step the compiled application software is analyzed using standard profiling tools. This process aids designers in

1530-1591/04 $20.00 (c) 2004 IEEE

identifying software functions that would benefit from acceleration.

Once the software functions are identified, Cascade analyses their instruction code and automatically maps the chosen functions onto a dedicated co-processor that has been architected to extract the maximum parallelism. Analysis is performed to extract both the control and data dependencies between instructions. At the end of this second step information is provided to the user about the estimated performance of the co-processor. This includes estimations of communication overhead with the main processor.

In the third step Cascade produces a cycle and bit accurate C model of the co-processor designed in step 2. By using the model in the context of a system level design environment, the user is able to understand the implications of offloading selected software functions within the context of the overall design. Users are able to perform rapid "what if" analysis with very quick turnaround.

Once satisfied with the co-processor's performance, an RTL form of the co-processor can be generated for simulation and synthesis using standard EDA tools. In this fourth and final step the co-processor microcode is generated. Microcode can be generated independently of the co-processor hardware, allowing new microcode to be targeted at an existing co-processor design. The original executable is modified automatically so that calls to the offloaded functions are automatically vectored to a communications library. This causes automatic handoff to the co-processor, passing parameters and results automatically between the processor systems.

4. Comparing design methodologies

Co-processor synthesis is based on the premise that software targeted at the main processor will form the starting point of any co-processor implementation; e.g. the less modification of this software, the lower the design risk. This is a divergence from the two most popular methodologies for implementing acceleration blocks at a high level of abstraction.

Behavioural Synthesis facilitates more rapid development of dedicated hardware by using a higher level of abstraction than RTL. Unfortunately, certain restrictions limit the expressiveness of behavioural constructs relative to mainstream software languages.

Behavioral synthesis tools generate hardware that is a mapping of the designer's description. As such the resulting hardware is fixed and non-programmable. In many of today's products it is important that software reprogrammability is provided within as many system functions as possible to provide end product flexibility. This is vital in broadening the chip's potential application domains and extending the life of the design.

Configurable Processors provide an effective design methodology that enables designers to add specific instructions to meet the needs of demanding operations in a given application. However, providing this flexibility introduces new tools and hardware implementation languages into the design process.

When used in the context of a co-processor, the configurable processor methodology forces the developer to deal with the complexities of multiple software development environments and explicit communication between independent processors. The alternative is to port the entire software application from the original processor onto the configurable microprocessor – a significant overhead when only a small set of software functions require acceleration.

5. Interfacing with the main processor

Hardware developed through co-processor synthesis is architected to communicate directly with the bus interface of the main processor; for example AMBA or CoreConnect. Issues of cache coherency between the processors are handled within the communications libraries, supported by specialized hardware within the co-processor. Thus the communication between the main processor and the co-processor is seamless to the user.

6. Derivative design

When looking to develop derivative products, processor reuse is vital. This is especially true for design teams wishing to add functionality to existing products where the main microprocessor is already heavily loaded and does not have sufficient remaining execution cycles to support new tasks. In this context co-processor synthesis provides designers with the opportunity to develop new functionality in software on the main processor, with the knowledge that is can be subsequently offloaded and accelerated through co-processor synthesis.

7. Conclusion

The co-processor synthesis methodology presented represents a step change in the ability for designers to efficiently identify, offload and accelerate key software functions within an application which requires software flexibility, but also greater performance than can be provided by the main processor. The approach is not bound to a particular implementation language or set of third party design tools. Instead, by statically extracting parallelism from executable code existing software development environments can be utilised and co-processor integration is seamless from a software perspective. These benefits are unique to co-processor synthesis, but remain limitations of the behavioural synthesis and configurable processor design methodologies. The benefits of co-processor synthesis have already been demonstrated through a pilot project undertaken with ST Microelectronics.