JPEG Encoding on the Intel MXP5800: A Platform-Based Design Case Study

Abhijit Davare
EECS, UC Berkeley
davare@eecs.berkeley.edu

Qi Zhu
EECS, UC Berkeley
zhuqi@eecs.berkeley.edu

John Moondanos
Intel Corporation
john.moondanos@intel.com

Alberto Sangiovanni-Vincentelli
EECS, UC Berkeley
alberto@eecs.berkeley.edu

Abstract

Multimedia systems are becoming increasingly complex and concurrent. The Platform-based design (PBD) methodology [8] tackles these issues by recommending the use of formal models, carefully defined abstraction layers and the separation of concerns. Models of computation [10] (MoCs) can be used within this methodology to enable specialized synthesis and verification techniques. In this paper, these concepts are leveraged in an industrial case study: the JPEG encoder application deployed on the Intel MXP5800 imaging processor. The modeling is carried out in the Metropolis [4] design framework. We show that the system-level model using our chosen model of computation allows performance estimation within 5% of the actual implementation. Moreover, the chosen MoC is amenable to automation, which enables future synthesis techniques.

1 Introduction

Data streaming systems such as audio, video, and image codecs as well as wireless communication will be the driving force behind consumer technology in the near future. With increasing system requirements and decreasing development times, the RTL design abstraction is no longer sufficient, a higher level which allows for rapid design space exploration must be used. High levels of abstraction not only shorten verification time, but also favor design reuse, formal verification, rapid prototyping and last but not least, the usage of synthesis techniques.

However, using a higher level of abstraction requires choosing an appropriate model of computation to describe, model, and reason about the system. At the RTL level, a synchronous MoC with combinational blocks being described by Boolean expressions is sufficient to handle the majority of designs. At the system level, the landscape is much more fragmented. No single MoC can satisfy most of the requirements. This implies that the modeling framework itself needs to support various models of computation.

The Platform-based design methodology advocates the use of formal models of computation, precisely defined abstraction layers and the separation of concerns to facilitate system level design. The separation between application and architecture is central to this methodology. The application includes the portion of the design that is implementation-independent and specifies which operations need to be carried out. The architecture, on the other hand, specifies how the operations will be carried out and at what cost. The definition of an operation and the rules for composing these operations are captured by the MoC – or the common semantic domain between application and architecture. Design space exploration is seen in this framework as the procedure of carrying out different mappings between the application and architecture models and evaluating the quality of implementations that are derived.

As with any other methodology, the proof of its power can only come applying it to real-world case studies. In this paper, we take a system from the multimedia domain and apply the design methodology with the help of the Metropolis framework, which is based on PBD principles.

2 The Case Study

The application for the case study primarily involves soft real-time requirements and data streaming while the chosen architecture is heterogeneous and highly concurrent.

2.1 JPEG Encoder Application

The JPEG encoder [13] application, is required in many types of multimedia systems, from digital cameras to high-end scanners. The application compresses raw image data and emits a compressed bitstream. This application was

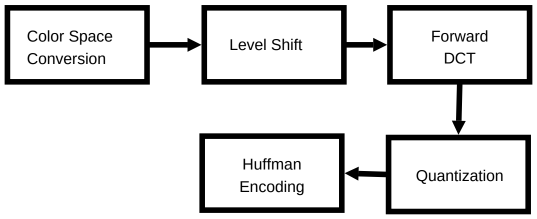

Figure 1. JPEG encoder block diagram

chosen since it is representative of a wide class of multimedia applications. In particular, the DCT, quantization, and Huffman blocks in the JPEG encoder algorithm are utilized in several video compression algorithms including the H.264 standard [15].

A block diagram for the JPEG encoder is shown in Figure 1. The input for the application is a stream of raw RGB data. In color space conversion, the raw data is first converted into YCbCr format, where each of the three components is a single unsigned byte.

Next, each of the component values is level shifted such that it can be stored as a signed byte. The values are then bundled into 8x8 blocks and processed independently. First, each 8x8 block passes through a forward integer DCT block.

The subsequent step in the algorithm is quantization. Each component in each 8x8 block is divided by a user-supplied coefficient from a quantization table. Two separate tables are used, each with 64 coefficients, one for the luminance components and the other for the chrominance components.

After the division has taken place, the next step is to rearrange the component values within each 8x8 block from row-major into zig-zag order. This ordering tends to group the higher frequency components together, preferably leading to long sequences of zeros.

The first part of the Huffman encoding step is run-length compression which takes long strings of zeros and represents them in a concise intermediate form. The second stage is the actual Huffman table lookup, which translates the intermediate form into compact bit sequences. Like the quantization tables, the Huffman tables are statically specified by the user.

The final JPEG image file consists of header data along with the compressed bitstream. The header data includes the quantization and Huffman tables for both the chrominance and luminance components.

2.2 The Intel MXP5800 Platform

The Intel MXP5800 digital media processor is a heterogeneous, programmable processor optimized for document image processing applications. It implements a data-driven, shared register architecture with a 16-bit data path and a core running frequency of 266 MHz. The MXP5800 provides specialized hardware to accelerate frequently repeated image processing functions along with a large number of customized programmable processing elements.

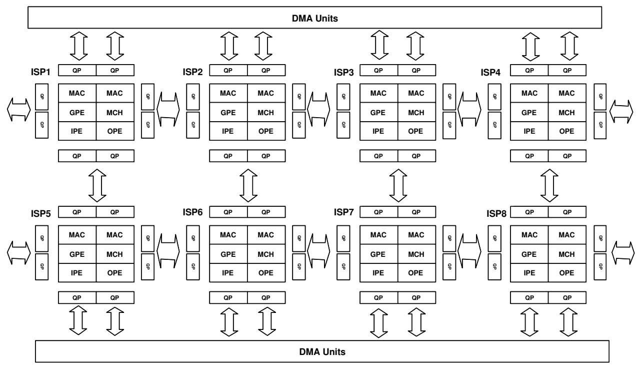

Figure 2. Block Diagram of MXP5800

The basic MXP5800 architecture, shown in Figure 2, consists of eight Image Signal Processors (ISP_1 to ISP_8) connected with programmable Quad Ports (8 per ISP). Quad Ports are used for data I/O and are essentially FIFOs of size 2 each. They provide blocking read and write semantics which ensures that all communication is data driven. Quad Ports are statically configured by the application developer according to the data flow topology of the application. In addition to Quad Port connections, various ISPs are connected to other units such as DMA channels and expansion ports.

Each ISP consists of five programmable Processing Elements (PEs), instruction/data memory, 16 16-bit General Purpose Registers (GPRs) for passing data between PEs and up to two hardware accelerators for key image processing functions. Two of the PEs are used for Data I/O: The Input PE (IPE) which is used to read data from the Quad Ports, and the Output PE (OPE) for writing data to a Quad Port. Of the remaining 3 PEs per ISP, one is for general purpose use (GPE) and two PEs have Multiply/Accumulate (MACPE) capabilities in addition to the general purpose functionality.

Each general purpose register in an ISP has a set of 8 data valid (DV) flags - one per PE. If all the DV flags for a register are cleared, a PE may atomically write data to the register and set the DV flags for all of the destination PEs. Each of the destination PEs can clear its own flag when it reads the data. In this way, the global registers serve as a single-place blocking-read, blocking-write FIFOs for multiple writers and readers. A Memory Control Handler (MCH) provides the interface to the SRAM data memory block and has support for a number of different read/write modes which support variable offsets and stride lengths.

Each ISP is optimized for a particular task and the hardware accelerators in the ISP reflect that optimization. ISP_2, ISP_5 and ISP_6 each have variable-tap and single-tap triangular filters. ISP_4 and ISP_8 contain Huffman encode/decode engines that are useful for many compression/decompression applications. ISP_3 contains G4 encode/decode blocks. ISP_7 contains 8x8 DCT/iDCT hardware. Finally, ISP_1 has an additional 16 KB of data SRAM instead of a hardware accelerator.

The major characteristic of this architecture platform is the extremely high degree of parallelism and heterogeneity. Harnessing the flexibility of the PEs to extract high performance is the main design challenge.

3 Choosing the Model of Computation

To apply the platform-based methodology, it is first necessary to choose the appropriate model of computation to describe both the application and architecture. The MoC will address many aspects of the design flow. It will dictate how the application and architecture modeling is carried out and which synthesis techniques can be applied. In choosing a MoC, there is a tension between modeling requirements on one hand and the accuracy and strength of claims on the other. For instance, if communication and computation requirements are simply estimated, then deadlock cannot be ruled out, and significant discrepancies may exist between actual and estimated performance.

3.1 Kahn Process Network variants

Since the application is data-streaming in nature and will be mapped onto a highly concurrent architecture, a Kahn Process Networks [7] (KPN) representation is natural. In KPN, a set of processes communicate via one-way FIFO channels. Reads from channels block when no tokens are present – processes cannot query the channel status. However, this model is Turing-complete, so the scheduling and buffer sizing problems are still undecidable. Therefore, occurrences of artificial [6] deadlock cannot be ruled out if the generic KPN model is used.

The solution is to consider refinements of the generic KPN model that allow stronger statements to be made while restricting expressiveness. One possible refinement is to use dataflow [9], which separates the execution of each process into atomic firing actions. However, many key properties are still undecidable in a general dataflow model.

At the other end of the spectrum, we can consider well-behaved refinements of KPN that excessively restrict expressiveness. Homogeneous and static dataflow [11] are two such examples. In homogeneous dataflow, each process consumes and produces the same number of data tokens on each firing, for all channels. In static dataflow, the number of tokens produced and consumed must be constant for each firing on all channels, but can vary between channels. Cyclo-static dataflow [5] is a slight generalization that permits the firing characteristics of each process to vary in a cycle. The main advantage of a cyclo-static dataflow model is reduced buffer size requirements.

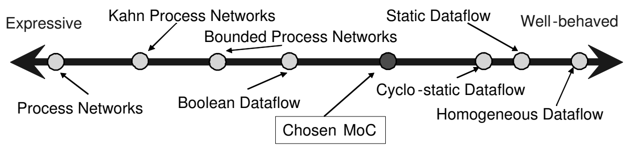

Figure 3. Classification of MoCs

A classification of these MoCs along an axis of expressiveness vs. strength of properties that can be proved is shown in Figure 3. The approximate location of the MoC that we have chosen is also shown in the diagram.

3.2 Chosen model of computation

The model of computation we have chosen is most similar to cyclo-static dataflow [5], but contains several extensions. Only one writer is permitted per channel, but multiple reader processes are allowed. For all channels, each reader process can read each data token exactly once. Also, we allow limited forms of data-dependent communication.

Like other dataflow models, processes in this MoC consume and produce tokens according to firing rules. Multiple firing rules can be specified for each process. Each process cycles between its firing rules in a fixed pattern. If a data-dependent number of tokens is to be exchanged on a channel, the sender is required to first indicate how many such tokens will be sent in a “header” token. In this way, the property of effectiveness [6] is guaranteed.

To enable support for executing multiple processes on a single processing element, this MoC has support for multitasking. In particular, a process may only be suspended between firings.

Scheduling, buffer sizing, and mapping are decidable problems for this MoC. Like many specialized dataflow models, our dataflow model induces stronger constraints on the application as opposed to the architecture. In fact, many dataflow models can be supported by multiprocessor architectures that allow efficient blocking read and blocking write operations.

4 Modeling the Case Study and Carrying out Design Space Exploration

Design space exploration for this case study is carried out in the Metropolis Framework. In this section, we begin with a short introduction of Metropolis, describe how

Implementation	Language	Concurrency	Lines of Code
IJG	C++	Sequential	18,000
MXP5800 Library	ASM	Concurrent	915
Metropolis	MMM	Concurrent	2,695

Table 1. JPEG encoder models

we modeled the application and the architecture within this framework, and present our approach to design space exploration.

4.1 The Metropolis Design Framework

The Metropolis Design Framework is an embodiment of the Platform-based design methodology. The Metropolis framework consists of a specification language – the Metropolis Metamodel – as well as a compiler, simulator, and a set of plugins that can interface with external tools. The Metamodel specification language is generic enough to encompass many different models of computation including discrete-event, synchronous reactive, and KPN.

There are three kinds of objects in Metropolis – processes, media, and quantity managers. Processes possess their own threads of control while media are passive objects that provide services to processes and other media. Processes generate events when they execute, these events can be annotated with various quantities of interest by quantity managers [3]. Examples of quantities include time, power, or access to a shared resource. Events may be related by using declarative constraints in Linear Temporal Logic [12] or the Logic of Constraints [2].

4.2 JPEG Application Modeling

Starting from both a sequential C++ implementation [1] and the concurrent assembly language implementation provided in the Intel MXP5800 development kit, we assembled an architecture-independent model of the JPEG encoder in Metropolis expressed in our dataflow model. The model carries out a full implementation of the 4:4:4 JPEG encoder baseline algorithm. A total of 20 FIFO channels and 18 separate processes are used in the application model. Characteristics of the original C++, assembly, and Metamodel designs are provided in Table 1.

To describe the application model in further detail, we will concentrate on the breakdown of the discrete cosine transform step in the algorithm. At the top level, the required two-dimensional DCT can be broken down into four basic steps: two one-dimensional DCT operations, each followed by a transpose operation. Each of the 1D-DCT block reads in 8 spatial data values (corresponding to either a row or a column) and outputs 8 frequency values. The algorithm used to carry out the 1D-DCT is based on the implementation given by Chen-Wang [14]. Each one dimensional DCT

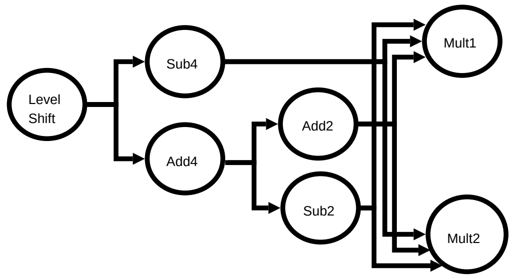

Figure 4. Dataflow model for 1D-DCT

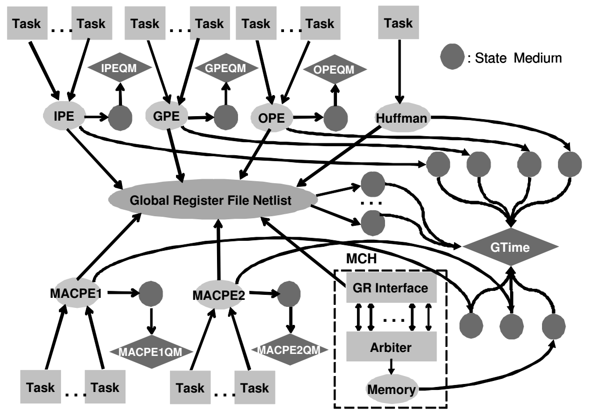

Figure 5. MXP5800 ISP Modeling in Metropolis

operation is broken down into concurrent steps, as shown in Figure 4. The diagram shows that all the channels require multiple readers. This observation, along with the fact that the architecture supports multiple readers through data valid bits, is the reason why our MoC explicitly supports this type of communication.

4.3 Architecture Modeling

The MXP5800 architecture platform can be modeled in Metropolis by using processes, media, and quantity managers in the Metropolis Metamodel. A single ISP is modeled as shown in Figure 5. The rectangles in the diagram represent tasks, the ovals represent media, while the diamond is the quantity manager. The ISP contains the Huffman hardware accelerator, and is sufficient for implementing the JPEG encoder application. If we want to use the DCT hardware accelerator, another ISP is needed. In this case, the two ISPs will be connected through Quad Ports. Modeling various ISPs with different hardware accelerators is very similar, we use the diagram in Figure 5 as an example.

Each PE is modeled as a medium, which supports multiple tasks running on it. Each task is modeled as a process.

Process	Hardware	Balanced	OPE Emph.	OPE Heavy
Level Shift	IPE	IPE	IPE	IPE
Add4-R	DCT HW Accelerator	OPE	OPE	OPE
Sub4-R		OPE	OPE	OPE
Add2-R		IPE	OPE	OPE
Sub2-R		IPE	OPE	OPE
Mult1-R		MACPE1	MACPE1	MACPE1
Mult2-R		MACPE2	MACPE2	MACPE2
Add4-C		IPE	IPE	OPE
Sub4-C		OPE	OPE	OPE
Add2-C		IPE	IPE	OPE
Sub2-C		OPE	OPE	OPE
Mult1-C		MACPE1	MACPE1	MACPE1
Mult2-C		MACPE2	MACPE2	MACPE2

Table 2. Mapping assignments

These processes represent the possible functionality executed on the PE. After mapping, the behavior of each task will be restricted by the corresponding process from the application; there is a one-to-one correspondence between architecture processes and application processes.

The scheduling of multiple tasks on a single PE is carried out by the quantity manager connected to this PE. The quantity managers support static scheduling, which is required by the MoC.

The communication between programming elements occurs through either global registers or local SRAM. The global register file is modeled as a netlist that contains 16 global registers. Each global register is modeled as a medium, which implements blocking write but allows multiple simultaneous reads. These global registers can be accessed by all PEs, hardware accelerators, and the MCH. To reduce the modeling complexity, an interface medium in the netlist is used to communicate with the PEs, accelerators, and MCH.

The SRAM is controlled by a memory command handler(MCH). The MCH contains a global register interface (GR interface), arbiter and memory. The GR interface is used to communicate with the global register file and is modeled as a process that waits for the appropriate data valid bits to be set in the global register file. The arbiter obtains memory access requests from the GR interface through multiple FIFO channels, then uses a round-robin access scheme to select one of them to access the local memory, which is modeled as a medium.

To model running time, a global time quantity manager is used. Every PE, every global register and the local memory are connected to it. Both computation and communication costs can be modeled by sending requests to this global time quantity manager and obtaining time annotations.

4.4 Design Space Exploration and Results

Given the application and architectural models in Metropolis as described in Sections 4.2 and 4.3 respectively, the design space can be explored by attempting different mappings between the application model and the architectural model. Each mapping scenario is specified in Metropolis with two types of information. The first is a specific set of synchronization constraints between the events in both models corresponding to the services that constitute the MoC. Along with these events – which represent the read, write, and execution services defined in our MoC – the parameters such as register location or local memory address can also be configured. The second is the set of schedules for the PEs that determine the execution order between the tasks. Both of these are relatively compact, meaning that new mapping scenarios can be created quickly and without modifying either the application or the architectural models.

The application is a total of 2,695 lines as shown previously in Table 1. The architectural model is 2,804 lines, while the mapping code is 633 lines. Each additional mapping scenario can be described with approximately 100 lines of additional code, and without modifying any of the other code.

To show the fidelity of our modeling methodology and mapping framework, we initially abstracted two mapping scenarios from the implementations provided in Intel MXP5800 algorithm library and carried out simulation in the Metropolis environment. We also tried an additional two scenarios which did not have a corresponding assembly language implementation. For all of the scenarios, only the mapping of the fine granularity 1D-DCT processes was varied.

The first scenario makes use of the DCT hardware accelerator and clearly has the highest performance. The other three scenarios use various software implementations of the row-wise and column-wise 1D-DCT operations. For these three scenarios, the transpose process is mapped to the MCH, which natively supports this type of operation. Register mappings are taken from the Intel library implementations and consist of 1, 2, or 4 global registers per FIFO channel. The second scenario uses a balanced partitioning of the processes among the available PEs, while the third and fourth scenarios put progressively more load on the OPE. The details for all four scenarios are provided in Table 2.

For each scenario, the number of clock cycles required to encode an 8x8 sub-block of a test image was recorded through simulation in Metropolis. For the first two scenarios, implementations from the MXP5800 library are available and were compared by running the code on a development board. The results are shown in Figure 6. The cycle counts reported with the Metropolis simulation are approximately 1% higher than the actual implementation since we did not support the entire instruction set for the processing elements. The latter two scenarios provide reasonable relative performance, but assembly implementations were not

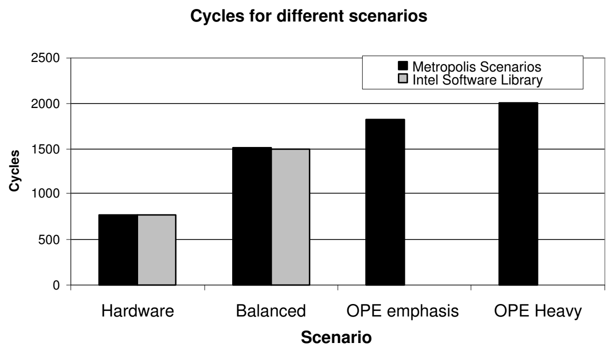

Figure 6. Performance Comparisons

available for comparison.

As long as the granularity of the each dataflow process is small (such as for most DSP-like systems), we expect that this model will provide very accurate estimates of performance. Regardless of the computational granularity, the schedules and deadlock analysis capabilities of this MoC will still remain valid.

5 Conclusions and Future Work

In this paper, we described a case study that makes use of the platform-based design methodology to enhance design space exploration. The main steps in the design flow are choosing the model of computation, modeling the application and architecture in a common framework using this MoC, and evaluating different mappings. In this paper, we illustrated how application/architecture separation, declarative synchronization statements and quantity annotation can be used within the Metropolis framework to realize the goals of the case study. Design space exploration can be carried out to analyze different mappings of the application on the architecture.

If the MoC chosen captures the important characteristics of the system, as shown, then accurate performance estimates can be obtained at a fraction of the cost and much faster than with other verification methods and tools. The main tradeoff when choosing a MoC, as described in Section 3, is between expressiveness and analysis capabilities. After determining that a particular MoC captures the main characteristics of a particular class of systems, automated design space exploration techniques can be developed.

6 Acknowledgments

This work was supported in part by a grant from the Consumer Electronics Group of Intel Corporation. We would like to thank Arup Gupta and Saurin Shah of Intel Corp. for their support and guidance throughout this project.

References

[1] Independent jpeg group, http://www.ijg.org.

[2] F. Balarin, J. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of HLDVT'01*, page 129. IEEE Computer Society, 2001.

[3] F. Balarin, L. Lavagno, and et al. Concurrent Execution Semantics and Sequential Simulation Algorithms for the Metropolis Metamodel. *Proc. 10th Int'l Symp. Hardware/Software Codesign*, pages 13–18, 2002.

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, 2003.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow. In *Proc. ICASSP'95*, volume 5, page 3255, Detroit, USA, 1995.

[6] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming*, 2003.

[7] G. Kahn. The Semantics of a Simple language for Parallel Programming. In *Proceedings of IFIP Congress*, pages 471–475. North Holland Publishing Company, 1974.

[8] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[9] E. Lee and T. Parks. Dataflow Process Networks. In *Proceedings of the IEEE, vol.83, no.5*, pages 773 – 801, May 1995.

[10] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, Dec. 1998.

[11] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[12] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 159–168. ACM Press, 1982.

[13] G. K. Wallace. The JPEG still picture compression standard. 34(4):30–44, Apr. 1991.

[14] Z. Wang. Fast algorithms for the discrete w transform and for the discrete fourier transform. In *IEEE Transactions on Acoustics, Speech, & Signal Processing*, volume ASSP-32, pages 803 – 816, August 1984.

[15] T. Wiegand, G. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.