Cross Layer Design to Multi-thread a Data-Pipelining Application on a Multi-processor on Chip

Bo-Cheng Charles Lai
EE Department
UCLA
CA 90095-1594
bclai@ee.ucla.edu

Patrick Schaumont
ECE Department
Virginia Tech.
VA 24061
schaum@vt.edu

Wei Qin
ECE Department
Boston University
MA 02215
wqin@bu.edu

Ingrid Verbauwhede
EE Dept. UCLA, CA
and
ESAT, K.U.Leuven, BE
Ingrid@ee.ucla.edu

ABSTRACT

Data-Pipelining is a widely used model to represent streaming applications. Incremental decomposition and optimization of a data-pipelining application onto a multi-processor platform spans multiple design layers, including the application layer, the system software layer, the architecture layer and the micro-architecture layer. For best results, designers have to consider multiple design layers (vertical exploration) and multiple architecture options (horizontal exploration). By using a data-pipelining JPEG encoder as the application driver, this paper presents a comprehensive analysis of mapping a data-pipelined application through multiple design layers, to a shared-memory SMP (Symmetric Multi-Processing) system. It is shown that a single-layered optimization ends up with a 110% worse design if the system effects from other layers are not taken into account. Compared to the nominal case, with appropriate mapping of the application, we achieve 47.5% improvement for high performance design and 77.6% energy reduction for energy efficient design under constant performance.

1. INTRODUCTION

Multi-processor systems offer superior performance as well as better energy-efficiency than single-processor systems [1][2]. Future systems are foreseeable to consist of multiple processing cores on a chip (MPSoC). In order to take advantages of the parallel computation capability provided by a multiprocessor system, applications are decomposed into multiple threads, which can be executed concurrently by the processors in the system.

Data-pipelining[3] is a common model to represent data-streaming applications, such as multimedia applications. It consists of multiple actors, where each actor performs one sub-task of the application. Actors are communicating through intermediate storage elements, similar to the pipeline stages in modern computer architectures. Multiple actors can be executed concurrently whenever there is enough input data for actors. Thus the data-pipelining model can be easily transformed into a multi-threaded program by initiating each actor as a thread. However, designing and mapping a data-pipelining-based application onto a multi-processor system is not a trivial task. It all depends on how well an application can be mapped through design-layers to the underlying multi-processor architecture. In order to reach the requirements such as high performance, real-time response and energy efficiency, mapping an application on a multi-processor system requires a tightly coupled design process between design-layers, including application software, system software, to hardware architecture (Fig.1). Most system optimizations do not reside on a single design-layer but instead are addressed over multiple layers. Therefore the design involves not only a solid understanding of the application, but also a thorough analysis of the system architecture over several design-layers.
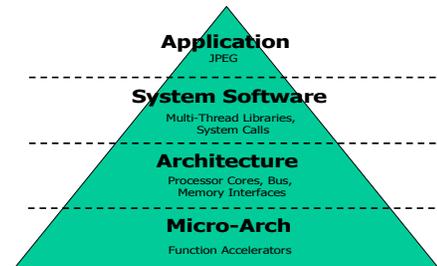
Fig.1: A design pyramid of a multi-processor system

This paper demonstrates a comprehensive analysis of mapping data-pipelining applications through multiple design layers, to a shared-memory SMP (Symmetric Multi-Processing) system[5]. The SMP system consists of multiple processors which execute the same instruction set. To achieve energy efficiency, processors can run at scaled operating frequencies and supply voltages. A data-pipelining-based JPEG encoder has been adapted as the application driver. By using a cycle-accurate multi-layered design environment[2], we demonstrate that a multi-layered vision is required to benefit the most from a MPSoC. Design optimizations focusing on a single design-layer might end-up as a 110% worse design if the system effects from other layers are not been taken into account. We also show that an appropriate mapping of the application to achieve the design criteria, e.g. high performance or energy-efficiency, can be found by looking at the design crossing multiple design-layers concurrently.

The paper is organized as follows. Section 2 addresses the previous work on data-pipelining models and cross-layered designs of multi-processor systems. Section 3 discusses the data-pipelining model used in this paper. By using the JPEG encoder as the application driver, Section 4 demonstrates cross-layer design examples of multi-processor systems. Section 5 will draw the conclusions.

2. RELATED WORK

The data-pipelining model [3] is a commonly used parallel programming model. The application is decomposed into processes. Each process is a sub-function of the application, and can potentially be executed concurrently. Processes are communicating through intermediate storage elements. This is similar to the dataflow computation model[6]. Thus it is very easy to program a dataflow application into data-pipelining model. The

Communicating Sequential Processes (CSP)[7] based programming model falls into a similar category. It exposes multiple threads of control at the source language level. Communication between processes is through fixed channels, and synchronization is achieved through use of blocking sends and receives.

Architectures have been specifically designed to implement these models [8][9]. Caspi et. al.[10] proposed the SCORE (Stream Computations Organized for Reconfigurable Execution) architecture, which is designed for a streaming multi-threaded model based on reconfigurable systems, such as FPGA (Field Programmable Gate Array) or CPLD (Complex Programmable Logic Device). This model-specific-hardware only supports dataflow type of applications and lacks capability to well-perform other types of models. However, modern multi-processor needs to be general enough to perform different types of parallel application models. Therefore the current MPSoC implementations from main micro-processor vendors[11][12][13] are based on SMP systems.

Many design frameworks have been proposed for SMP systems. However, the cross-layered design concept is not well supported in these frameworks. MPARM[14] returns cycle-accurate performance and energy consumption of a multi-processor system. It is based on SystemC which makes it possible to explore different numbers of processor cores. Using RTEMS [15] as the OS to handle multi-tasking makes it hard for designers to explore their own scheduling strategies. *StepNP™* [16] is a flexible domain-specific multi-processor architecture platform. It allows integration of a range of general-purpose to application-specific processor models. It supports several parallel programming models, however, it does not provide an environment to add custom hardware modules in the simulation framework. ARTS[4] is an abstract system level modeling framework to support the MPSoC designers in modeling the different layers and understanding their causalities. It embeds cross-layered design concept in the design framework. However, ARTS focuses on higher abstraction level and does not return cycle-accurate results. The importance of cross-layer design and optimizations starts to be recognized in other fields also. For instance, it is shown in [17] for energy efficient wireless communication.

This paper is among the first to conduct comprehensive analysis and demonstrate the cross-layered design for a MPSoC system. By using a software and hardware co-design environment, designers are able to explore the design space crossing multiple layers. It has been demonstrated in the paper that the multi-layered design is crucial in mapping a data-pipelining application to the underlying multi-processor architecture.

3. DATA-PIPELINING APPLICATION
3.1 Data-Pipelining Model
The data-pipelining computation model used in this paper is shown in Fig.2. There are three actors, A1, A2 and A3, which perform different data processing tasks for the same application. A1 and A2 are processing the data and "producing" the input data for A3. A3 will be executed and "consume" data once there are enough data in the intermediate queues Q1 and Q2. The intermediate queues implement a simple FIFO (First-In-First-Out) scheme.
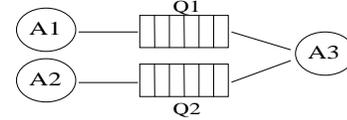
Fig.2: The data-flow computation model

3.2 Data-Pipelining JPEG Encoder
This paper uses a data-pipelining JPEG encoder as the application driver. Fig.3 shows three different data-pipelining implementations of the JPEG encoder. Each actor represents a sub-function of the JPEG. The actors are communicating through the intermediate queues which are located in the main memory of the multiprocessor systems. In the data-pipelining JPEG encoder, each actor is implemented as a thread and can be executed by any processor in the system. Ideally, if the intermediate queues have enough data to initiate the following "consumer" actors, all the actors can be executed concurrently. Fig.3(a) shows a design which makes every sub-function of the JPEG encoder as an actor. In Fig.3(b) and Fig.3(c), some actors are merged together. Therefore, in the application layer, Fig.3(a) has the highest parallelism, which is supposed to have the best performance in a multi-processor system. However, when we execute the JPEG on a four processor system, Fig.3(b) turns out to perform the best.
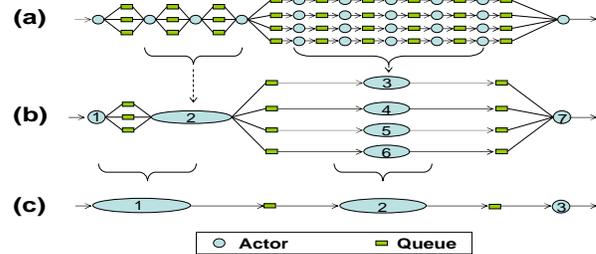
Fig.3: Data-pipelining JPEG encoder (a) original (b) the first actor agglomeration (c) the second actor agglomeration
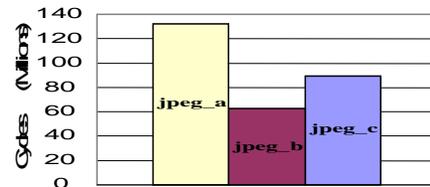
Fig.4: Execution cycles on a quad-processor system for the JPEG encoder implemented as in Fig.3(a) ~(c)

The execution cycles are shown in Fig.4. We use jpeg_a, jpeg_b and jpeg_c to represent the JPEG encoder implemented as in Fig.3(a), Fig.3(b) and Fig.3(c) respectively. The jpeg_a and the jpeg_c perform 110% and 29.8% respectively worse than jpeg_b. Compared with jpeg_b, jpeg_a has higher parallelism in the application layer. However, jpeg_a performs worse due to the overhead introduced in the system software layer and the hardware architecture layer. When an input queue does not have enough data to feed the subsequent thread, the thread has to yield and initiates a context switch to another idle thread. Too many context switches will hurt the system performance. The effect happens when we try to expose parallelism only at the application layer and have too many actors for the JPEG encoder. Exposing parallelism without looking at the other design layers hurts the performance as is clear from this example. This paper uses a

multi-layered cycle-accurate SMP design environment [2]. The design platform provides not only quick and cycle-accurate simulation results, but also allows designers to explore design options in multiple design-layers.

4. MULTI-LAYERED DESIGN EXAMPLE

Five optimizations (Table-1) have been applied, including optimizations residing in a single design-layer as well as optimizations crossing several design-layers. Please note that the examples shown here are only for the demonstration of the necessity of cross-layered design in a MPSoC system. The design environment can be expanded with many other types of cross-layer optimizations, because it is an open environment.

Example(1) – Actor agglomeration. The first optimization is to evaluate and decide an appropriate partitioning configuration of a data-pipelining application. As we can see from the example shown in Section 3, exposing the most parallelism in application layer does not always give the best performance. If required, several actors are grouped into larger actors to reduce overhead and improve performance. Fig.5 shows the execution cycle counts of the JPEG encoder on systems with different numbers of processors. Compared with jpeg_a, jpeg_b performs 27.5% to 52.6% better. The jpeg_c outperforms jpeg_a by 29.8% to 46%.

Table 1: Five design optimizations

	Optimizations	Method	Design-layer
1	Actor agglomeration	Merge actors and reduce context switches	Application layer
2	Proper number of processors	Find out appropriate number of processors	Architecture layer
3	Use on-chip memory	Map thread stack to on-chip memory	Crossing system software layer to architecture layer
4	Change energy-scale mode	Change power modes of processors to achieve energy-efficient execution	Crossing application layer to architecture layer
5	Custom the thread scheduling	Assign the bottleneck actor to a faster processor	Crossing application layer to Architecture layer

Example(2) – Determine proper number of processors. The second optimization is to find out the most appropriate number of processors for an application. Of course, more processors provide higher parallel computation capability. However, synchronization is required between processors. More processors in a system results in more synchronization traffic on the central bus interconnect, which might turn out to be the bottleneck of the system. If we compare jpeg_c running on different processor schemes, a dual-processor system achieves the best performance of 40.1% faster than a single-processor system. When running on a tri-processor system, jpeg_c runs at 12.1% slower than a dual-processor system. This is due to the extra overhead of synchronization between processors which compromises the parallel computation capability provided by more processor cores.

Example(3) – Use on-chip memory. Examples (1) and (2) have focused only on a single design-layer. But multi-processor system designs require concerns crossing several design-layers. Example (3) tries to map thread stacks on an on-chip memory to save long latencies of accessing off-chip memory. It crosses design-layers from the system-software layer to the architecture layer. Designers can allocate thread stacks to the on-chip memory block. Therefore, accessing thread stacks will be translated as accessing

the on-chip memory, which has shorter latencies than an external memory. Fig.5 shows that by mapping thread stacks of jpeg_b to the on-chip memory (jpeg_b_ex3) can further enhance the performance from 5.5% to 12.6%.

Example(4) – Change energy-scale mode. Energy efficiency is one of the crucial factors to evaluate a modern multi-processor system. It can be achieved by using energy-scaled designs. We assume a processor has two power modes, high-power mode (fast mode) and low-power mode (slow mode). Previous designs (examples 1, 2 and 3) focus on optimizing system performance. Here we try to achieve an energy-efficient design. Fig.6 shows energy consumption and execution cycle counts of jpeg_b on different energy configurations. The labels on x-axis represent the energy configurations of a multi-processor system. For example, 2HL represents a dual-processor system with one processor in high-power mode and the other one in low-power mode. It shows that jpeg_b_3HHH achieves the fastest execution and jpeg_b_1L has the lowest energy consumption. Designers can easily explore the design space for energy-efficient designs.
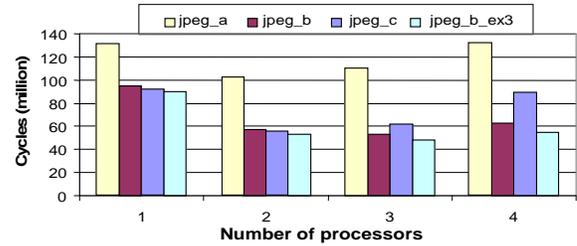
Fig.5: Execution cycles of the JPEG.
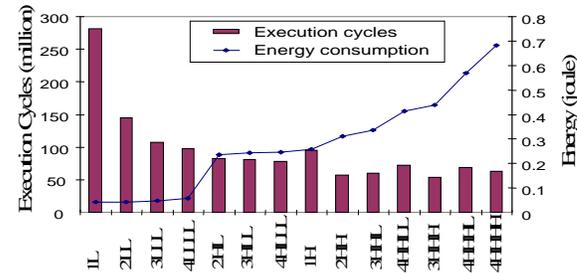
Fig.6: Energy consumption of jpeg_b for different energy-scaled schemes

Example(5) – Custom the thread scheduling. The basic thread library uses a simple FIFO (First-In-First-Out) scheduling scheme. The first thread in the queue will be executed by one of the idle processors. In the application dataflow shown in Fig.3(b), we can see that actor 3 to 6 have to wait for actor 2 to be completed before they can start execution. In an energy-scaled multi-processor system, if actor 2 is executed by a slow processor (low-power mode), it will take longer to complete actor 2, which delays the execution of actor 3 to 6. Therefore, we would like to assign actor 2 to a fast processor (high-power mode) if there is any available. To achieve this, the system requires a crossing-layer optimization. In the application layer, a designer has to find out the bottleneck actor thread, which is actor 2 in this case. In the system software layer and the architecture layer, this actor thread should be specified and scheduled to be executed only by a fast processor. Fig.7 shows the execution cycle counts after applying the custom schedule, and the performance is enhanced up to 19.7%.
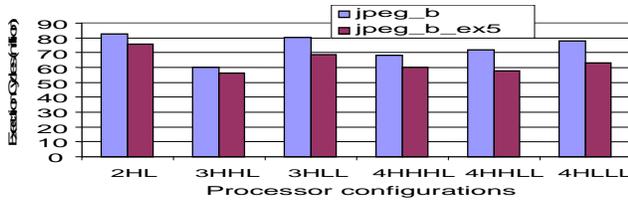
Fig.7: Performance for example(5)

Design space exploration. Fig.8 shows a sample scope of the design space that can be covered with the SMP architecture. We only choose a few design points to make the figure readable. We use different labels to indicate different configurations. For example, jpeg_c_ex3_2HL represents that the JPEG encoder implemented as in Fig.3(c) after the optimization described in example(3) is running on a dual-processor architecture with one processor running at high-power(H) mode and the other one at low-power(L) mode. Because jpeg_a performs too much worse (30%~50%) than the other cases, we choose jpeg_c_1H as the nominal case to compare with. It is represented by design point A. Fig.8 allows a designer to choose an optimal system configuration and optimizations based on the specification requirements. If a design, for example, has a cycle budget of 100 million cycles, then point H (jpeg_b_ex3_4LLLL) would be the most energy-efficient. It achieves 77.6% energy reduction and only 3% slower than the nominal design case at point A. Design point F (jpeg_b_ex3_3HHH) achieves the fastest execution of 47.5% faster than point A. Point C (jpeg_c_ex3_1L) consumes the lowest energy of 84% less than point A. None of these cases can be found by designing the system in a single design-layer.
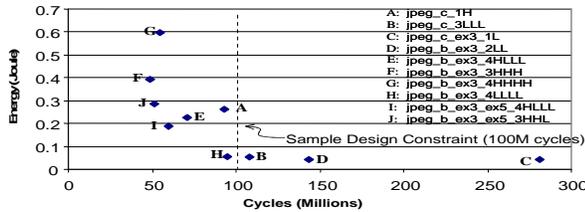
Fig.8: Sample scope of the design space covered with SMP-JPEG architecture

5. CONCLUSIONS

Design of a multi-processor system includes a thorough analysis and understanding of the system architecture crossing different design-layers. Focusing on a single design-layer might end up as a sub-optimal solution if system effects from other design-layers are not taken into account. By using a JPEG encoder as the application driver, this paper demonstrates examples of cross layer designs for data-pipelining applications.

We use five examples to demonstrate that a multi-layered design is required to get the maximum benefit from a multi-processor system. There are some important lessons designers should learn from these examples. First, a single-layered optimization might end up as a worse design. As shown in example(1), jpeg_b performs 27.5% to 52% better than jpeg_a, even though the jpeg_a has higher parallelism in application layer. Extra context switches and memory accesses in system software layer and architecture layer have compromised the parallelism exposed in application layer. Second, since design layers of a MPSoC are tightly coupled, many optimizations do not reside in only a single design layer. Instead it spans multiple layers and co-design

between layers is required to enable the optimizations. Examples (3), (4), and (5) have demonstrated this point. Third, the development tools can no longer focus on a single design layer. Rather it should make design options transparent cross design layers so that designers can understand the real impact of each design options on the whole system, and make the right decision. Fourth, although it requires cross-layered concerns, the optimizations at higher design layers usually result in more significant impact on the system. Evidence can be found by comparing the performance enhancement of the examples. Optimizations in examples (1) and (2) improve the overall performance more than those in examples (3) and (5). Last but not the least, cross-layered design of a MPSoC requires a fast and accurate design framework, which not only simulate the system, but also enables designers to apply(or remove) different optimizations in different layers. The design environment used in this paper provides an integrated co-design environment to enable cycle-accurate multi-layered designs, and the capability to quickly explore the design space of a multi-processor system.

6. ACKNOWLEDGEMENT

7. REFERENCES

[1] L.Hammond, et. al., "A Single-Chip Multiprocessor," IEEE Computer, Volume 30, No.9, pp.79-85, 1997.

[2] P. Schaumont, B.C. Lai, W. Qin, I. Verbauwhede, "Cooperative multi- threading on embedded multi-processor architectures enables energy-scalable design," Proc. 2005 DAC, pp. 27-30, June 2005.

[3] Luís Moura e Silva, Rajkumar Buyya, "Parallel Programming Models and Paradigms," in R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems: Volume 2", Prentice Hall PTR, NJ, USA, 1999.

[4] S. Mahadevan, K. Virk, J. Madsen, "ARTS: A SystemC-based Framework for Modelling Multiprocessor Systems-on-Chip," *Design Automation of Embedded Systems*, 2006

[5] D. Culler, J.P. Singh, A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann, 1999, ISBN 1-55860-343-3

[6] J.B.Dennis, "Data flow supercomputer," Computer, Vol.13, pp.48-56, Nov.1980.

[7] C.A.R.Hoare, "Communicating Sequential Processes," International Series in Computer Science, Prentice-Hall, 1985.

[8] B.S.Ang, Arvind, and D.Chiou, "StartT – The Next Generation: Integrating Global Caches and Dataflow Architecture," Technical Report 354, Laboratory for Computer Science, MIT, Aug.1994.

[9] R.A.Iannucci, "Toward a dataflow / Von Neumann Hybrid Architecture," Proc. 15th Symp. Computer Architecture (ISCA-15), pp.131-140, May 1990.

[10] E.Caspi, A.DeHon and J.Wawrzynek, "A Streaming Multi-threaded Model," Third Workshop on Media and Stream Processors, 2001.

[11] Intel Corp., http://www.intel.com

[12] Advanced Micro Devices, Inc, http://www.amd.com

[13] ARM Ltd, http://www.arm.com

[14] M.Loghi, M.Poncino, L.Benini, "Cycle-Accurate Power Analysis for Mutliprocessor System-on-a-chip," GVLSI, pp.401-406, Apr. 2004.

[15] RTEMS Home page, http://www.rtems.com/

[16] P. Paulin, et. al.," Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management," CODES+ISSS'04, pp.48-53, Sept. 2004.

[17] W. Eberle, et. al., "From myth to methodology: Cross-layer Design for Energy-Efficient Wireless Communication," DAC, 2005.